

Klassisches Unix

1. Einführung
2. Geschichte
3. Systemaufbau
4. Dateisystem
5. Dateisystem intern
6. Prozesse
7. Kommunikation zwischen Prozessen
8. Sockets

1. Einführung

Die erste wirklich bekannte Betriebssystem namens Unix war die 7. Ausgabe von dem Unix System, dass die Firma Bell Telephone 1978 herausbrachte. Einige Strukturen und Konzepte aus dieser Unix Version haben ihren Weg bis heute in die modernen Unix Ableger gefunden.

Der Umfang von Unix macht eine umfassende Beschreibung in diesem Vortrag nicht möglich und für viele Details sei der Leser auf die Literatur zu dem Thema verwiesen (zum Beispiel beim O'Reilly Verlag). An einigen Beispielen sollen daher einige Konzepte gezeigt werden.

Schon 1974 hatten Richie und Thompson die folgende Liste von Designvorgaben für Unix veröffentlicht:

1. Eine einheitliche Schnittstelle für Dateien, Geräte und Kommunikation zwischen Prozessen.
2. Ein hierarchisches Dateisystem mit Laufwerken, die dynamisch an- und abgemeldet werden können.
3. Die Möglichkeit für Prozesse andere Prozesse asynchron zu starten.
4. Eine für jeden Benutzer frei wählbare Benutzeroberfläche (Shell).
5. Hunderte von Programmen und Werkzeugen inklusive einem Duzend Programmiersprachen.
6. Eine hohe Portierbarkeit durch die Verwendung der Programmiersprache C anstelle von Assembler Code.

Die folgenden Abschnitte zeigen die Umsetzung der ersten drei Punkte dieser Liste.

2. Geschichte

Entnommen der Webseite http://www.cl.uni-heidelberg.de/kurs/ws01/zyklus/unix_geschichte.mhtml.

1965 - 1969

Die Bell Telephone Laboratories starten gemeinsam mit der General Electric Company und dem Projekt MAC des Massachusetts Institute of Technology ihre Bemühungen das neue Betriebssystem Multics zu entwickeln. Die Ziele dieses Projekts waren es, ein Betriebssystem (OS) zu entwickeln, dass es einer grossen Gruppe von Benutzern erlaubt gleichzeitig an einem Rechner zu arbeiten, das allen den Zugriff auf einheitliche Rechenpower und einheitlichen Speicher ermöglicht und das es den Benutzer erlaubt einfach auf gemeinsame Daten zu zugreifen.

1969 - 1971

Obwohl eine einfache laufende Version von Multics erstellt werden konnte, erfüllte es weder die angestrebten Ziele, noch konnte abgesehen werden, wann es diese erfüllen würde. Deshalb beendeten die Bell Laboratories ihre Teilnahme an dem Projekt. Zahlreiche Mitarbeiter dieses Projekts arbeiteten später beim Entwurf von Unix mit.

Mitarbeiter des Computer Science Research Center der Bell Labs - darunter Ken Thompson und Dennis Ritchie - erarbeiteten einen Entwurf fuer ein Filesystem, das ihre Programmierumgebung verbessern sollte. Dieses Filesystem entwickelte sich später zur ersten Version des Unix-Filesystems. Thompson programmierte Programme, die das Verhalten dieses Filesystems und der Programme, die darauf zugreifen, simulierten. Ausserdem entwickelte er einen kleinen Kernel fuer den GE 645 Computer. Zur gleichen Zeit entwickelte er in Fortran auf einem GECOS-System das Computerspiel Space Travel. Das Programm funktionierte aber nicht wie gewünscht. Vor allem die Steuerung des Spaceships gestaltete sich schwierig. Auch war das Programm äusserst langsam. Später fand Thompson einen wenig genutzten PDP-7 Computer, der eine gute Anzeige hatte und fuer relativ wenig Geld eine hohe Rechenleistung lieferte. Allerdings war es sehr Aufwendig fuer den PDP-7 zu entwickeln, weil man erst auf dem GECOS-System den PDP-7 Assembler erzeugt werden musste und dann den PDP-7 mit den gewonnen Lochbänder füttern musste. Deshalb implementierten Thompson und Ritchie ihr Systemdesign fuer den PDP-7 und integrierten hier auch ihre frühe Version der Unix Filesystem, eine Prozess-Subsystem und eine kleine Sammlung von Werkzeugen. So benötigte das neue System nicht länger das GECOS-System als Entwicklungssystem, sondern könnte sich selbst weiterentwickeln. Das neue System nannte Brain Kernighan - Multics veralbernd - UNIX.

1971 - 1973

Obwohl diese frühe Version von UNIX schon einiges hielt was Multics versprochen hatte, konnte es sein vollen Potential nicht entfalten, bevor es nicht zu einem offiziellem Projekt gemacht wurde. Das UNIX wurde bereits in der Patentabteilung der BELL Labs als Textprozessor eingesetzt, als es auf die PDP-11 portiert wurde. Das System hatte zu diesem Zeitpunkt folgende Eigenschaften:

- 16 Kb System
- 8 Kb fuer User-Programme
- 512 Kb Festplatte
- 64 Kb maximale Dateigröße

Nach diesen frühen Erfolgen machte sich Thompson an die Arbeit Fortran auf das neue System zu portieren. Was aber dabei herauskam war die Programmiersprache B, die von der Sprache BCPL beeinflusst war. B war eine Interpretersprache und hatte somit die typischen Performanz-Probleme solcher Sprachen. Deshalb entwickelte Kernighan B weiter. Das Ergebnis nannte er C. C konnte in Assembler übersetzt werden, man konnte Datentypen deklarieren und neue Datenstrukturen definieren.

1973 - 1977

Man entschied sich dazu UNIX neu in C zu schreiben, was damals ein ungewöhnlicher Schritt war, die Akzeptanz von UNIX aber deutlich erhöhte. Die Anzahl der UNIX-Installationen in den Bell Labs wuchs auf 25, weswegen fuer den internen Support die UNIX System Group gegründet wurde.

Zu dieser Zeit durfte AT&T - aufgrund des Consent Decree von 1956 - keine Computer-Produkte vermarkten. UNIX wurde aber fuer Universitäten verfügbar gemacht, die einen grossen Bedarf an einem Betriebssystem für Lehrzwecke hatten.

