Betriebssysteme

Kap. 2: Prozesse und Threads
2.1 Prozesse

Stand: WS 08/09 (19.11.08)

Prof. Dr. Wolfgang Küchlin

Dipl.-Inform., Dr. sc. techn. (ETH)

Arbeitsbereich Symbolisches Rechnen
Wilhelm-Schickard-Institut für Informatik
Fakultät für Informations- und Kognitionswissenschaften

Universität Tübingen

Steinbeis Transferzentrum
Objekt- und Internet-Technologien (OIT)

Wolfgang.Kuechlin@uni-tuebingen.de
http://www-sr.informatik.uni-tuebingen.de

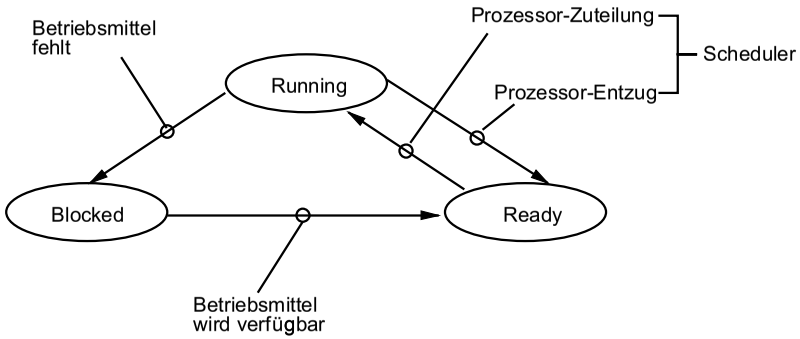
Kap 1.2.1 Prozesse

- **Inhalt**
  - Konzept des Prozesses
  - Prozesszustände
  - Speichermodell
  - Virtueller Speicher – Realer Speicher
  - Behandlung von Seitenfehlern
  - Page Daemon
  - Erzeugen von Prozessen
  - Löschen von Prozessen

Konzept des Prozesses

- Verwaltungseinheit zur Ausführung von Programmen
  - Programmcode + Betriebsmittel
  - Repräsentiert durch Prozess-Leitblock (process control block PCB)
- Betriebsmittel
  - Prozessor(en)
  - Registersatz mit Programmzähler
  - Speichersegmente (Stack + Heap) mit memory map
  - Files, Netzwerkverbindungen, Puffer
  - Spezielle Geräte
- Rechte
  - User-Prozess, Kernel-/BS-Prozess
  - Prioritäten

Grundlegende Prozess-Zustände

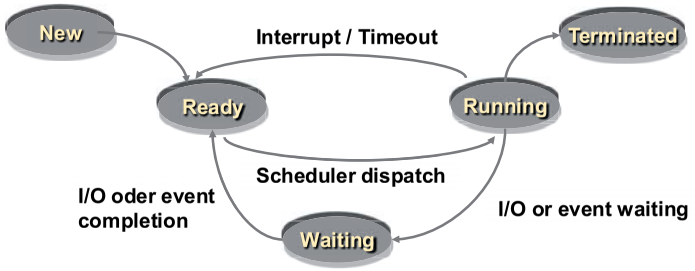


Prozesszustände

- Betriebssystem verwaltet Prozess-Zustände
  - Neu
  - Laufend
  - Blockiert / wartend (ausgelagert oder resident)
  - Bereit / lauffähig (ausgelagert oder resident)
  - Beendet
- Laufend
  - Prozess hat alle nötigen Betriebsmittel inkl. Prozessor
  - (Pro Prozessor-Core) nur *ein* laufender Prozess.
- Blockiert / wartend
  - Prozess wartet (ohne Prozessor) auf Betriebsmittel
  - **Lauffähig**: nur Betriebsmittel Prozessor fehlt


