Betriebssysteme

Kap. 2: Prozesse und Threads
2.1 Prozesse

Stand: WS 12/13 (27.11.12)

Prof. Dr. Wolfgang Küchlin

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

Arbeitsbereich Symbolisches Rechnen
Wilhelm-Schickard-Institut für Informatik
Mathematisch-Naturwissenschaftliche Fakultät

Universität Tübingen

Steinbeis Transferzentrum
Objekt- und Internet-Technologien (OIT)

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

SR

Kap 1.2.1 Prozesse

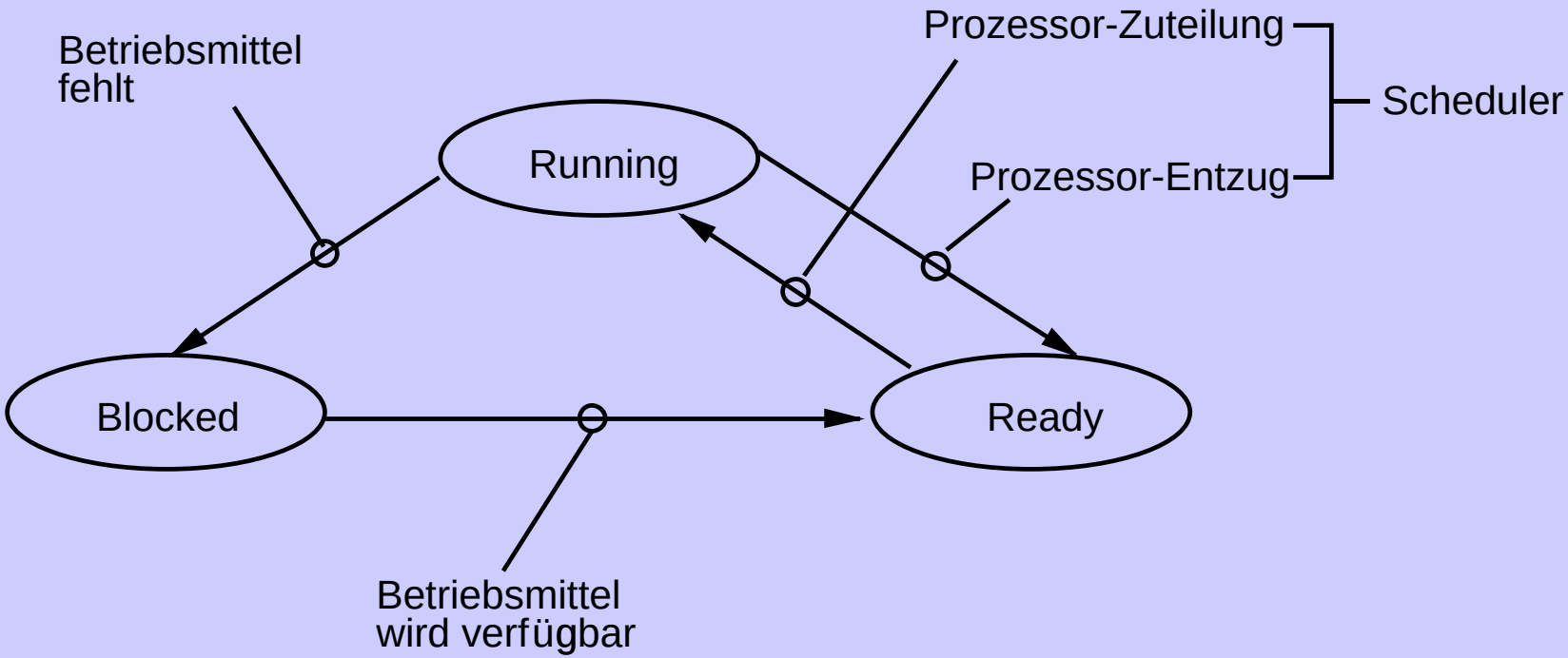
Inhalt

- 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

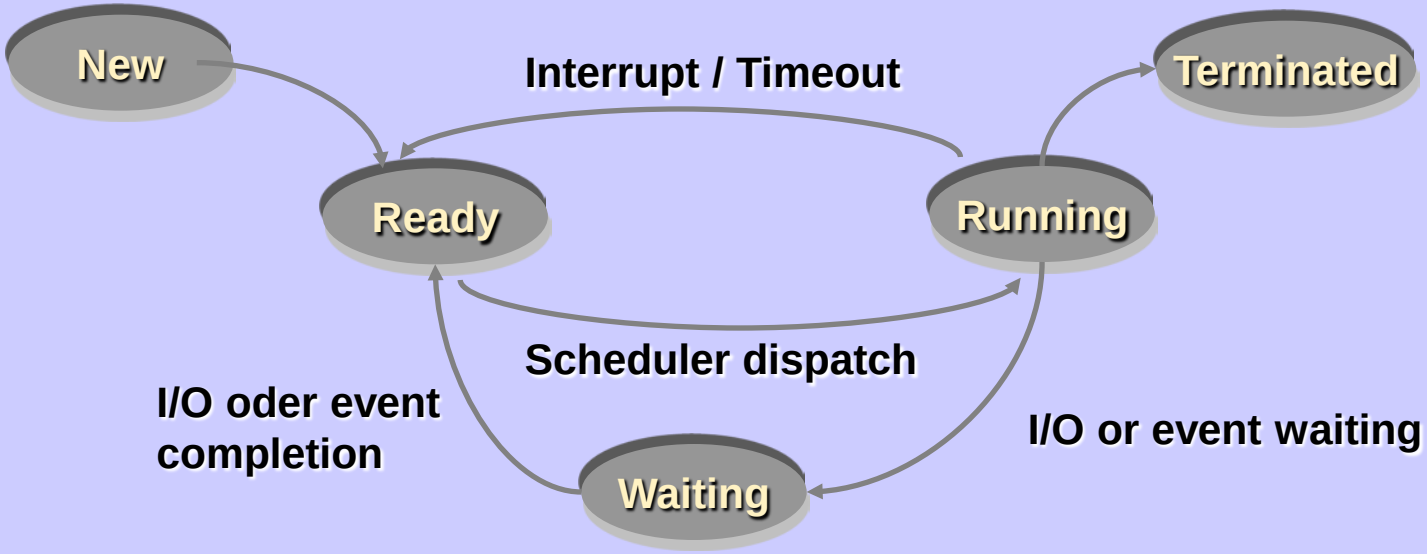


Prozesszustände

- 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

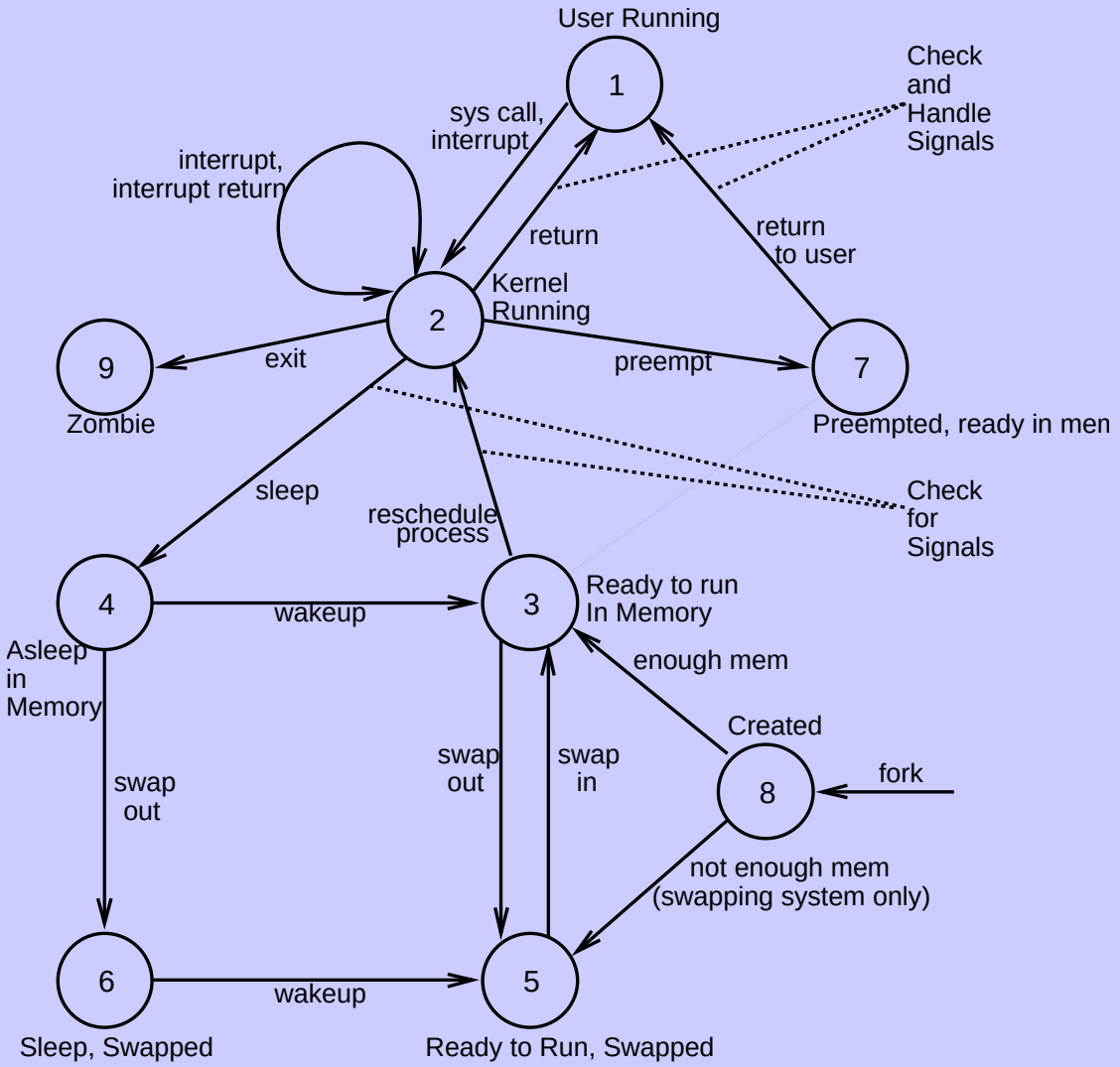