1977 - 1982

UNIX wird immer populärer. Die wachsende Anzahl von Microprozessorsystemen und das saubere, elegante Design von UNIX bewegen viele Entwickler ihre eigenen Varianten zu entwickeln. Die Anzahl der Installationen beläuft sich auf über 500, davon ca. 125 bei Universitäten. UNIX-Systeme sind vor allem bei Telefongesellschaften beliebt. Interactive System Cooperation waren die ersten, die eine Kommerzielle Lizenz von einem um Office Automations Anwendungen erweitertes UNIX vertrieben. Ausserdem wurde UNIX das erste mal auf eine andere Plattform als PDP portiert: Interdata 8/32

Mit der wachsenden Popularität von UNIX portierten immer mehr Hersteller das System auf ihre Maschinen. AT&T verbanden mehrere Varianten in einem neuem System, das UNIX System III genannt wurde. System III wurde bei AT&T weiter entwickelt und wurde so zu System V.

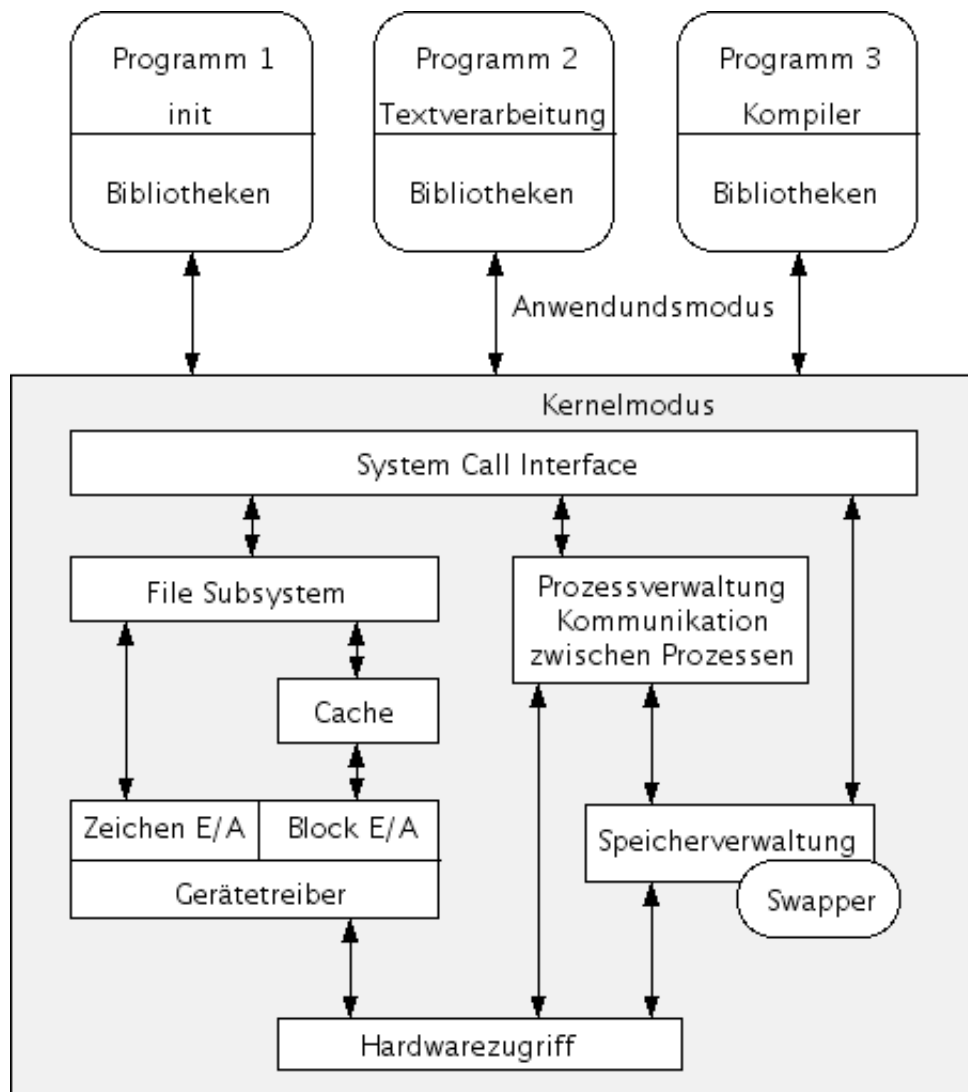
1983

AT&T bieten kommerziell Support fuer System V an. Gleichzeitig haben Leute an der University of California und Berkley eine eigene Variante für VAX entwickelt, die einige interessante neue Eigenschaften hatte.

3. Systemaufbau

Unix ist ein Kernel Betriebssystem mit dem mehrere Benutzer in verschiedenen Programmen gleichzeitig arbeiten können. Das System ist dabei in die zwei Ebenen Kernel und Programme geteilt. Das Kernel hat dabei den alleinigen den Zugriff auf die Geräte (Hardware) und entscheidet über die Vergabe von Ressourcen wie zum Beispiel der CPU Zeit. Außerdem stellt das Kernel sicher, dass Prozesse derart voneinander getrennt sind, dass kein Prozess einen anderen zum Absturz bringen kann.

Üblicherweise greifen Programme auf eigene Funktionsbibliotheken zu. Diese Bibliotheken, die in der Regel für die jeweilige Programmiersprache Funktionen anbieten, greifen selber auf das so genannte System Call Interface zurück. In diesem Interface sind alle öffentlich zugänglichen Kernelfunktionen gesammelt. Die Funktionen im Interface werden im Kernel Modus aufgeführt und prüfen erstmal die Parameter. Zur Erfüllung der Anfragen greifen sie auf einige Subsysteme zu wie das Datei Subsystem, die Speicherverwaltung oder die Interprozesskommunikation.



Die Kernelmodule bei Unix als Grafik

Im folgenden gehen wir die verschiedenen Subsystem einmal durch:

- Das System Call Interface über das Programme und ihre Bibliotheken auf das Kernel zugreifen.