Zustands-Übergangsdiagramm

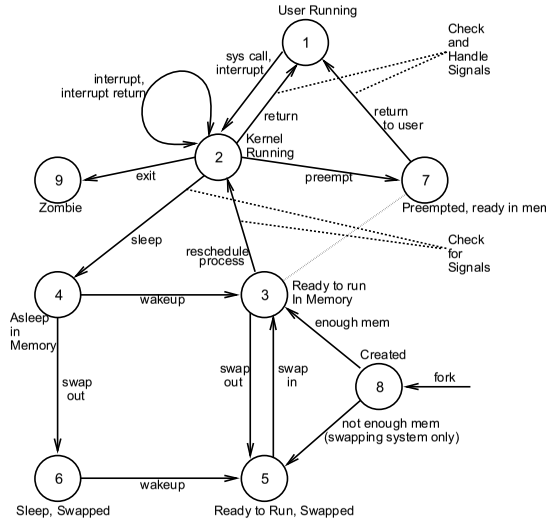
- Bei Ausführung ändert sich ggf. Prozess-Zustand
- Betriebssystem kontrolliert die Übergänge




2.2 Prozesse

- Implementierung von Prozessen in UNIX
- Realisierung von `fork()`
- Scheduling
- Leichte Prozesse (Threads of Control)
- Prozess-Synchronisation


Prozess-Zustände in UNIX



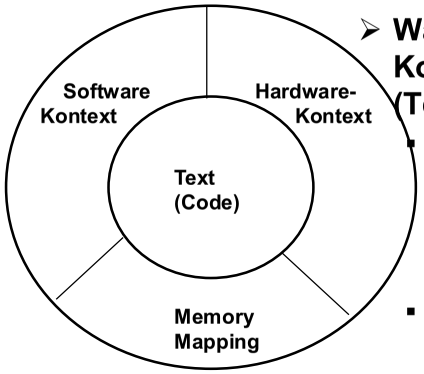

Prozess-Leitblock (*Process Control Block - PCB*)

- Alles, was es zu wissen gibt, z.B.
  - Prozesszustand
  - Prozessnummer
  - Programmzähler
  - Registerinhalte
  - Scheduling Informationen
  - (Verweise auf) Speichersegmente
  - Liste der offenen Files
  - Liste der Netzwerkverbindungen
  - Signale u. Signalmasken


Prozess-Implementation in UNIX

- Prozesstabellen-Eintrag *(process table entry)*
  - Scheduling Information
  - Speicherinformation
  - Anstehende Signale
- Benutzerstruktur *(u.-structure)*
  - Registerinhalte
  - Status des Systemaufrufs
  - Kernel Stack
  - (File) Descriptor Table
  - Abrechnung (*Accounting*)
  - Signal mask / handlers

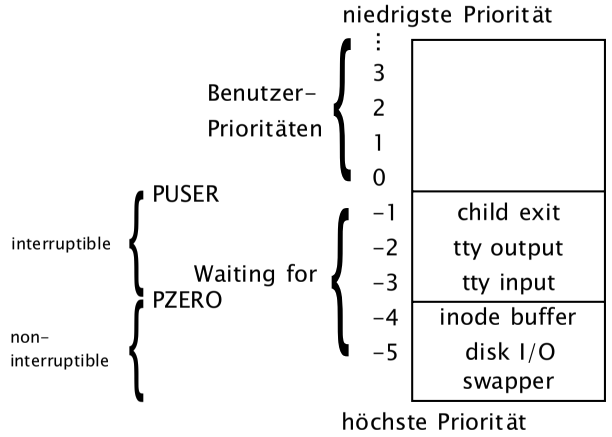

Prozesskontext (UNIX)



- **Während der Ausführung, Kontext um Programmcode (Text)**
  - **Software-Kontext: Prozess-spezifische Daten des Kernels**
    - PCB / Eintrag in Prozesstabelle
    - Benutzerstruktur
    - Kernelstack
  - **Hardware-Kontext:**
    - Register
  - **Memory-Mapping:**
    - Abbildung anhand Page-Tabelle
    - Virt. Adressraum
      - phys. Speicheradresse