- Bei Ausführung ändert sich ggf. Prozess-Zustand
- Betriebssystem kontrolliert die Übergänge



2.2 Prozesse

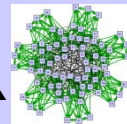
- Implementierung von Prozessen in UNIX
- Realisierung von `fork()`
- Scheduling
- Leichte Prozesse (Threads of Control)
- Prozess-Synchronisation

Prozess-Zustände in UNIX



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

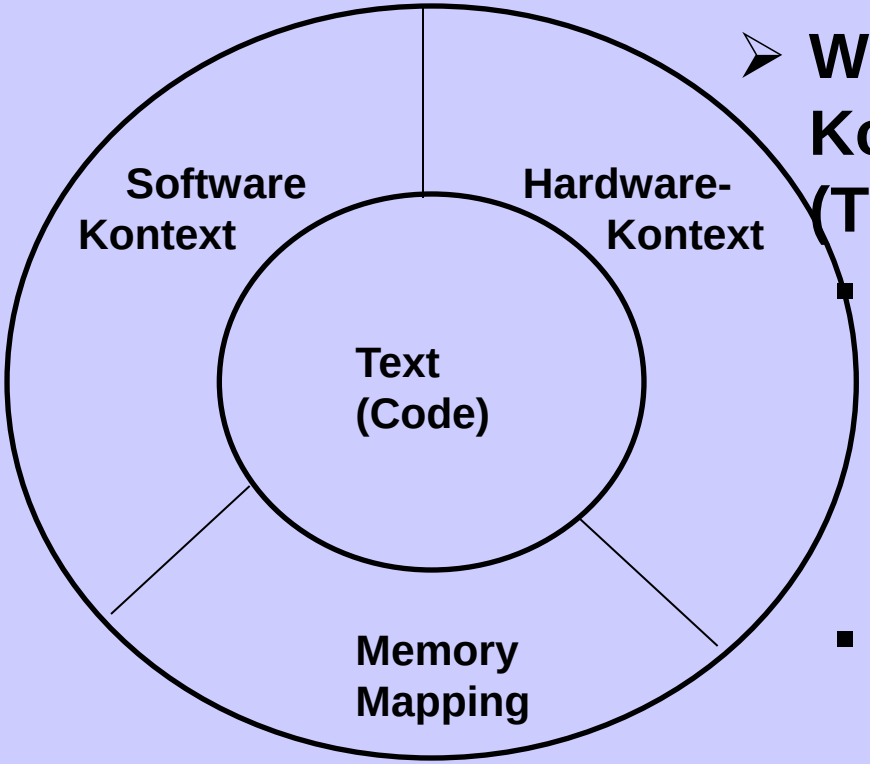
- Alles, was es zu wissen gibt, z.B.
  - Prozesszustand
  - Prozessnummer
  - Programmzähler
  - Registerinhalte
  - Scheduling Informationen
  - (Verweise auf) Speichersegmente
  - Liste der offenen Files
  - Liste der Netzwerkverbindungen
  - Signale u. Signalmasken