- Die Module für Zeichen- und Blockorientierten Ein- und Ausgabegeräte als Teil des Dateisystems.
- Den Dateicache, der alle Dateizugriffe puffert um bei wiederholten Zugriffen schneller Daten zu liefern oder mögliche Zukünftige Zugriffe voraus berechnet und optimiert.
- Das Dateisystem, das den UNIX Namenbaum verwaltet, in dem hierarchisch alle Dateien, Ordner und Geräte verzeichnet sind. Alle Einträge in diesen Namenbaum werden über die gleiche Schnittstelle angesprochen, so dass das Dateisystem erst einmal feststellen muss, welches Gerät bzw. welcher Gerätetreiber wirklich zuständig ist.
- Das Speichersubsystem, das den physikalischen Speicher verwaltet und an die Programme verteilt.
- Die Prozessverwaltung, die Prozesse startet und terminiert, die CPU Zeit vergibt und die Kommunikation zwischen Prozessen steuert.

Zum System Call Interface:

Alle Systemfunktionen aus dem Unix Kernel werden von Programmen als ganz normale Funktionsaufrufe aufgerufen. Dabei wechselt dann die CPU den Status, denn das Kernel läuft in einem Modus der Hardwarezugriffe direkt erlaubt, was einem einzelnen Programm nicht gestattet wird.

Das Kernel wird bei Unix direkt aufgerufen und nicht durch kleine Botschaften benachrichtigt, wie es bei größeren Systemen nötig wird. Außerdem ist das Kernel in weiten Teilen nicht preemptiv, was bedeutet, dass während eine Kernelfunktion arbeitet eine andere Kernelfunktion nicht durch ein anderes Programm aufgerufen werden darf. Allerdings kann ein Programm natürlich eine Kernelfunktion aufrufen, die etwas länger dauert (zum Beispiel eine Datei lesen). Dann wird diese Funktion natürlich in einer Warteschleife stehen und in dieser Zeit erlauben, dass ein anderes Programm ausgeführt werden kann.

Diese Restriktionen stammen aus der Zeit als Unix für einen Prozessor entwickelt wurde und daher ein einfaches Design ausreichend war. Mit mehreren Prozessen oder sogar einem Wechsel zwischen den Prozessen muss Code ergänzt werden, der Zugriffskonflikte löst, die auftreten können, wenn in zwei Prozessen (durchaus jeder auf einem eigenen Prozessor) die Kernelfunktionen auf gemeinsame Kernelnaten zugreifen. Kritische Codesequenzen müssen dann auch vor einem Prozesswechsel geschützt werden.

Zur Geräteverwaltung

Unix unterscheidet zwei Arten von Geräten. Einerseits blockorientierte Geräte (z.B. Datenträger bzw. Festplatten) und andererseits Zeichenbasierte Geräte (z.B. Tastatur und Bildschirmkonsole).

Die Gerätetreiber sind zwar weitgehend in C geschrieben aber doch recht Hardware abhängig und teilweise in Assembler realisiert.

Interessant zu erwähnen ist, dass die blockorientierten Gerätetreiber eine Liste von Anforderungen verwalten. Alle Schreib und Lese Anweisungen aus den verschiedenen Programmen landen in einer Liste, die dann so sortiert wird, dass der Schreib/Lese Kopf sich möglichst wenig bewegen muss.

Zu dem Dateicache

Um die Blockorientierten Geräte schneller benutzen zu können, speichert ein Unix System die gelesenen Daten im Arbeitsspeicher sofern dieser nicht durch wichtigere Daten belegt wird. Greift ein Programm auf Daten zu, die vorher schon gelesen wurden, dann wird diese Anforderung direkt im Speicher ausgelesen und der Gerätetreiber wird nicht aufgerufen.

Bei dem Speicher, der für die Zwischenlagerung von Festplatteninhalten verwendet wird, muss durch geeignete Mechanismen der Zugriff koordiniert. So können zwei Programme gleichzeitig die gleichen Daten anfordern. In einem Prozess liest dann das Kernel von der Platte die Daten und in einem anderen Prozess wartet das Kernel bis es die Daten dem Cache entnehmen kann. Sollte ein Block

beschrieben werden, muss natürlich auch der Cache überarbeitet werden.

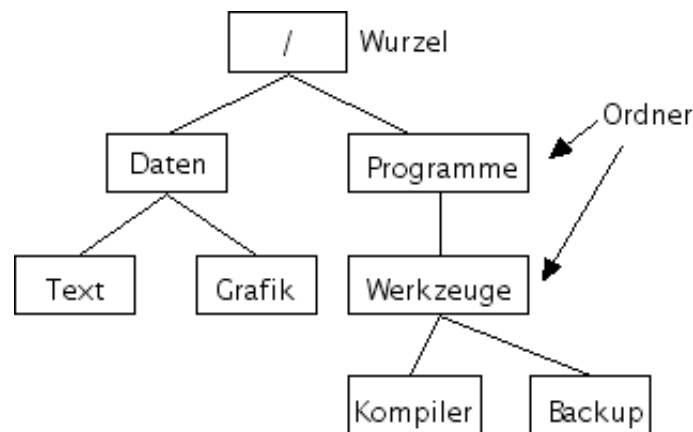
Durch den Cache kommt es zu möglichen Problemen:

- Ein Programm schreibt etwas auf die Festplatte und bekommt eine Bestätigung über den Erfolg dieser Operation, aber die Daten befinden sich nur im Cache und werden erst später geschrieben.
- Ein Programm kann seine Schreiboperationen in einem bestimmten Muster abschicken. Allerdings stört dabei die Tatsache, dass sich die Reihenfolge der Schreiboperationen anhand der Kopfbewegung ändern kann. Durch einen Crash entstehen möglicherweise Einträge in einem Inhaltsverzeichnis für Daten, die nicht geschrieben wurden.
- Während des später ausgeführten Schreibprozesses entsteht ein Fehler (z.B. Benutzer hat die Diskette schon entnommen). Um das Programm darüber zu informieren, werden alle Schreiboperationen für eine Datei spätestens dann erledigt, wenn die Datei geschlossen wird.

Außerdem enthält Unix Funktionen um explizit den Schreibcache zu leeren (Flush). Dies wird auch dann regelmäßig gemacht, wenn ein Benutzer sich abmeldet oder eine bestimmte Zeitspanne (zum Beispiel 30 Sekunden) abläuft.