Wechsel des laufenden Prozesses auf CPU: Context Switch

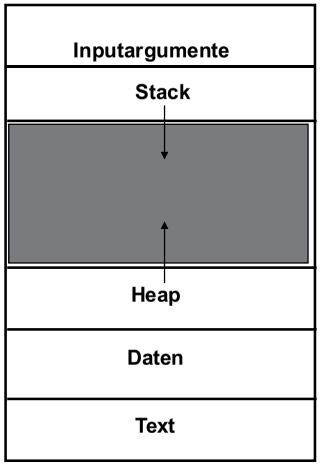

Prozess-Prioritäten in UNIX




2.4 Das UNIX-Speichermodell

- Speicherverwaltung
- Das Speicherbild eines Prozesses
- Behandlung von Seitenfehlern
- Virtueller Speicher
- Swapping (Prozesstausch-Verfahren)
- Paging (Seitentausch-Verfahren)
- Der UNIX Seitentausch Algorithmus


Virtueller Adressraum (klassisches UNIX)



Argv, envp

Lokale Variablen

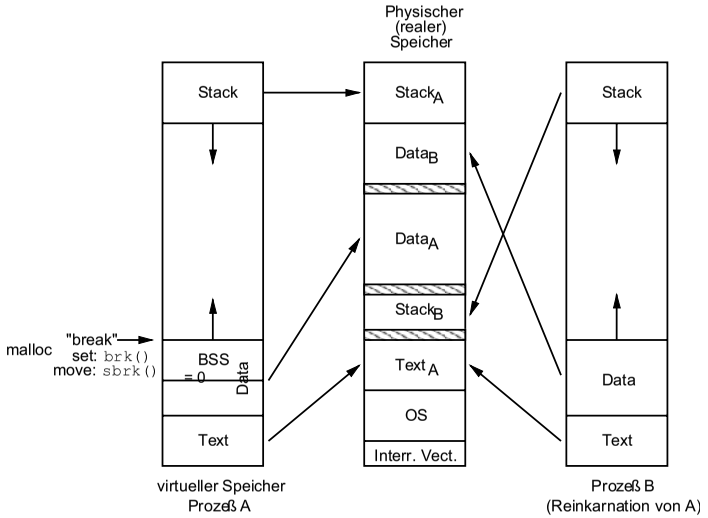
C-Programm Aufruf:
int main (int arc, char *argv[ ], char *envp[ ]);

Malloc, new

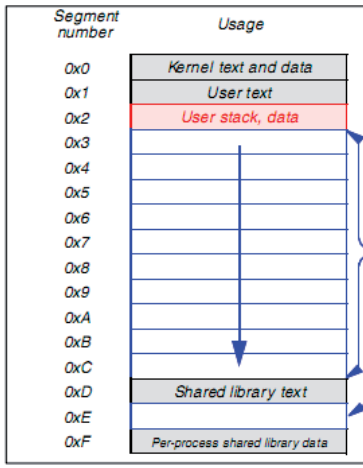
Globale Variablen

Programmcode


Speichermodell von UNIX Prozessen




Beispiel: Speichermodell von AIX



- 16 Segments @ 256MB each = 4GB address space
- 0x0 segment: user process access prohibited.
- 0x1 segment: user process text. If another process shares the same executable file, there will be only one instance of the text pages for that executable file on the system.
- 0x2 segment: user data, heap, and stack. □

- 0xD segment: shared library text. This segment is shared by all 32-bit user processes. The virtual addresses of loaded shared text objects can be examined using the genkld command.
- 0xF segment: per-process shared library data.