PCB-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)

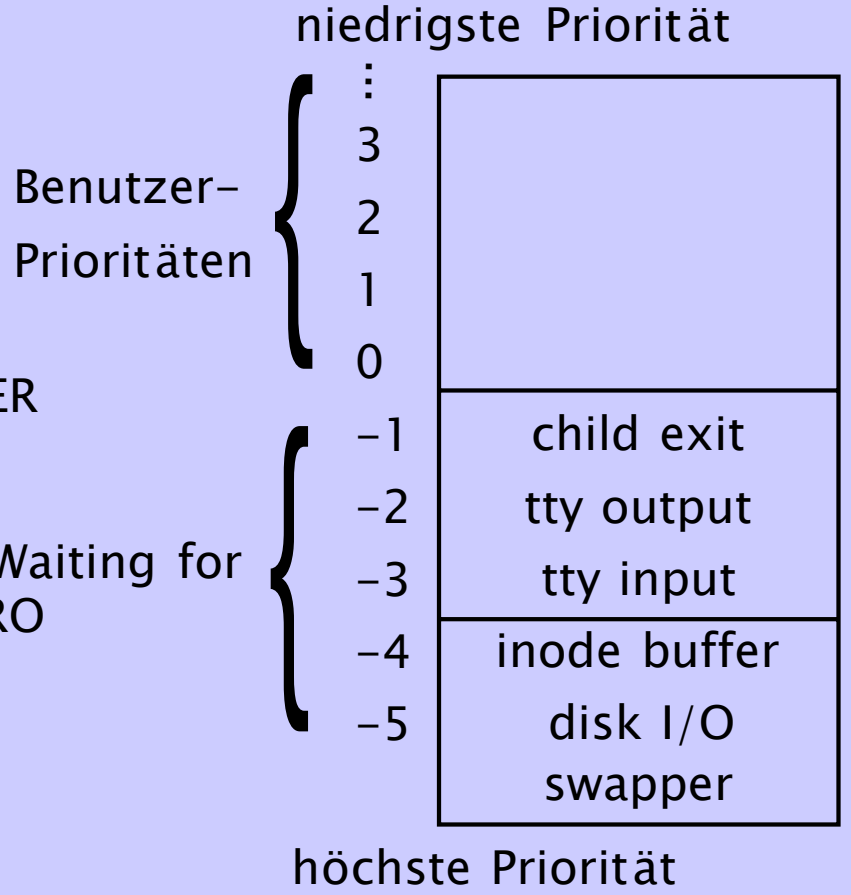


Wechsel des laufenden Prozesses auf CPU: Context Switch

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

Klassische Prozess-Prioritäten in UNIX

- Ein Prozess hat 2 Prioritäten
  - user priority (while in user mode)
  - kernel priority (while sleeping and after awakening while in kernel mode)

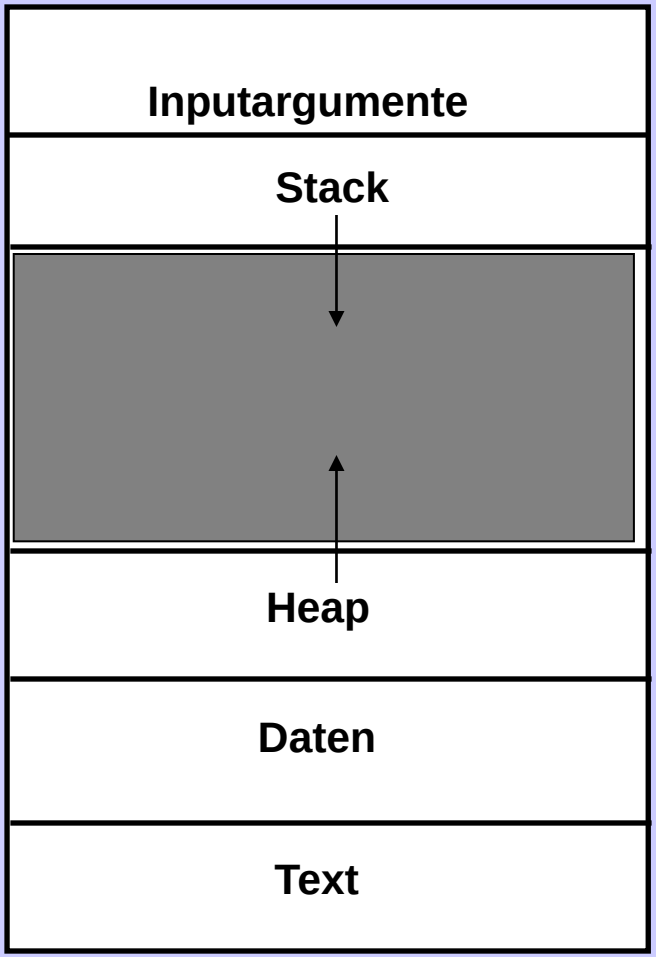


- Processes asleep between PZERO and PUSER are awakened by a signal (in 4.4 BSD the PCATCH flag must also be set)
- processes asleep below PZERO are never awakened by a signal