4. Dateisystem

Das Unix Dateisystem ist ein streng hierarchisches Dateisystem. In einer Baumstruktur knüpfen Ordner als Äste und Dateien als Blätter an. Dateien sind in diesem Fall einfache Byteströme von denen man jederzeit jedes Byte lesen kann. Dies ist ein großer Unterschied zu Dateien anderer System aus dieser Zeit, die Dateien als eine Sammlung von Datensätzen gespeichert haben.



Das hierarchische Dateisystem von Unix.

Der Dateibaum kann auch spezielle Dateien enthalten, die nicht auf der Festplatte gespeichert sind. So gibt es einen Ordner /dev in dem Dateien liegen wie zum Beispiel den Kernel Speicher "/dev/kram", die erste Festplatte "/dev/disk0", die erste Partition auf der ersten Festplatte "/dev/disk0s1" oder das Terminal "/dev/console". Natürlich stehen diese Dateien nicht jedem Programm zur Verfügung. Aber ein Programm, was beispielsweise eine beliebige Datei in einem Backup auf Band speichern kann, könnte mit entsprechenden Zugriffsrechten auch eine ganze Festplatte sichern ohne dass das Programm geändert werden müsste.

Es gibt nur ein paar wenige Systemaufrufe für Eingabe- und Ausgabeoperationen. Beim Öffnen oder Erzeugen einer Datei gibt man neben dem Pfad an, ob man schreiben und/oder lesen möchte. Daraufhin bekommt man eine Nummer zugeteilt unter der das System Anfragen betreffend der offenen Datei beantwortet. Das System speichert dazu in einer Tabelle zu allen offenen Dateien einige Informationen wie die aktuelle Position in der Datei.

Die Funktionsaufrufe:

```
nr = open(Pfad, Modus) // Datei öffnen
Fehlercode = close(nr) // Datei schliessen
Anzahl Bytes = read(nr, Speicheradresse, Anzahl Bytes) // Lesen
Anzahl Bytes = write(nr, Speicheradresse, Anzahl Bytes) // Schreiben
Fehlercode = seek(nr, Neue Position) // Position ändern
Fehlercode = create(Pfad) // Neue Datei erzeugen
```

Beispiel:

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <sys/stat.h>

#define data "Hello World!"

int main (int argc, const char * argv[]) {
    // insert code here...
    printf("Only a small test application\n");

    int handle=creat("testfile.txt", S_IRWXU | S_IRWXG | S_IRWXO);
    printf("handle for file: %d \n",handle);

    int c=write(handle,data, strlen(data));
    printf("wrote %d bytes. \n",c);

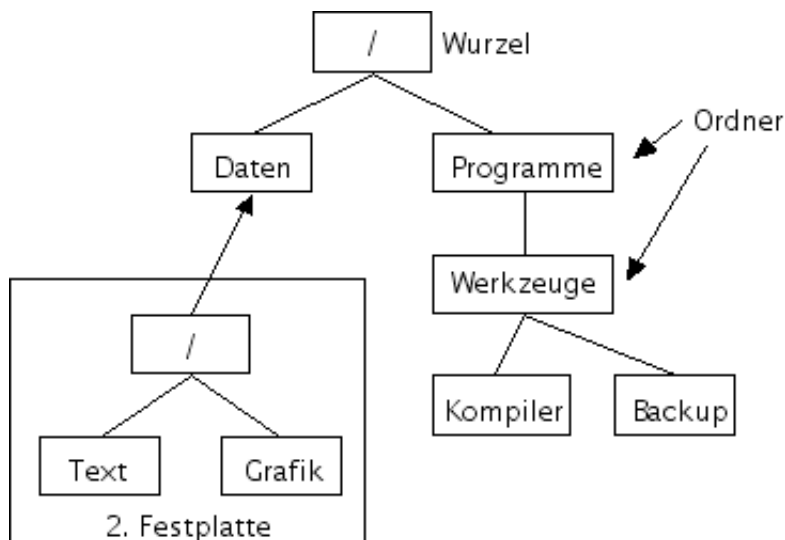
    int e=close(handle);
    printf("close returned: %d \n",e);