Figure 3-2 Default memory model (segment usage)[4]


(Standard) Speichermodell von AIX (Detail)

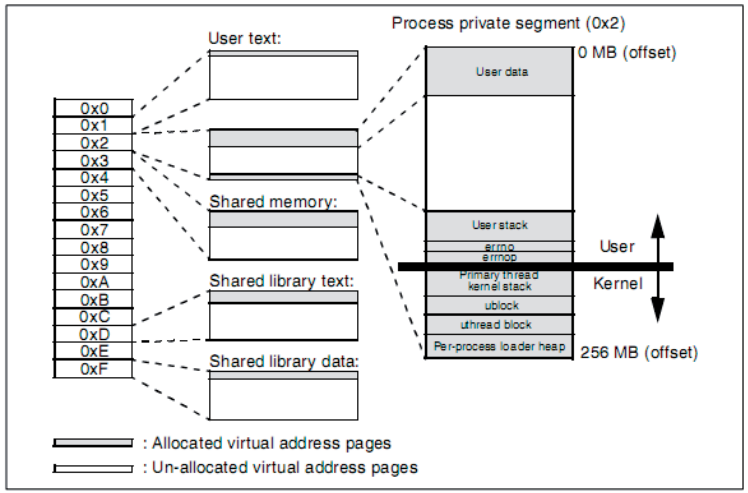
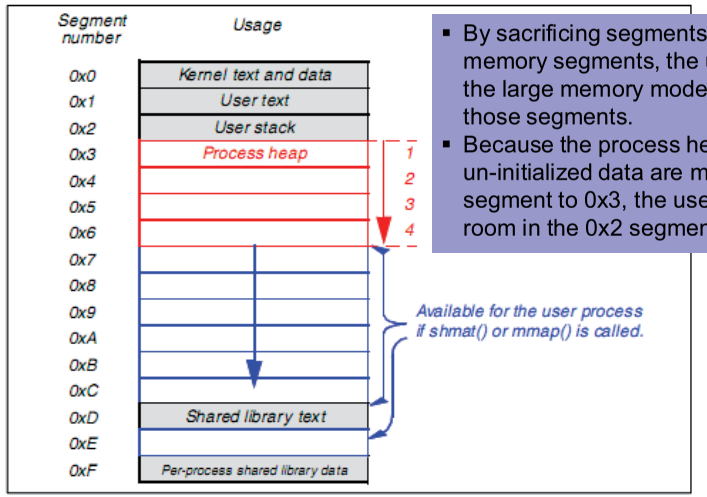


Figure 3-3 Default memory model (detail)

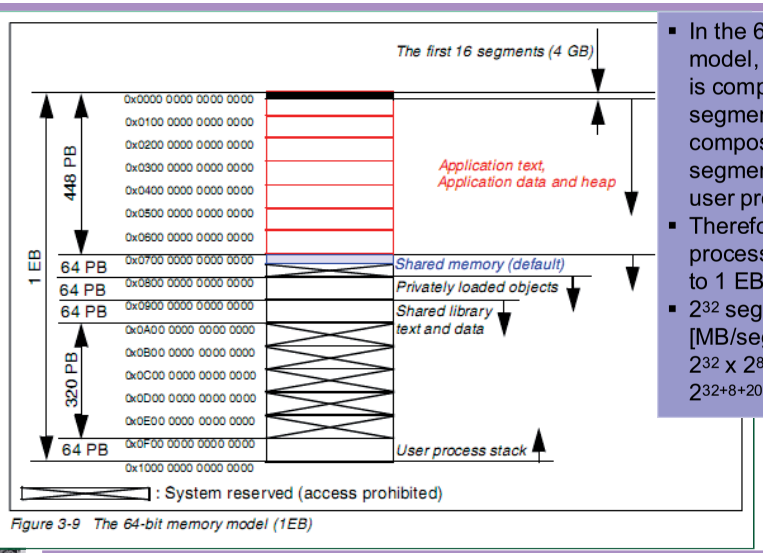
AIX Large Memory Model



- By sacrificing segments available for shared memory segments, the user process running in the large memory model can place its heap on those segments.
- Because the process heap and initialized and un-initialized data are moved from the 0x2 segment to 0x3, the user stack can enjoy more room in the 0x2 segment in this model.