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

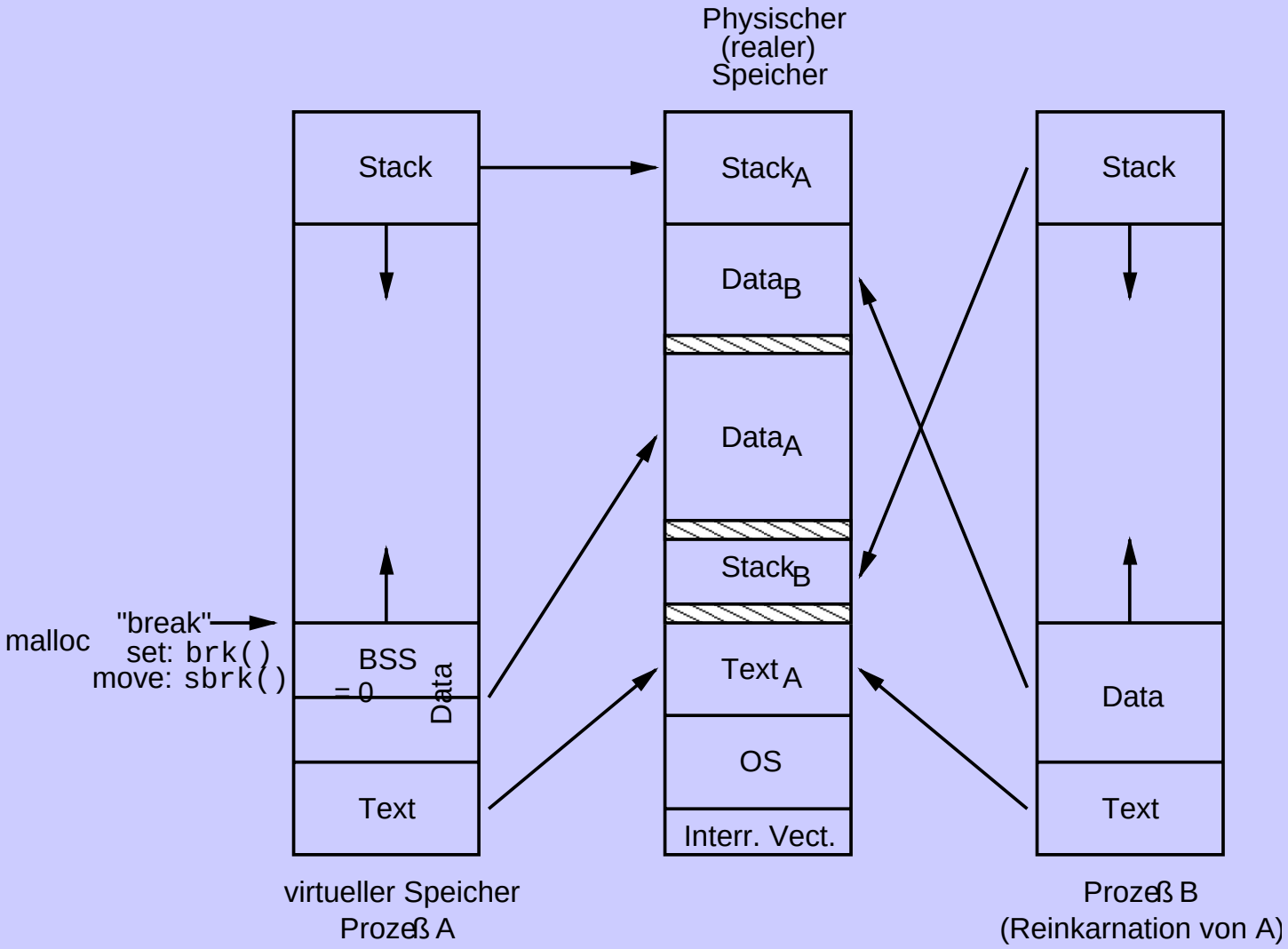
C-Programm Aufruf:
int main (int arc, char *argv[ ], char *envp[ ]);

Malloc, new

Globale Variablen

Programmcode

Segment-basierte Speicher-Abbildung



Beispiel: Speichermodell von AIX

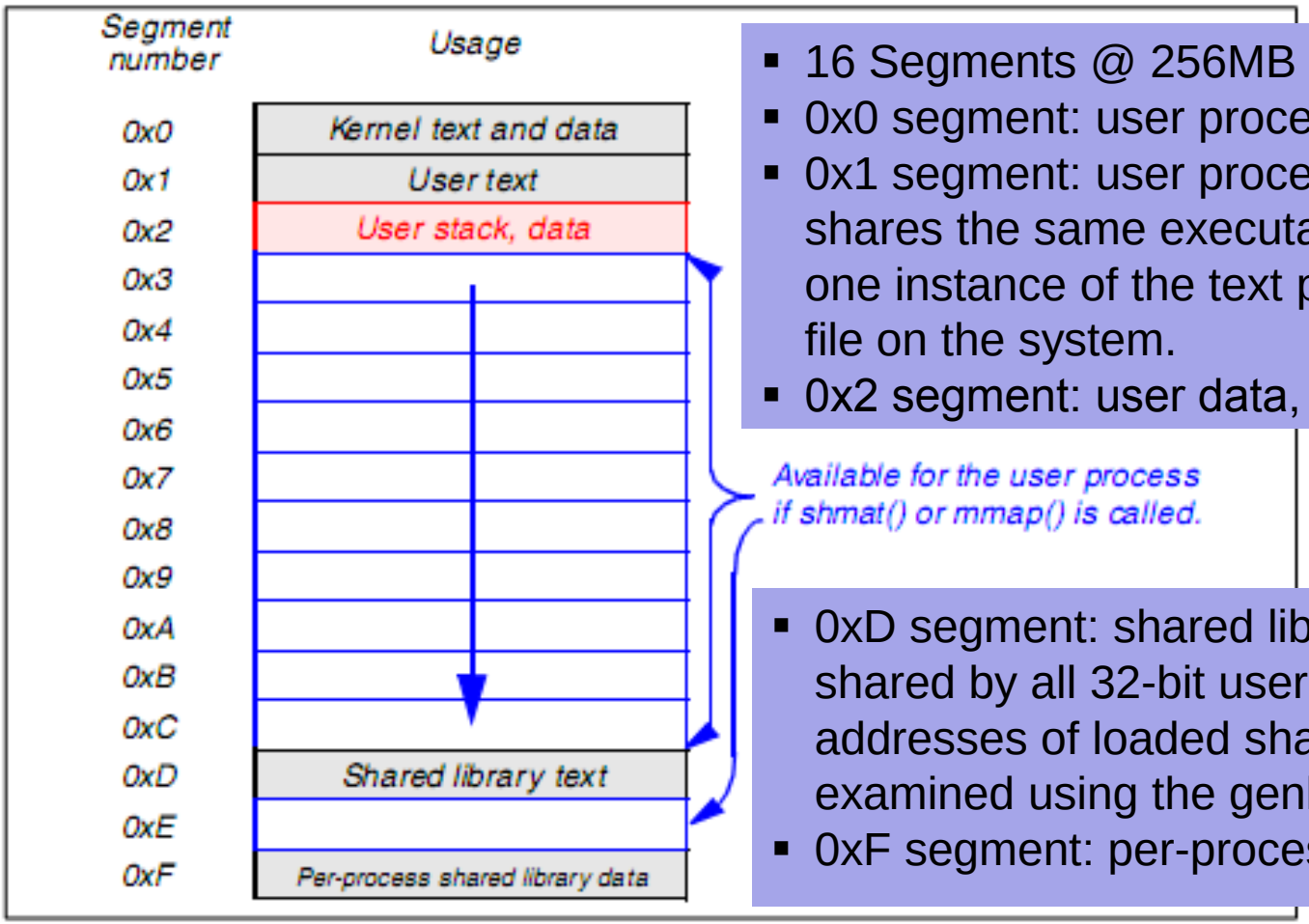


Figure 3-2 Default memory model (segment usage)[4]

- 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.