    return 0;
}
```

Mounten

Ein Unix Computer hat in der Regel eine Festplatte, auf der sich das Dateisystem befindet. Wenn eine zweite Festplatte hinzukommt (oder ein anderer Datenträger), dann wird dieser neue Datenträger mit seinem Wurzel im Dateisystem in den vorhandenen Baum eingehängt.

Zum Beispiel hängt der Befehl "mount /dev/disk1 /daten" die Wurzel des Dateisystems auf der 2. Festplatte an die Stelle von dem Ordner /daten in den Dateisystembaum ein. Die Datei mit dem Pfad "/text" auf der zweiten Festplatte hat anschliessend den Pfad "/daten/text" innerhalb des Unix Dateibaums.



Das Dateisystem mit einer in /Daten eingehängten zweiten Festplatte.

Navigation im Dateibaum

Da es recht umständlich ist immer mit absoluten Dateipfaden zu arbeiten, gibt es einen aktuellen Arbeitspfad. Das Kernel speichert folglich zu jedem Programm den aktuellen Pfad mit der Folge, dass Programme auf Dateien in diesem Arbeitspfad direkt zugreifen können ohne selber ihren Arbeitspfad zu kennen. Wenn der aktuelle Pfad `"/daten"` ist und ein Programm auf die Datei mit dem Pfad `"text"` zugreift, dann ergänzt das System diesen Pfad automatisch zu `"/daten/text"`. Im Gegensatz zu den relativen Pfaden beginnen absolute Pfade mit `"/"`. Alle Programme und der Benutzer können den aktuellen Pfad über den Befehl `chdir` ändern.

Die Shell, als die Kommandoschnittstelle zum Benutzer, kennt zudem noch einen Homepfad als den Pfad zu dem Ordner eines Benutzers, wo dessen persönliche Dateien liegen. Wenn der Benutzer sich anmeldet, dann befindet er sich innerhalb des Dateisystems in seinem Homepfad.

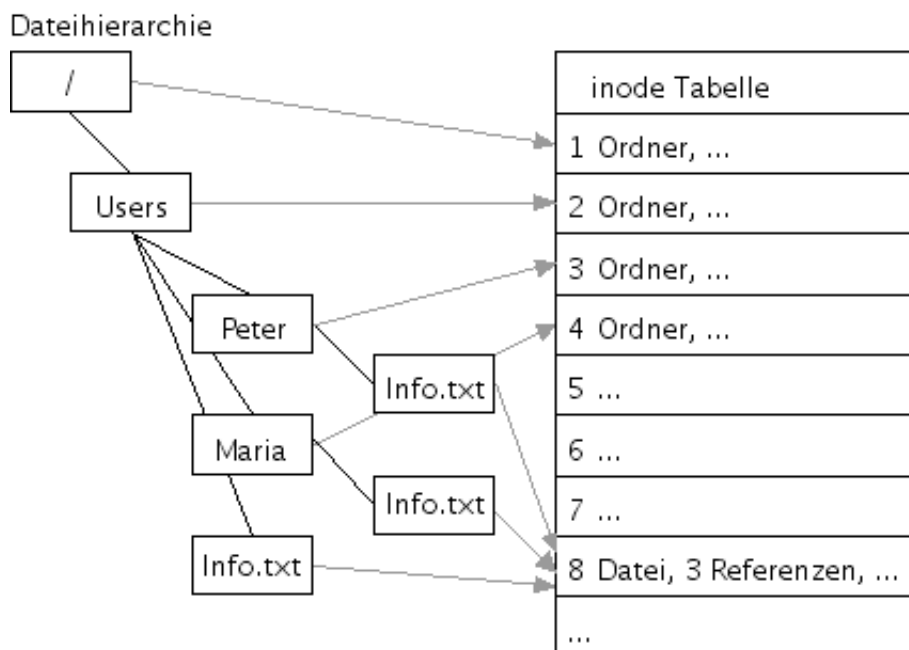
Zur Navigation im Dateibaum gibt es in jedem Ordner ausser der Wurzel die Ordner `"."` und `".."`. Der Ordner `"."` ist der aktuelle Ordner und `".."` der Ordner eine Stufe höher. Befinden wir uns also in `"/users/Max"`, dann zeigt der Pfad `"../Moritz/Text"` auf `"/users/Moritz/Text"`.

5. Dateisystem intern

Im Laufe der Zeit hatte jedes Unix seine Implementation des Dateisystems. Zum Beispiel UFS als das Unix File System, HFS Plus als das Mac OS X Dateisystem oder ext3 als ein aktuelles Linux Dateisystem. Neuere Unix Versionen können allerdings auch auf Formate anderer Systeme zugreifen und teilweise ist das Dateisystem für ein gegebenes Unix frei wählbar. Daher seien hier nur einige allgemeinere Charakteristika beschrieben.

inodes

Jedes Dateisystem speichert eine Liste, in der alle Objekte, die sich auf der Festplatte befinden, verzeichnet sind. In dieser Liste werden zu jeder Datei und jedem Ordner verschiedene Metadaten gespeichert. Die Einträge dieser Liste werden bei Unix inodes genannt und für den schnellen Zugriff im Arbeitsspeicher gehalten.



In dieser kleinen Ordnerhierarchie gibt es drei Ordner für eine Datei.

Zu den Metadaten eines inodes gehört unter anderem die Nummer des Besitzers (Owner ID), die

Nummer der Benutzergruppe (Group ID), Zugriffsrechte, das Erstelldatum, das Änderungsdatum, ein Referenzzähler, ein paar Flags und die Position der für dieses inode belegten Blöcke auf der Festplatte. Bei den Flags wird unter anderem gespeichert, ob dieses inode eine Datei, ein Gerät oder einen Ordner beschreibt.

Ordner

Inodes, die als Ordner markiert sind, enthalten als Inhalt eine Liste von Datensätzen, die Namen auf inodes abbilden. Ein inode kann dabei in mehreren Ordnern enthalten sein. Einen Link zwischen Name und inode nennt man Hard Link. Folglich werden die Metadaten zentral für eine Datei bzw. einen Ordner abgelegt, während die Datei selber unter verschiedenen Namen in verschiedenen Ordnern erreichbar ist.

Zugriffsrechte

Jede Datei (inode) hat einen Benutzer Nummer und eine Gruppen Nummer. Daher ist bekannt, wem die Datei gehört. Gespeichert wird jetzt zu einer Datei für die 3 Benutzer Besitzer, Gruppe und Andere ob diese Lesen, Schreiben und Ausführen können. So kann eine Datei mit dem Attribut "-rw-r-----" den Besitzer zum Lesen und Schreiben ermächtigen, seiner Gruppe das Lesen erlauben und alle anderen aussperren. Bei "-rwxrwxrwx" dürfen alle alles. Wenn jetzt noch das so genannte Set-User-Bit gesetzt ist, "-r-sr-sr-x" (zum Beispiel beim Ping Systemprogramm), dann wird beim Ausführen des Programms dieses Programm mit den Zugriffsrechten des Besitzer (bei Ping der Administrator) von der Programmdatei gestartet. Ähnlich zum Set-User-Bit gibt es auch noch ein Set-Group-Bit Flag.

Belegung der Blöcke beim Dateisystem

Im ersten Block einer Festplatte steht ein kleines Programm, was beim Starten des Computer kurz läuft und von der Festplatte das Betriebssystem lädt. Im zweiten Block, dem Superblock, werden Daten über die Festplatte gespeichert wie zum Beispiel die verschiedenen Größenangaben: Anzahl Blöcke, Anzahl belegter bzw. freier Blöcke und Größe der Blöcke. In den folgenden Blöcken befindet sich eine Liste der freien bzw. belegten Blöcke. Zum Schluss die Datenblöcke. Verschiedene Unix Versionen speichern diese Daten aus mehrmals auf der Festplatte zur besseren Wiederherstellung nach einem Systemausfall.

Dateisystem Daten im Speicher

Zu jeder Festplatte wird im Arbeitsspeicher den Superblock um schnellen Zugriff auf dessen Daten zu haben.

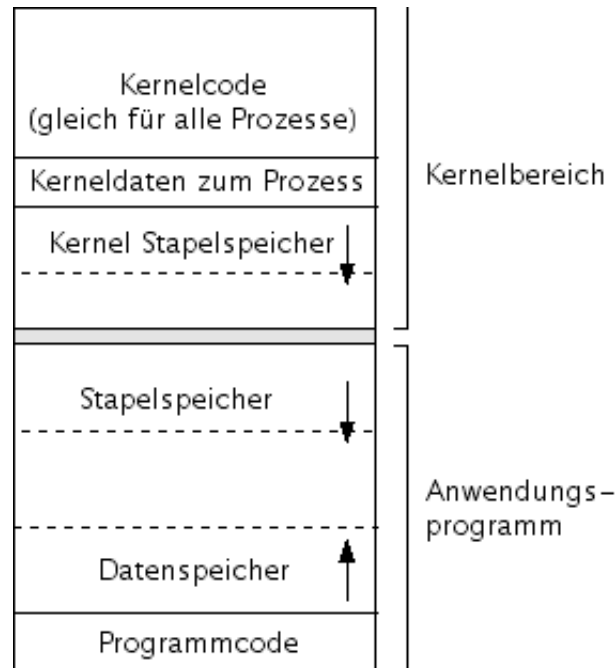