Figure 3-4 Large memory model (segment usage)

AIX 64-bit Speichermodell



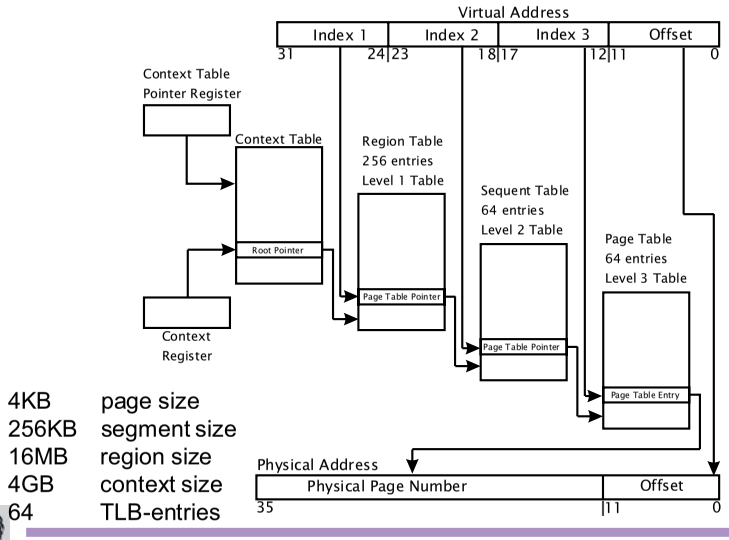
- In the 64-bit user process model, an address space is composed of 2^{32} segments, while it is composed of $2^4 = 16$ segments in the 32-bit user process model.
- Therefore, the 64-bit user process can address up to 1 EB (exabytes)
- 2^{32} segments x 256 [MB/segment] = $2^{32} \times 2^8 \times 2^{20}$ bytes = $2^{32+8+20} = 2^{60} = 1$ EB

Figure 3-9 The 64-bit memory model (1EB)

MMU für große Adressräume

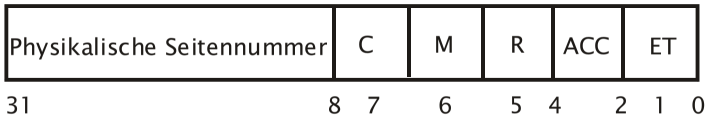
- Große Adressräume (insbes. 4-bit) sind großteils leer
- Einstufige Seitentabellen sind ungeeignet
  - 4 GB mit 4 KB Seiten → 1.000.000 Seiten
  - Seitenadresse hat mindestens (32-12)=20 bit
  - Eintrag in Seitentabelle hat 3 – 4 Byte Größe
  - Seitentabelle hat 4MB – hauptsächlich „invalid“ Einträge
  - Seitentabellen für 1.000 Prozesse brauchen 4GB Platz
- Mehrstufige Seitentabellen
  - Kennzeichnung großer Segmente oder Regionen (= large pages 1MB) als „leer“ – ohne Seitentabellen.

Architektur der SuperSPARC on-chip MMU



4KB	page size
256KB	segment size
16MB	region size
4GB	context size
64	TLB-entries

Seitentabellen-Eintrag der SuperSPARC MMU



- C: Cacheable. Page may be cached.
  c==0 if page maps i/o devices (contains volatile data)
- M: modified (dirty). MMU sets M=1 after write to page.
- R: referenced. MMU sets R=1 when referencing page.
- ACC: access permissions.
- ET: entry type and validity. ET==0 if page is not resident. If ET==2 then bits 8—31 are physical page #

Seitentabellen-Eintrag der SuperSPARC MMU

- Access permissions depend on execution mode

ACC	User Access
0	Read only
1	Read / write
2	Read / exec
3	R / w / x
4	Exec only
5	Read only
6	No access
7	No access

ACC	Supervisor access
0	Read only
1	Read / write
2	Read / exec
3	Read / write / execute
4	Exec only
5	Read / write
6	Read / execute
7	Read / write / execute

Schritte der Adress-Umsetzung (VA → PhA)

1. MMU bekommt virtuelle Adresse
2. MMU durchsucht TLB. if Erfolg, return PhA.
3. Else Table-Walk. if Erfolg, return PhA.
4. Else ungültig markierter Eintrag in Tabelle (V = 0): Seitenfehler (page fault).
5. HW sichert Info zum Wiederaufsetzen der Instruktion. Kernel Trap bzw. Interrupt.
6. Aufruf der Seitenfehlerbehandlungsroutine (page fault handler) PFH im BS.

