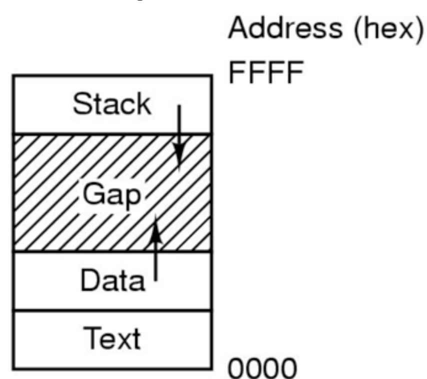
2 Prozesse

Definitionen

Ein Prozess ist ein Programm in Ausführung.
Sequentielle Ausführung von (Maschinen-)befehlen.
(Das ist eine von vielen möglichen Definitionen)

Jedem Prozess wird ein Adressraum zugeordnet, den dieser nicht verlassen kann.

Aufteilung des Adressraumes in Regionen:



Zusätzlich noch evtl. Shared Libraries

Jeder Prozess besteht aus:

- ProzessID
- Adressraum: Programm (Text), Stack, Heap (Bild)
- Aktueller Zustand des Prozesses: Befehlszeiger, Stackzeiger, Register, Allozierte Ressourcen
- Weitere Infos: BenutzerID, GruppenID, Priorität, Startzeit, , verbrauchte Rechenzeit, VaterID, Kindprozesse, Signale, geöffnete Dateien

Prozesstabelle speichert alle Informationen zu den Prozessen im Process Controll Block (PBC) außer dem Adressraum selber.

BS schaltet zwischen Prozessen um: Multiprogrammierung (Multitasking)
„Taskswitch“, „Contextswitch“

Geschwindigkeit der Abarbeitung nicht reproduzierbar (->Echtzeitbetreibssysteme)
Programm „merkt“ nichts vom Multitasking

Prozesserzeugung

- Initialisierung
- Systemaufruf
- Benutzeranfrage (“Doppelklick“, Kommandozeile)
- Stapelverarbeitung

„Hintergrundprozesse“ = Dienste, Daemons

Technisch identischer Vorgang: Prozess kann nur von existierendem Prozess erzeugt werden.
Unix: fork(), exec()

fork() erzeugt identische Kopie eines Prozesses.
Vater-Kind-Relation

Eine simple Kommandozeile:

```
#define TRUE 1

while (TRUE) {                              /* repeat forever */
     type_prompt( );                        /* display prompt on the screen */
     read_command(command, parameters);     /* read input from terminal */

     if (fork( ) != 0) {                    /* fork off child process */
         /* Parent code. */
         waitpid(-1, &status, 0);           /* wait for child to exit */
     } else {
         /* Child code. */
         execve(command, parameters, 0);    /* execute command */
     }
}
```

Grund für diesen Mechanismus: Vater und Kind teilen sich Dateideskriptoren und können somit Kommunizieren.
Vater und Kind unterscheiden sich nur durch ProzessID (im Gegensatz zur Natur hat ein Prozess nur einen Elternteil).
Prozess liefert Statuscode als Ergebnis, muss vom Vater abgeholt werden.

Nach dem fork() haben beide Prozesse eigenen Adressraum.
Aber: Text kann identisch sein (read-only).
Aber: Auch Stack/Heap können geteilt sein.

Copy-on-write: Erst wenn ein Block geschrieben wird, wird eine Kopie angelegt (Vorgriff: Virtueller Speicher)

Windows: Ein Systemaufruf CreateProcess()

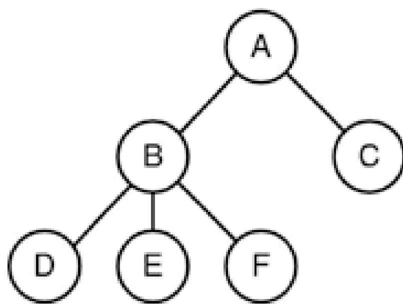
Prozessbeendigung

- Normales beenden (freiwillig)
- Bennden wg. Fehler (freiwillig)
- Schwerwiegender Fehler (unfreiwillig)
- Durch anderen Prozess (unfreiwillig)

Freiwilliges Beenden durch exit(), sonst durch das BS.

Prozesshierarchie

Nur bei Unix, wg. fork()



Prozess „init“ steht ganz oben und erzeugt beim Systemstart Prozesse, die auf Login-Versuch reagieren. Das Programm „login“ prüft Berechtigung. Danach Starten der Shell.

E/A-Kanäle eines Prozesses:

- Standardeingabe
- Standardausgabe
- Fehlerausgabe

Pipes: Die Standardausgabe eines Prozesses wird mit der Standardeingabe eines Kindprozesses verbunden.