Außerdem existiert im Arbeitsspeicher eine Liste aller offenen Dateien mit einigen Statusinformationen (z.B. aktuelle Position in der Datei) und eine Kopie vom inode. Sollte sich was an dem inode ändern wird es beim Schliessen der Datei wieder auf die Festplatte geschrieben.

6. Prozesse

Die Prozessverwaltung in Unix kümmert sich um das Erzeugen und Terminieren von Prozessen. Außerdem verteilt sie die CPU Zeit auf die Prozesse.

Jeder Prozess hat seinen eigenen logischen Adressbereich, der bei einem Zugriff über passende Tabellen auf physikalischen Speicher abgebildet werden kann. Zugriff auf Speicherbereiche, der vorher nicht von der Speicherverwaltung angelegt wurden, erzeugen einen Fehler und führen zum Programmabbruch. Der logische Adressbereich ist in zwei Teile geteilt. Im unteren Bereich liegt der Programmcode, der Datenspeicher und der Stapelspeicher. Ganz oben ist in allen Programmen der Kernelcode eingeblendet. Darunter befinden sich die Kerneldaten, die zu diesem Prozess gehören sowie der Kernel Stapelspeicher für diesen Prozess. Die CPU wird so eingestellt, dass das Programm nicht in den Codeteil oder in die

Kernelbereiche schreiben kann. Sollte allerdings eine Kernelfunktion aufgerufen werden, dann wechselt die CPU den Modus und erlaubt den Zugriff auf alle Speicherbereiche. Beim Zugriff auf eine Speicheradresse prüft die Memory Management Unit (MMU) als Teil der CPU, ob auf die Adresse zugegriffen werden darf. Wenn ja schaut sie in einige Tabellen um von der logischen zur physikalischen Adresse zu kommen. Dabei muss möglicherweise die dort befindliche Speicherregion von der Festplatte geladen werden, wenn sie vorher ausgelagert wurde (virtueller Speicher).



Die Aufteilung des Adressraumes eines Prozesses.

Da die MMU einen Fehler meldet, wenn auf eine nicht vergebene Speicherseite zugegriffen wird, belegt man eine Speicherseite an der Adresse 0 um Zugriffe auf nil Pointer abzufangen. Außerdem wird zwischen Stapelspeicher und Datenspeicher und zwischen den Stapelspeichern vom Kernel und vom Programm eine Seite reserviert um einen Überlauf abzufangen.

Für die Entwickler von Programmen bieten höhere Bibliotheken Funktionen an um Speicher auf dem Heap zu reservieren. Zum Beispiel gibt es dafür in der C Laufzeitbibliothek die Befehle free und malloc.

Prozesse erzeugen

Die Kernelfunktion fork erstellt eine Kopie des aktuellen logischen Adressbereichs mit dem gesamten Inhalt. Der Aufrufende Prozess (Parent) bekommt dabei dann die neue Prozess ID zurück vorhin gegen der neue Prozess (Child) eine 0 zurück bekommt. Der Befehl execve lädt ein Programm und ersetzt das aktuelle Programm im prozess mit dem Code und den Daten aus der neuen Programmdatei. Wenn man fork und execve zusammen benutzt kann man relativ einfach ein neues Programm starten.

Das Parent Programm, also das ältere bei der Prozesserzeugung kann mit dem Befehl wait auf das Ende des Child Programmes warten. Dabei erhält es eine Zahl zurück, die das Child Programm mit dem Befehl exit beim Beenden abliefern.

Beispiel: top &

Prozesse starten beim Systemstart

Beim Systemstart wird das Kernel geladen. Das Kernel setzt einige Daten auf ihre Startwerte und lädt das Dateisystem. Anschliessend wird der Prozess 0 gestartet, der später als der Swapper weiterläuft um den virtuellen Speicher zu verwalten. Der Prozess 0 startet mittels fork den Prozess init (Prozess 1), der alle Terminals überwacht. Wenn ein Terminal frei ist, wird dort login gestartet. Bei einem erfolgreichen

Login wird dann die Shell geladen um Befehle vom Benutzer entgegen zu nehmen.

Prozesse starten von der Shell aus

Der Benutzer kann über eine passende Pfad eingabe einen Prozess in der Shell starten. Dabei erzeugt die Shell mit fork einen neuen Prozess und führt darin das neue Programm aus. Wenn beim Programmstart ein "&" am Ende des Befehls steht, dann wartet die Shell nicht auf das Ende des Programms, sondern nimmt direkt neue Befehle entgegen.

CPU Scheduling

Unix benutzt eine dynamische Verteilung der CPU Zeit nach Bedarf. Jeder Prozess kann vom Benutzer bzw. vom Administrator der Rechneranlage eine bestimmte Priorität bekommen. Allerdings bekommt ein Prozess, der gerade eine Kernelfunktion ausführt eine höhere Priorität, da das Kernel bekanntlich nicht redundant aufgerufen werden darf (non preemptive). Prozesse die auf einen Event warten (zum Beispiel auf das Ende eines Lesevorgangs) bekommen, wenn dieser Event eintritt, eine höhere Priorität um auf den Event zu reagieren. Da Prozesse nicht benutzte Zeit abgeben können, bekommen wird die Priorität derart angepasst, dass Großverbraucher tendenziell mehr CPU Zeit bekommen.

Der Swapper

Wenn zu wenig physikalischer Speicher verfügbar ist, dann prüft der Swapper wie er neuen Speicher beschaffen kann. Speicherseiten, die zum Beispiel aus einer Programmdatei geladen wurden und länger nicht mehr benötigt wurden, können direkt freigegeben werden (zum Beispiel der Startcode aus einem Programm). Selten benutzte Speicherseiten können auch auf die Festplatte ausgelagert werden. Vorzugsweise werden dafür natürlich Seiten genommen, die schon auf der Festplatte stehen, so dass eine erneutes Schreiben unnötig wird. Bei einem Zugriff auf eine ausgelagerte Seite wird diese dann wieder eingelesen und eventuell vorher dafür Speicher frei gemacht.