Behandlung von Seitenfehlern

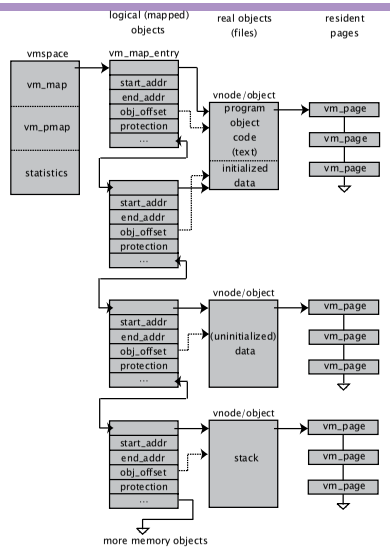
7. PFH ermittelt die verursachende virtuelle Adresse und Seite (aus MMU oder indirekt aus Instruktion).
8. PFH stellt fest, ob in den benutzten Bereich des virtuellen Adressraums adressiert wurde (=Pseudofehler) oder in den unbenutzten Bereich (=echter Fehler).
   - Hierzu wird eine Liste aller virtuellen Speicherbereiche (= -Objekte) gebraucht (memory map).
   - Jedes Objekt ist verzeichnet mit virt. Anfangs-Adr. und End-Adr.
   - Durchsuchen der Liste, wozu die umzusetzende Adresse gehört.
9. Bei echtem Fehler (Adresse gehört zu keinem Objekt) SIGSEGV an Prozess und neues Scheduling; exit.
10. Bei Pseudofehler, Einlagern des zur virtuellen Seite gehörenden Blocks im „backing file“ des Objekts auf neue Kachel, und Eintragen der Kacheladresse in die Seitentabelle.


Behandlung von Seitenfehlern

Zum Bestimmen des Blocks im backing file (swap file oder File, das mit mmap in den Adressraum eingeblendet wurde) werden 2 Offsets benötigt:

1. Offset der VA bezüglich Anfang des Memory Objekts
   - einfache Adressrechnung
2. Offset des Memory Objekts bezüglich des Anfangs des Backing File (falls nur Teil dieses Files eingeblendet wurde)
   - Dieser offset ist im mmap-Eintrag verzeichnet


Memory Map in 4.4 BSD UNIX

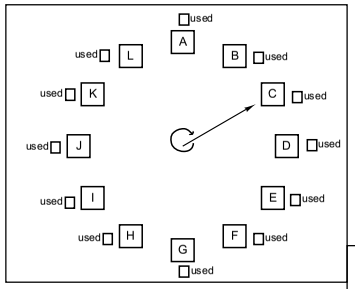



Page Fault Dispatch in 4.4 BSD

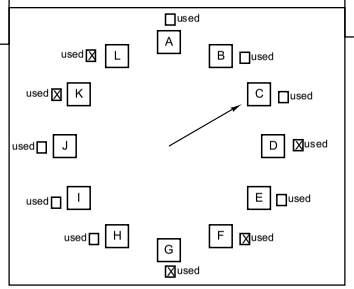
Virtual Address VA causes page fault.