(Standard) Speichermodell von AIX (Detail)

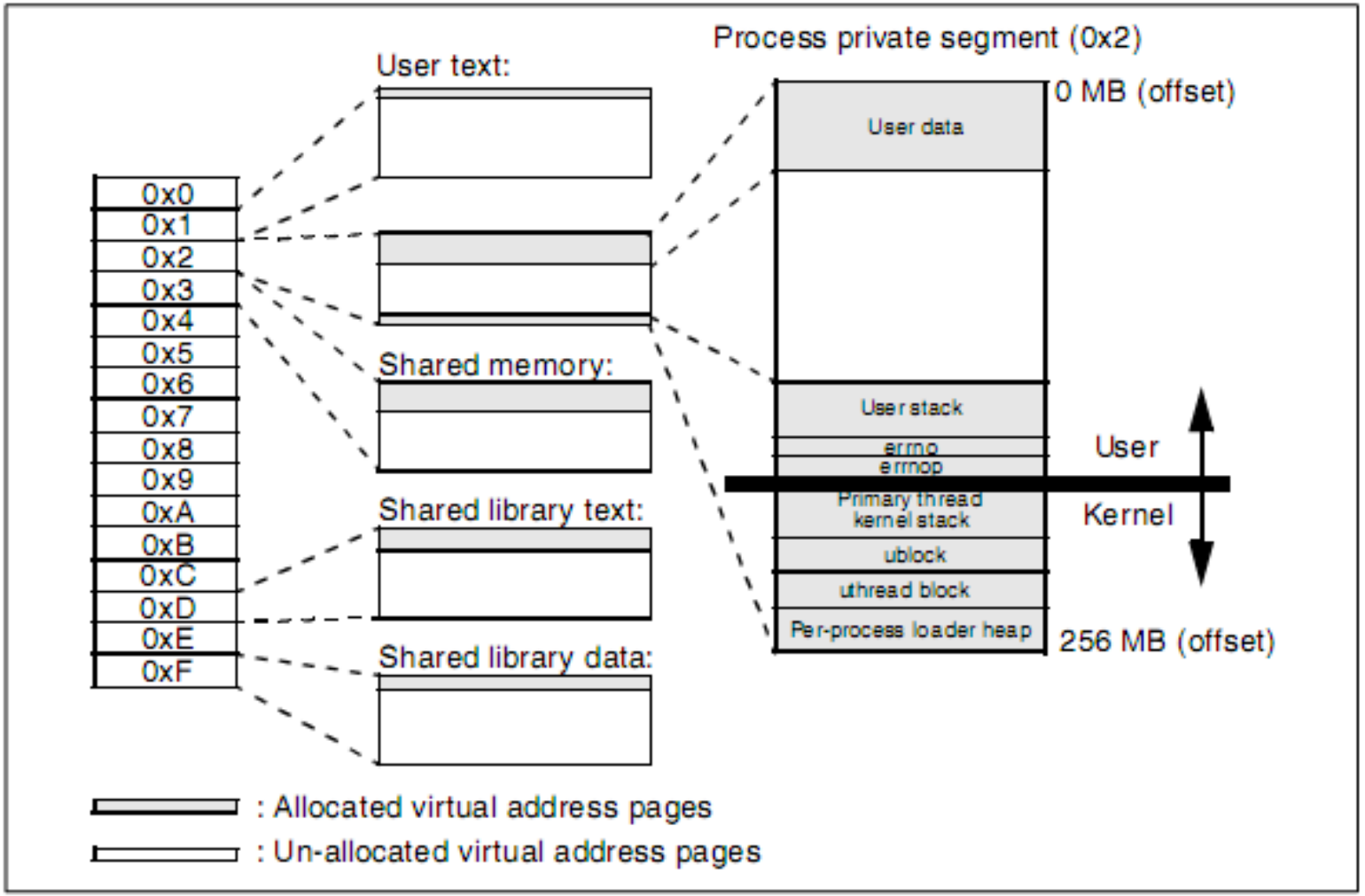


Figure 3-3 Default memory model (detail)