Im ersten Swapper wurden ganze Prozesse ausgelagert, was allerdings recht viel Zeit in Anspruch nimmt. Daher werden ausgelagerte Prozesse für mindestens 3 Sekunden ausgelagert um die Effizienz zu erhöhen. Moderne Unix Versionen lagern den Speicher in festen Blockgrößen aus (zum Beispiel 4 KByte).

7. Kommunikation zwischen Prozessen

Klassisches Unix bietet zwei Kommunikationswege für Prozesse, die auf einem Computer laufen: Erstens Pipes (Rohre) und zweitens Signale. Eine Pipe ist ein Byte Stream in eine Richtung. Ein Prozess kann Daten schreiben und ein andere liest während das System mit einem Puffer dazwischen ausgleicht und eventuell auch die Prozesse blockiert bis der andere liest oder schreibt.

Beispiel: `cat "Proseminar Unix.txt" | grep "Swapper" | less`

Das Programm cat schreibt den Inhalt der Textdatei in die Pipe. Grep liest aus der Pipe und sucht darin alle Zeilen mit dem Wort "Swapper" und schreibt diese in die zweite Pipe. Dort liest less das Ergebnis und gibt es zeilenweise im Terminal aus.

Alle Programme die Daten von der Tastatur über die Datei stdin lesen bzw. über stdout auf den Bildschirm schreiben, können mit Pipes benutzt werden.

Signale:

Signale sind asynkron ausgeführte Events. Beispielsweise gibt es das Signal namens SIGCHILD. Es wird

immer dann benutzt, wenn ein Childprozess endet. Für jedes Signal kann ein Programm einen Eventhandler installieren, der dann durch das System aufgerufen wird.

Einige der wichtigeren Signale sind:

- Prozesse werden über das SIGCHLD Signal informiert, wenn ein von ihnen gestarteter Prozess beendet wird.
- Fehler im Programm werden durch Signale gemeldet, wie eine Division durch 0 (SIGFPE) oder das Schreiben in eine Pipe ohne das jemand am anderen Ende horcht (SIGPIPE).
- Zugriffe auf ungültiger Speicheradressen führen zu einem SIGSEGV (Segmentation violation).
- Signale von der Prozessverwaltung um zum Beispiel einen Prozess zu beenden. (SIGKILL).
- Signale zum Debuggen von Programmen bzw. um die Aktivitäten eines Programme zu überwachen.

Signale werden in einem Prozess nur dann verarbeitet, wenn eine Kernelfunktion fertig ist oder das Programm innerhalb vom Kernel in einer Warteschleife sitzt. Sollte keine Kernelfunktion aufgerufen werden, wird das Programm nicht auf das Signal reagieren und wenn ein neues Signal kommt, dann überschreibt es das letzte Signal, da im Prozessmanager keine Liste ausstehender Signale vorhanden ist.

Klassische Mechanismen für die Kommunikation zwischen Prozessen

In Anwendungen

Für Anwendungen stehen Signale und Pipes für die Kommunikation zur Verfügung:

- Signale sind nicht wirklich hilfreich, da es nur eine kleine Anzahl möglicher Signale gibt. Daher werden sie nur für bestimmte Zwecke verwendet, wie zum Beispiel den Transfer des Fehlercodes eines Programmes zum aufrufenden Programm.
- Nur ein Signal kann pro Prozess bis zur Verarbeitung gespeichert werden.
- Es kann lange dauern bis ein Prozess eine Kernelfunktion aufruft und damit die Gelegenheit hat das Kernel auf die Signale reagieren zu lassen.
- Pipes können nur begrenzt benutzt werden, da sie nur in eine Richtung fließen und keine definierte Struktur haben.
- Prozesse werden bei der Verwendung von Pipes blockiert, wenn beim Lesen keine Daten vorhanden sind oder beim Schreiben der interne Puffer noch voll ist.

Klassisches Unix kennt keine Server Prozesse, die Anfragen von verschiedenen Prozessen direkt verarbeiten können.

Im Kernel

- Das Kernel benutzt Signale.
- Für gemeinsame genutzte Daten werden besondere Speicherbereiche benutzt, die in den Adressraum der jeweiligen Prozesse eingebunden werden. Da es kein preemptiven Prozesswechsel gibt und auch nur eine CPU müssen lediglich die Interrupts abgestellt werden um Zugriffe zu synchronisieren.
- Ein schlafender Prozess wird regelmäßig aufgeweckt, damit er überprüfen kann, ob seine Resource inzwischen frei wurde.

8. Sockets

Für die Kommunikation zwischen Prozessen, die durchaus auf mehrere Computer verteilt sein können, kann man seit der BSD 4.1 Unix Version (ca. 1984) auf die sogenannten Sockets zurückgreifen.

Sockets sind ähnlich den Pipes Datenströme zwischen zwei Prozessen. Allerdings können Daten in beide Richtungen übertragen werden. Je nach Unix Version können verschiedene Protokolle und Adressierungsarten benutzt werden. Die Beispiele hier beschränken sich auf das heute gebräuchliche TCP/IP Protokoll mit der IPv4 Adressierung.

Die Socket API

Die Socket Funktionen ergänzen die vorhandenen Funktionen zum Lesen und Schreiben. Dabei sind die Funktionen `read` und `write`, die auch bei Dateien benutzt werden synchron. Die asynchronen Funktionen melden über das Signal `SIGIO` den Erfolg ihrer Arbeit.

Die Funktion `socket` erzeugt ein neues Socket. Dieses Socket kann mit der Funktion `connect` zu einer angegebenen Adresse eine Verbindung aufbauen. Solange die Verbindung besteht können die Funktionen `read` und `write` Daten synchron übertragen. Alternativ kann die Funktion `bind` das Socket auch an eine Adresse binden. Nach einem `listen` Funktionsaufruf sammelt das System eingehende Anfragen. Mit einem Aufruf von `accept` nimmt man eine Anfrage an. Mit der `close` Funktion schliesst man ein Socket wie jede andere offene Datei.

Die Socket Funktionen:

```
int socket(int family, int type, int protocol);
int bind(int sockfd, struct sockaddr* address, int addresslength);
int connect(int sockfd, struct sockaddr* serveraddress, int addresslength);
int listen(int sockfd, int backlog);
int accept(int sockfd, struct sockaddr* clientaddress, int addresslength);
```

Die allgemeinen Datei Funktionen, die für Sockets mit verwendet werden:

```
int close(int fd);
int read(int fd, char *buf, int size);
int write(int fd, char *buf, int size);
```

Beispielprogramme

Client

Das Beispielprogramm ermittelt zuerst die IP Adresse der Webseite. Dann wird eine Verbindung aufgebaut und eine minimale Anfrage nach dem HTTP Protokoll abgeschickt. Die Webseite, die als Antwort kommt wird auf der Konsole ausgegeben.

```
#include <stdio.h>
#include <netinet/in.h>
#include <string.h>
#include <netdb.h>

#define servername "www.uni-koblenz-landau.de"
#define serverport 80

#define httprequest "GET / HTTP/1.1\nHost: www.uni-koblenz-landau.de\n\n"

int main (int argc, const char * argv[])
{
    int e;

    printf("First, lookup IP at DNS Server.\n");

    struct hostent *host=gethostbyname(servername);
    if (host==0)
```

```

    return 1;
printf("host: %s \n", host->h_name);

printf("Second, open a socket.\n");
int handle=socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
printf("socket handle %d \n",handle);

printf("Third, connect to host.\n");
struct sockaddr_in a;
bzero(&a, sizeof(a)); // clear structure
a.sin_family=AF_INET;
a.sin_addr.s_addr=((unsigned long*)(&host->h_addr[0])); // copy the address
a.sin_port=serverport;

printf("IP: %d.%d.%d.%d \n",
      (unsigned char)host->h_addr[0],
      (unsigned char)host->h_addr[1],
      (unsigned char)host->h_addr[2],
      (unsigned char)host->h_addr[3]);

e=connect(handle, (struct sockaddr*)&a, sizeof(a));
printf("result from connect: %d \n",e);

if (e>=0) // success
    {
    printf("Forth, do http get.\n");

    printf("httprequest: %s",httprequest);
    printf("strlen(httprequest): %d \n",(int)strlen(httprequest));
    e=write(handle, httprequest, strlen(httprequest));
    printf("result from write: %d \n",e);

    char buf[1000];
    int count=0;

    while (e != 0)
        {
            e=read(handle, buf, sizeof(buf));

            if (e>0)
                {
                count=count+e;
                buf[e]=0; // end mark
                printf("%s\n",buf);
                }
            }
        printf("read byte count: %d \n",count);
    }

e=close(handle);
printf("result from close: %d \n",e);

return 0;
}

```

Server

Dieses Programm wartet auf Port 8000 auf eine ankommende Anfrage. Wenn diese kommt, wird eine fest einprogrammierte HTML Seite als Antwort abgeschickt. Danach beendet sich das Programm.

```

#include <stdio.h>
#include <arpa/inet.h>

```

```

#include <netinet/in.h>
#include <string.h>
#include <sys/socket.h>

#define serverport 8000
#define httpanswer "HTTP/1.1 200 OK\nContent-Type: text/html\n\nHello World\n"

int main (int argc, const char * argv[])
{
    int e;

    printf("First, open a socket.\n");
    int handle=socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    printf("socket handle %d \n",handle);

    printf("Second, bind to port.\n");
    struct sockaddr_in a;
    bzero(&a, sizeof(a)); // clear structure
    a.sin_family=AF_INET;
    a.sin_addr.s_addr=INADDR_ANY;
    a.sin_port=serverport;

    e=bind(handle, (struct sockaddr*)&a, sizeof(a));
    printf("result from connect: %d \n",e);

    printf("Third, listen for an incoming connection.\n");
    e=listen(handle, 5); // 5 items in waiting quere
    printf("result from listen: %d \n",e);

    if (e>=0) // success
        {

            struct sockaddr_in cli_addr;
            int clilen=sizeof(cli_addr);
            int sock=accept(handle, (struct sockaddr*) &cli_addr, &clilen);

            printf("second socket handle %d \n",sock);

            printf("Read request.\n");

            char buf[5000];
            e=read(sock, buf, sizeof(buf));
            if (e>0)
                {
                    buf[e]=0; // end mark
                    printf("%s \n",buf);
                }
            printf("Send answer.\n");
            e=write(sock, httpanswer, strlen(httpanswer));
            printf("result from write: %d \n",e);

            close(sock);
        }

    e=close(handle);
    printf("result from close: %d \n",e);

    return 0;
}

```

Zusammenfassung

Klassisches Unix ist recht einfach gehalten, da die ursprünglichen Rechner nur eine CPU hatten und kein preemptives Multitasking konnten. Jedesmal wenn die Entscheidung zwischen Effizienz und Einfachheit stand, wurde der einfache Weg genommen.

Trotzdem ist Unix vor allem ein universelles und modular erweiterbares Betriebssystem.

Quellen

Operating Systems von Jean Bacon & Tim Harris (2003, Addison Wesley). Die Abbildungen sind zum Teil aus dem Buch entnommen.

"Unix Network Programming" von W. Richard Stevens (1990, Prentice-Hall Inc.), ISBN 0-13-949876-1.

Der Abschnitt zur Geschichte ist einem Text auf der Webseite http://www.cl.uni-heidelberg.de/kurs/ws01/zyklus/unix_geschichte.mhtml entnommen.

Christian Schmitz, schmitzi @ uni-koblenz.de