Systemaufrufe

- Schnittstelle zw. BS und Anwendungsprogrammen, Übergabe der Kontrolle an das BS

- Für jedes BS anders
- Sicht der virtuellen Maschine
- Wie ein gewöhnlicher Prozeduraufruf, jedoch wird in den Kernelmodus gewechselt
- Zentrales Problem: Wechseln zugreifbaren Speicherbereiches (!)
- Beispiel: öffnen einer Datei: `count = read( fd, buffer, nbytes );`
- Detail am Rande: Pointer immer nur in den Userspace, nie in den Kernelspace
- Liefert Anzahl gelesener Bytes (-1 bei Fehler, Fehlernummer in globaler Variable)
- I. allg. mehrere hundert verschiedene Systemaufrufe im BS

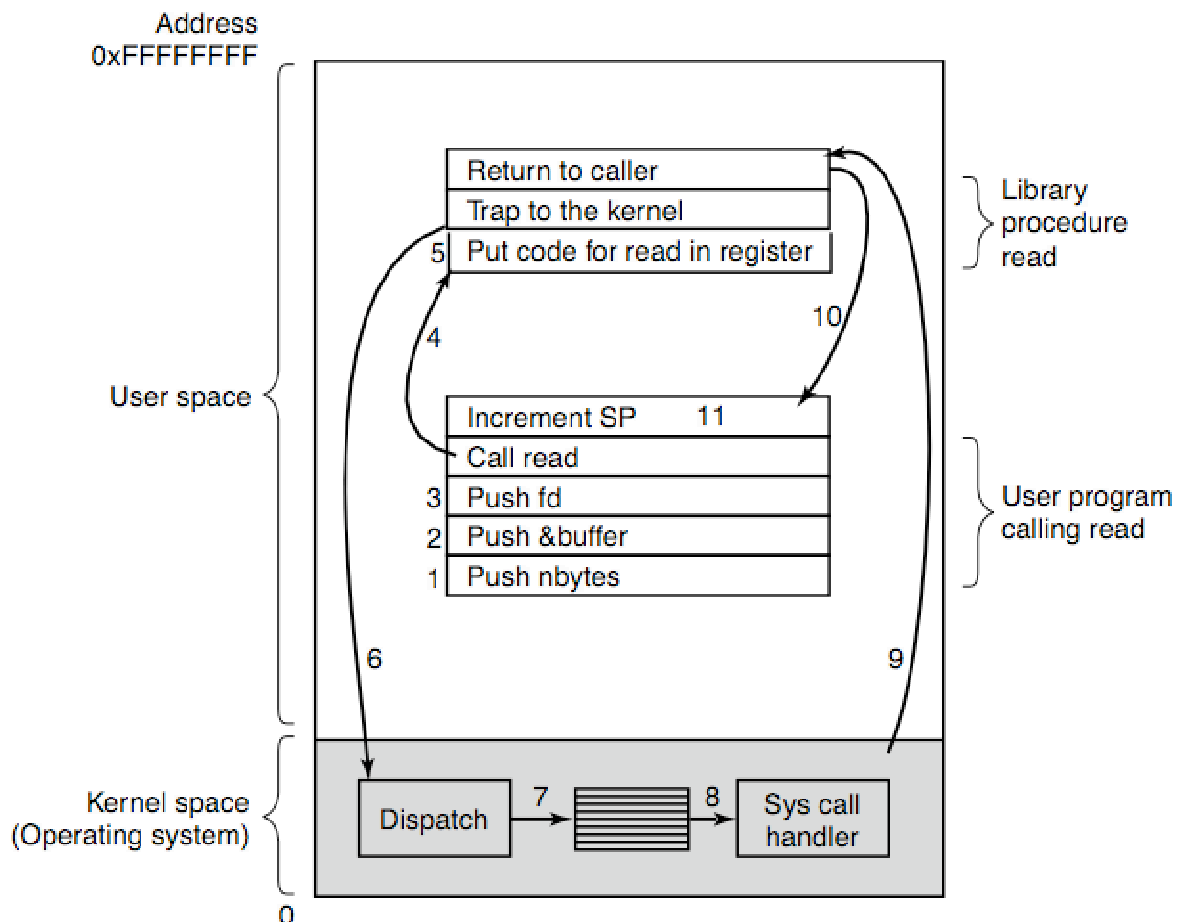


Abb.: Basisaufbau eines Systemaufrufes

- Wechsel mit TRAP. Funktion ist ein Parameter (i. Allg. ein Integer, welcher über einen Dispatch-mechanismus angesprungen wird).
- Wesentliche Designentscheidung eines BS: Welche Funktionen sind im Kernelmodus und welche im Usermodus?
- Beispiel: GUI: In Unix komplett in Usermodus, unter Windows im Kernelmodus, daher hat Windows wesentlich mehr Systemaufrufe

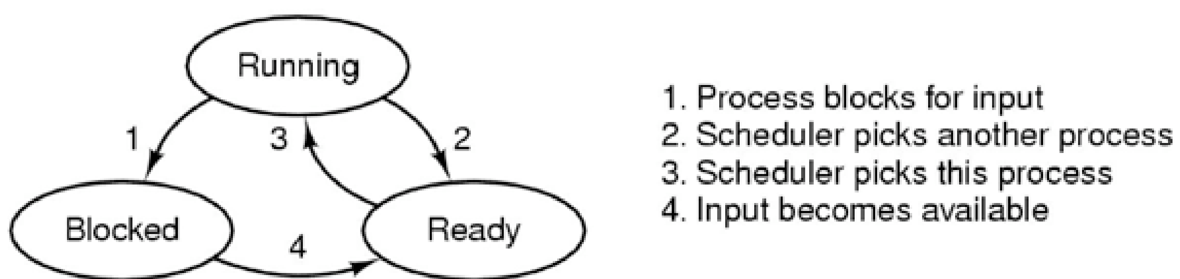
Prozesszustände

Beispiel: ls | grep 'D'
Listet ein Verzeichnis auf und filtert alle Zeilen, welche den Buchstaben „D“ enthalten.