AIX Large Memory Model

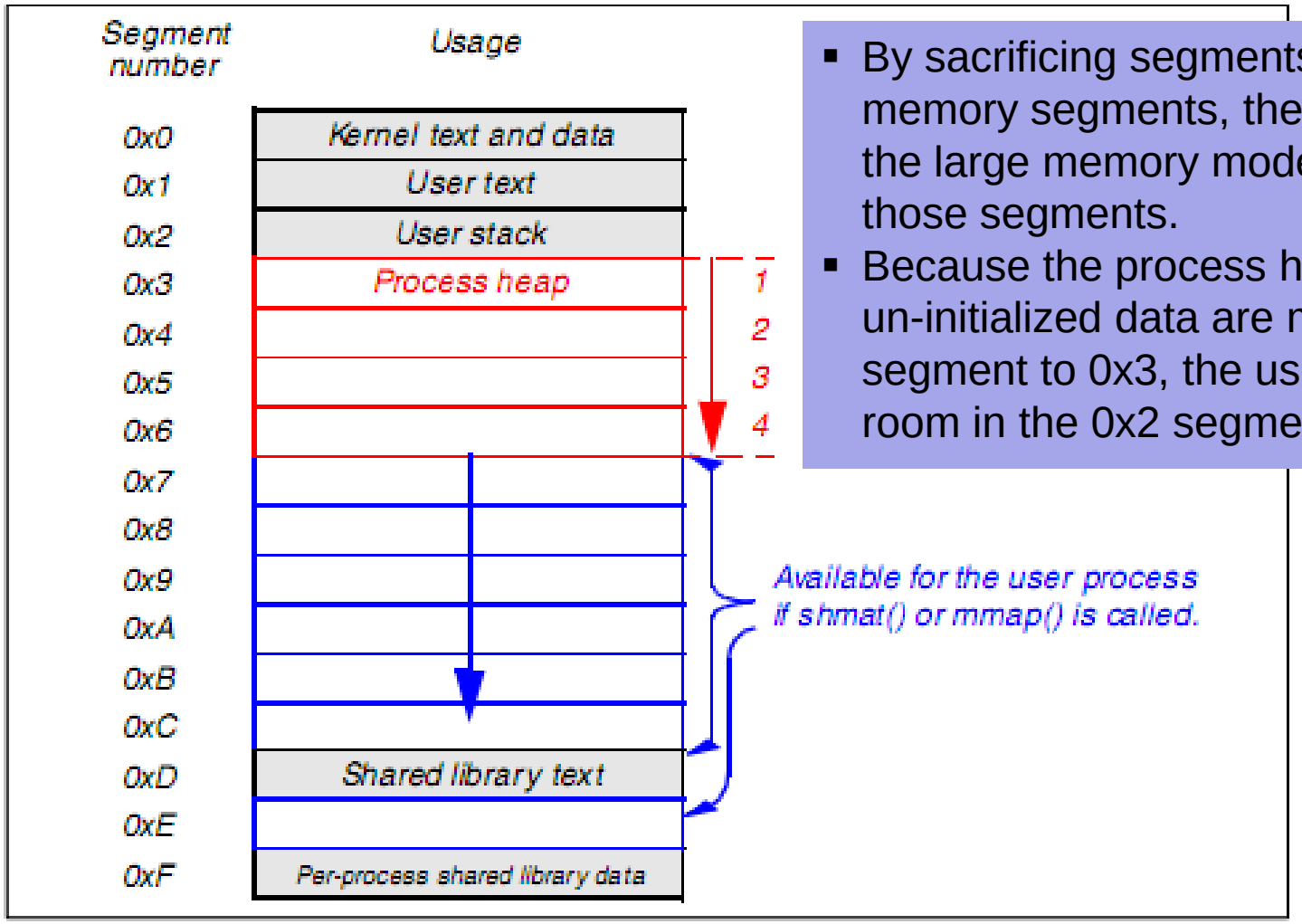


Figure 3-4 Large memory model (segment usage)

- 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.

AIX 64-bit Speichermodell

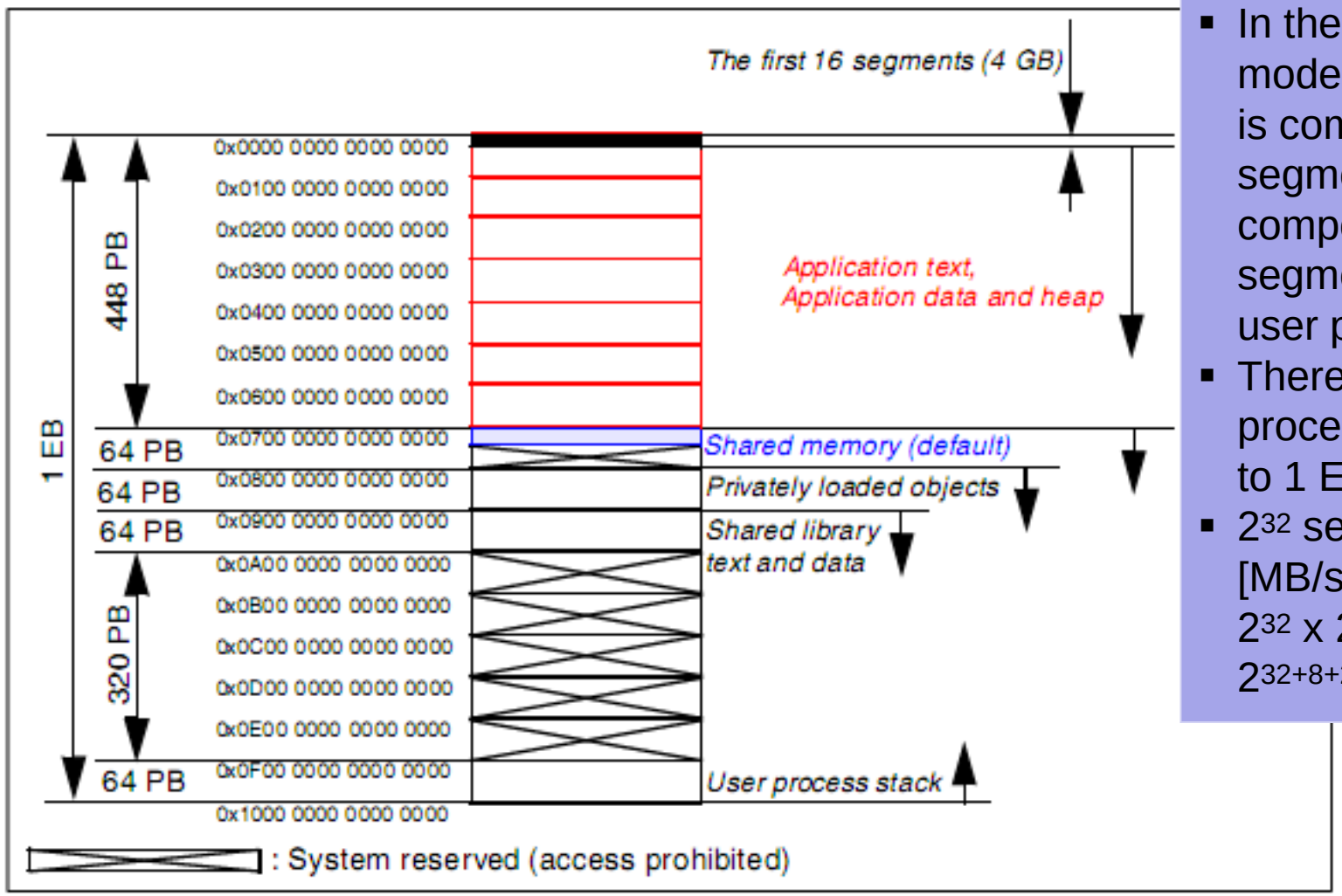


Figure 3-9 The 64-bit memory model (1EB)