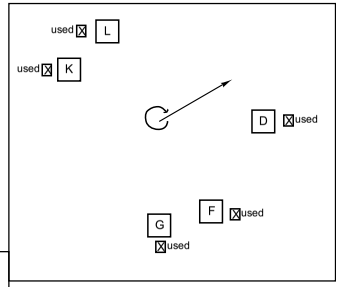
1. Find the vmspace structure and the vm_map_entry list
2. Check for each entry e whether start_addr <= VA <= end_addr. If no such e exists, send SIGSEGV to process.
3. Otherwise convert VA to offset within mapped object as follows: object_offset = VA – e->start_addr + e->object_offset
4. Present object_offset to mapped object, which allocates a vm_page structure and uses its pager to fill page. The object returns a pointer to the vm_page.
5. The OS maps this page into the process address space by setting the page table entry.


Seitentausch-Algorithmus



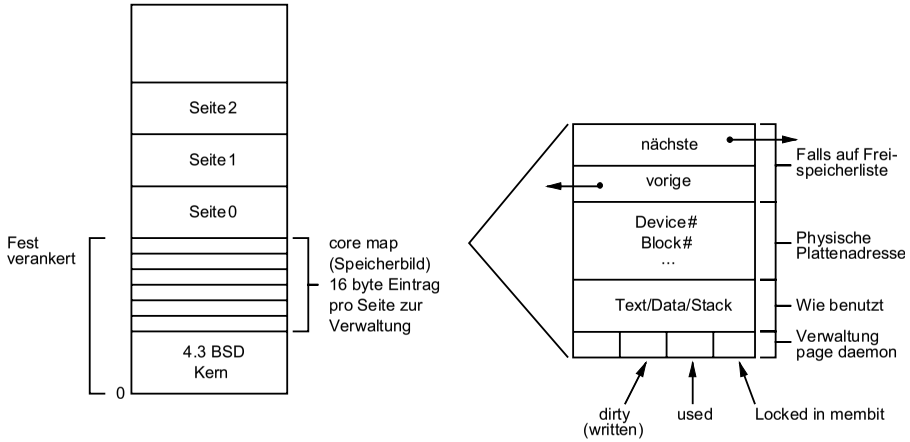
1. Durchlauf:
Alle „benutzt“ Bits werden zurückgesetzt



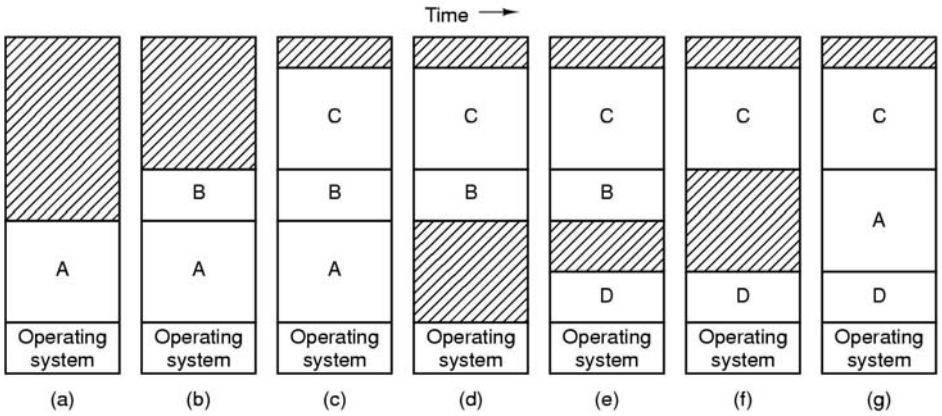


2. Durchlauf:
Alle unbenutzten Seiten werden freigesetzt.
(Schmutzige Seiten werden auf Platte zurückgeschrieben)

Seiteneinteilung im Hauptspeicher



Swapping



Fork und exec (UNIX)

Erzeugen eines neuen (geklonten) Prozesses

```
pid_t fork(void);
```

Eltern- und Kindprozess kehren nach id=fork() zurück;
Eltern mit Prozess-ID des Kindes, Kind mit 0.