Realisierung:
- Shell forkt den Prozess ls
- Shell forkt den Prozess grep
- Shell Verbindet die Ausgabe von ls mit der Eingabe von grep
- Pipe ist ein Puffer beschränkter Größe

Problem: Einer muss ggfls. Auf den anderen warten.

Prozess muss blockieren, weil:
- er auf Eingabedaten wartet
- er vom Betriebssystem die Ressource „CPU“ entzogen bekommt



1, 4: Prozess kommt automatisch von Running nach Blocked, da er einen Systemaufruf durchführt. Kein explizites Abgeben der Ressource CPU notwendig.
2, 3: BS-Scheduler (->Scheduling)

Frage: Wie kommt man rein und raus?
Frage: Wieviele Prozesse können in welchen Zuständen sein?

Prozessmodell unabhängig von Interrupts und low-level-Kommunikation.

Signale:
Ein Prozess kann einem anderen Prozess oder sich selbst (gleicher Benutzer) ein Signal schicken (Systemaufruf: kill()):
- Beende dich baldmöglichst (ctrl-c)
- Standardeingabe/ausgabe wurde beendet

- Anhalten/Fortfahren
- Timer abgelaufen
- Fataler Fehler (Division durch 0, Speicherzugriffsfehler)
- Sofort beenden

Signale sind Flags, die beim Wechsel des Prozesszustandes abgefragt werden. Prozess kann diese selber abfangen (ausser letzterem). Wenn nicht abgefangen, terminiert Prozess sofort. Versenden eines Signals mit „kill“

Frage: wann bleibt ein Signal unbeachtet?

Zombies:
Prozesse, die Terminiert sind, deren Vaterprozess den Status noch nicht abgeholt hat. Belegen Platz in Prozesstabellen („<defunct>“), aber keinen Speicherplatz mehr.

Anmerkung: Prozesse, welche im Kernelmodus arbeiten, sind i.allg. nicht unterbrechbar (!)

Threads

Prozess = Einheit, welche Ressourcenverwendungen zusammenfasst.
Thread=(Ausführungs-)Faden: Bestehend aus:

- Befehlszähler
- CPU-Register
- Stack

Threads sind Erweiterung des Prozessmodells, bei dem mehrere Ausführungsfäden in einem Prozess existieren können.

Unterschied zu Prozessen?
„leichtgewichtige Prozesse“, „Multithreading“.

Zustand eines Threads wie bei Prozessen.

Jeder Thread hat eigene Ablaufhistorie, aber (globale) Variablen/Objekte werden geteilt.

Thread-Bibliothek:

- Erzeugen eines thread
- Beenden eine thread
- Warten auf einen thread
- yield (freiwillige Abgabe der Rechenzeit)

Vorteile:

- Erzeugen/Zerstören/Wechsel zwischen Threads schneller als zwischen Prozessen
- Verteilung von Threads auf mehrere CPUs
- Blockierende Prozesse innerhalb eines Prozesses
- Trennung der Aufgaben, Programmiermodell wird einfacher (Beispiel: Scrollbalken)

Nachteile/Probleme:

- Kein Speicherschutz zwischen Threads
- Was passiert bei fork()?

Beispiele für Threads: Dokumentformatierung, Autosave, Tooltips, Browser: Dokument laden während formatiert/gescrollt wird.

Zwei mögliche Implementierungen: Im Prozess selber oder im Kernel.

Unix-Programme nutzen eher Prozesse, Windows-Programme eher auf Threads.

```
public class Listing2209
extends Thread
{
  static int konto1 = 1000;
  static int konto2 = 1000;

  public static void main(String[] args)
  {
    Thread t1 = new Listing2209();
    Thread t2 = new Listing2209();
    t1.start();
    t2.start();
  }

  public void run()
  {
    while (true) {
      for( int i = 0; i<10000; i++ ) {
        konto1 = konto1 + 500;
        konto2 = konto2 - 500;

        konto1 = konto1 - 500;
        konto2 = konto2 + 500;
      }

      System.out.println(konto1+"/"+konto2);
    }
  }
}
```

Konvertieren von Single-Thread-Programmen in Multithread-Programme:

- Problem der globalen Variablen
- Prozeduren oft nicht ablaufinvariant (reentrant)
- Signale

Fragen: Welche Aufgaben mit Threads, Welche mit Prozessen, Welche mit Interrupts?