- 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

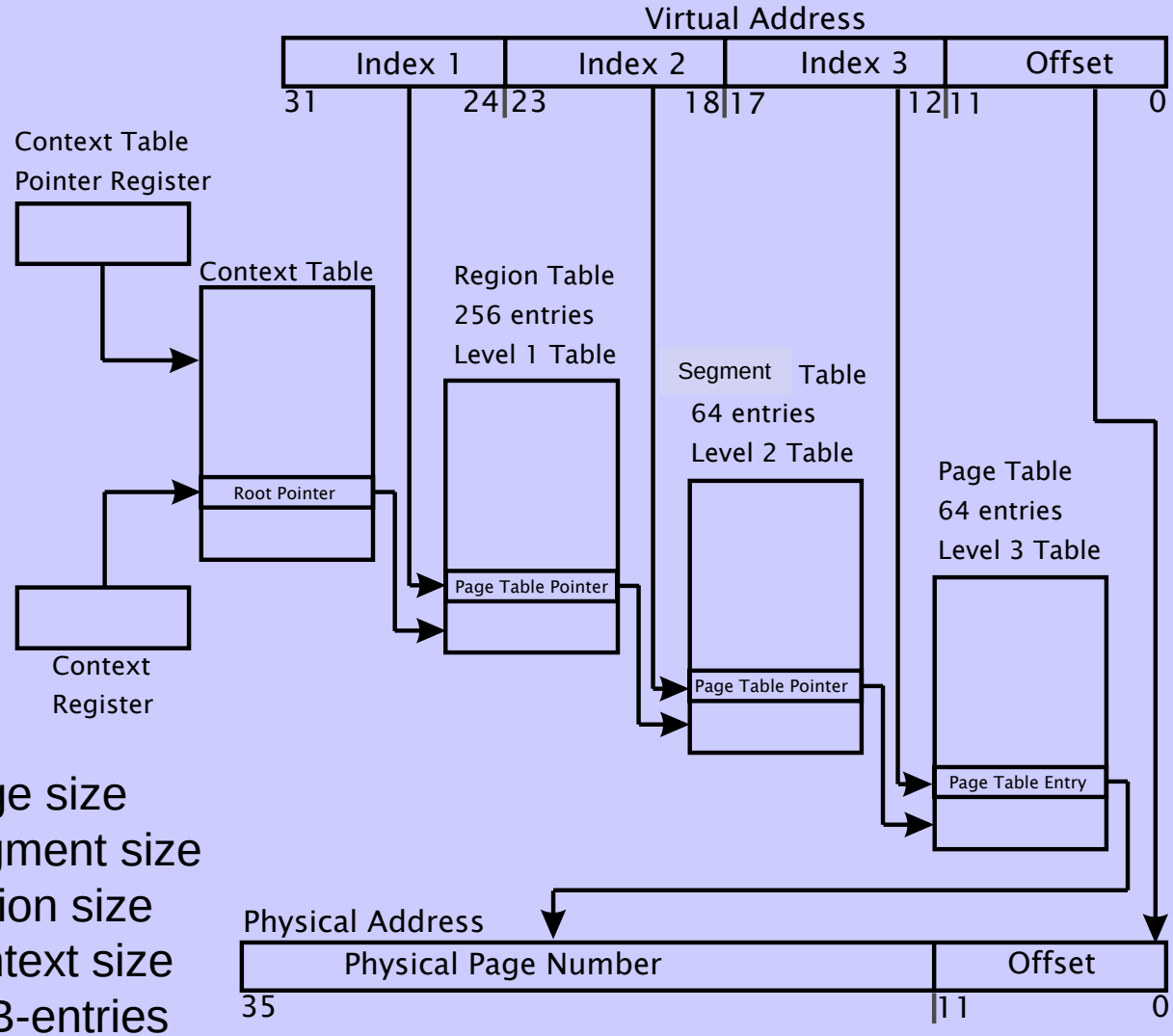
Seiten-basierte Speicherabbildung

- Abbildung / Verschiebung von Segmenten benötigt nur wenige Basisadressregister, ist aber grob-granular.
  - Es bleiben zahlreiche ungenutzte Lücken – „externer Verschnitt“
- MMU Hardware erlaubt Abbildung individueller virtueller Seiten (ca. 1 – 4 KB) auf physische Kacheln.
  - Jede Lücke ist Vielfaches der Seitengröße, damit nutzbar.
  - Es gibt internen Verschnitt – unvollständig genutzte Seiten.
- Jeder Prozess hat eigene Seitentabelle
  - vom Betriebssystem (BS) im Hauptspeicher (HSP) angelegt
  - von der MMU unabhängig genutzt
  - einstufig: eine einzige Tabelle mit page table entries (PTE)
  - PTE = <virtuelle Seitenadresse, physische Seitenadresse>

Seitentabellen für große Adressräume

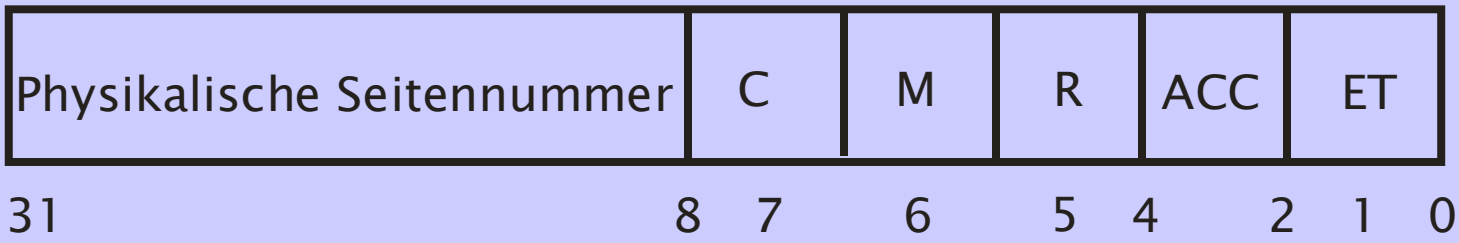
- Große Adressräume (insbes. 64-bit) sind großteils leer
- Einstufige Seitentabellen sind ungeeignet
  - 4 GB mit 4 KB großen Seiten → 1.000.000 Seiten
  - Seitenadresse hat mindestens (32-12)=20 Bits
  - Eintrag in Seitentabelle hat 3 – 4 byte Größe
  - Seitentabelle hat 4MB – hauptsächlich „invalid“ 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

Page Table Entry (PTE) of the 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}.
   - TLB = Translation Look-aside Buffer. Assoziativspeicher speichert die letzten *n* berechneten Adresspaare <VA, 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. (synchroner) Interrupt.
6. Aufruf der Seitenfehlerbehandlungsroutine (page fault handler) PFH im BS.