Typische Benutzung:

```
if (id=fork()) Eltern_Code();
        else Kind_Code();
```

Start eines Programmes, aktuelles Programm wird ersetzt

```
int execvp (char *file, char *argv[]);
int execve (char *path, char *argv[], char *env[]);
```

Vererbung einiger Attribute

Attribut	Fork (2)	Exec (2)
Prozeß-ID	-	+
Elternprozeß-ID	-	+
Reale User-ID	+	+
Effektive User-ID	+	evtl.
Kontroll-Terminal	+	+
Arbeitsverzeichnis	+	+
Offene File-Descriptors	+	evtl.
Signalmaske	+	+
Signalhandler	+	-
Umgebungsvariablen	+	evtl.


Beispiele für fork und exec

- Nebenläufige Ausführung eines Programmes (concurrent, asynchron)
  - ```
    if ( !fork() )  // jetzt sind wir im Kind-Prozess
      execvp(pgm, argv);
    ```
- Synchrone Ausführung eines Programmes
  - ```
    ...
      childpid = fork();
      if ( !childpid )     //  this is the child !
    execvp (pgm, argv);
            else // this is the parent
    waitpid (childpid, &status, 0);
    ```
- Optimierung (BSD UNIX, Solaris)
  - Statt *fork* besser *vfork* (kein Kopieren)


Termination eines Prozesses

- **void _exit(int status)** gibt **status** an Eltern
- **void exit(int status)** führt Exithandler (**atexit(3)**) aus, schließt I/O Streams und ruft **_exit(2)** auf.
- Terminationsmöglichkeiten
  - return in main (**=exit(3)**)
  - Aufruf von **exit(3)** oder **_exit(2)**
  - **abort(3)** erzeugt Signal **SIGABRT**
  - Empfangen eines Signales
- Warten auf einen Prozess
  - **pid_t wait (int *statusp);** Bei Aufruf von wait(2) blockiert der Prozess bis zur Termination eines Kindes.
  - **pid_t waitpid(pid_t pid, int *statusp, int opts);** wartet auf spezifisches Kind


Daemons (UNIX)

- BS-Prozesse für spezielle Dienste (z.B. Administration, Netzwerk, Druckersteuerung), laufen im User-Mode unter spezieller User-ID (z.B. root)
- Start entweder beim booten (durch init) oder aufgrund eines System Calls durch den Kernel
- Häufige Funktionsweise:
  - **Bei jeder Anfrage wird ein Prozess zur Bearbeitung geforkt**.


Fehlerbehandlung (UNIX)

- Fehler bei System Call
  - Fehlerart in globaler Variable **errno**
- **void perror (char *msg);**
  - gibt msg und aktuellen Fehler mit Information aus (vgl. errno.h)
- **char *sterror(int errcode)**
  - liefert Pointer auf Fehlermeldung zum Fehlercode errno

Regel:
Jeden System Call abfangen und entsprechende Fehlermeldung ausgeben

Beispiel: Betriebssystem Mach

- Microkernel Architektur
- Kernel und Prozesse kommunizieren über Ports (mailboxen)
- Wenige traditionelle System Calls
  - Port und Message handling
  - port _allocate, msg_send, msg_receive, msg_rpc, port_status
  - Andere Systemfunktionen damit implementiert:
    statt syscall() message: msg_send(system_port, syscall)
- IPC über FIFO-Mailboxen (Mach: Port)
- Jedem Prozess (Task) sind Mailboxen für Kommunikation mit BS zugeordnet

Beispiel: Betriebssystem Mach

Problem:
Effizienz (Kopieren der Nachrichten)

Lösung:
BS bildet die entsprechenden virtuellen Adressen auf die gleichen physikalischen ab (Shared-Memory)

Signals

- Signals are used in UNIX systems to notify a process that a particular event has occurred
- A **signal handler** is used to process signals
  1. Signal is generated by one process (or the operating system)
  2. Signal is delivered by the operating system to another process (by marking the process table entry)
  3. Dispatcher finally sees the signal mark, looks up the corresponding signal handler, sets up the signal stack and the start address of the handler instead of the saved stack and address. If there is no handler for the signal, abort the process.
  4. Signal is handled