Behandlung von Seitenfehlern

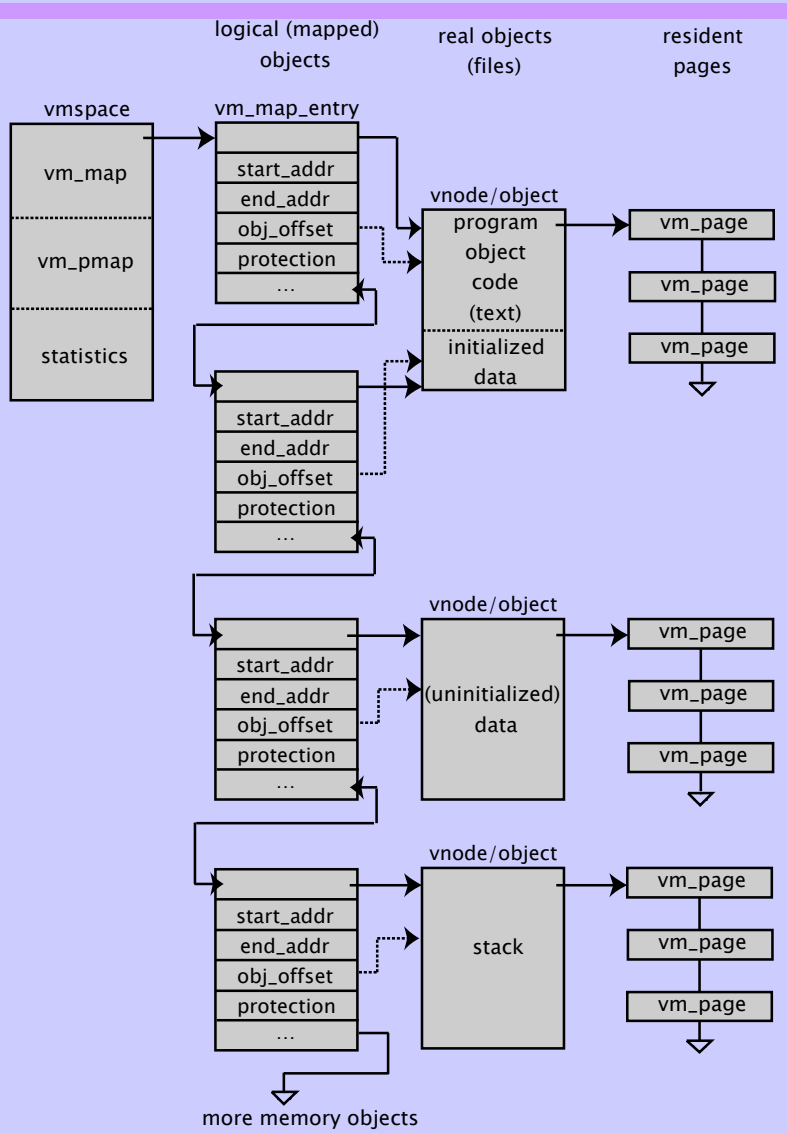
7. PFH ermittelt die verursachende virtuelle Adresse und Seite (aus MMU oder indirekt aus Instruktion).
8. PFH stellt fest, ob in den benutzten Bereich des virtuellen Adressraums adressiert wurde (=Pseudofehler) oder in den unbenutzten Bereich (=echter Fehler).
   - Hierzu wird eine Liste aller virtuellen 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), sende Signal SIGSEGV an Prozess und starte neues Scheduling.
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

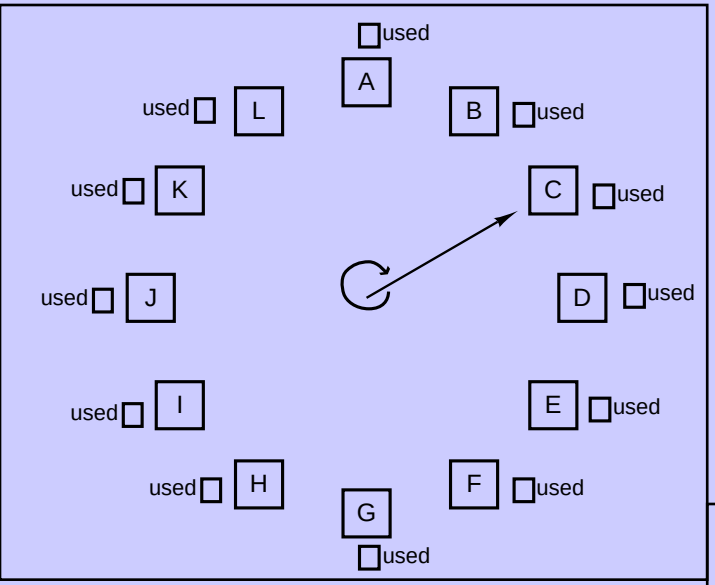


Page Fault Dispatch in 4.4 BSD

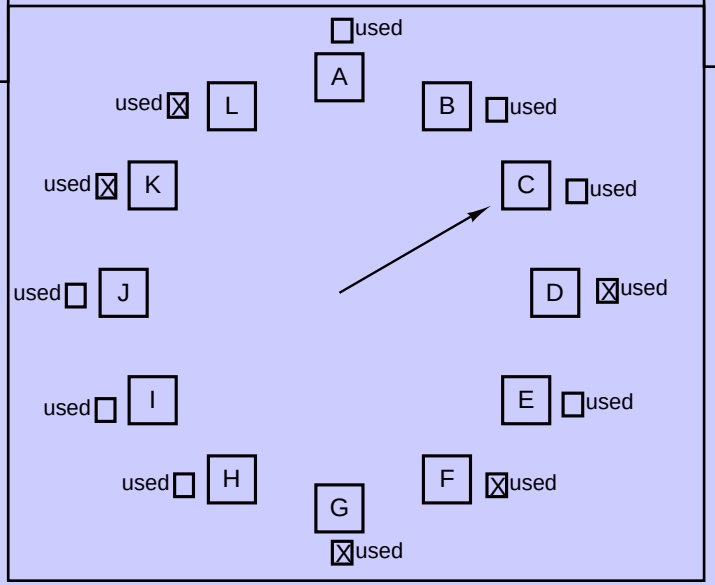
Virtual Address VA causes page fault.

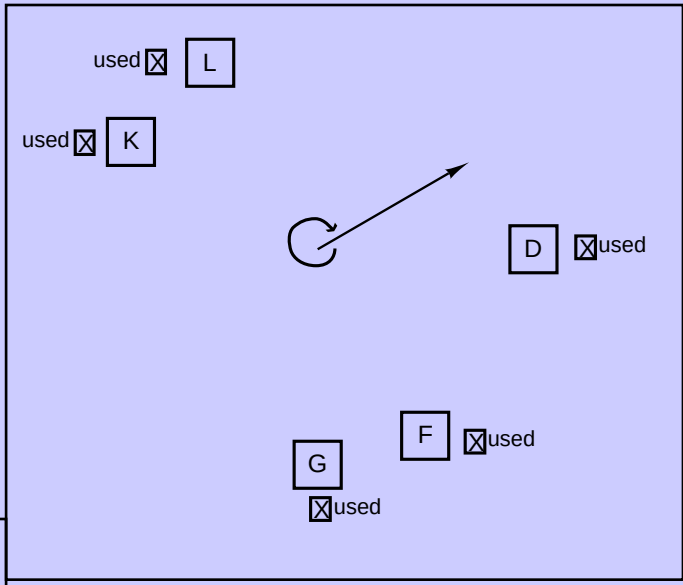
1. Find the vmspace structure and the vm_map_entry list
2. Check for each entry e whether start_addr <= VA <= end_addr. If no such e exists, send SIGSEGV signal to process.
3. Otherwise convert VA to offset within mapped object as follows: object_offset = VA – e->start_addr + e->object_offset
4. Present 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



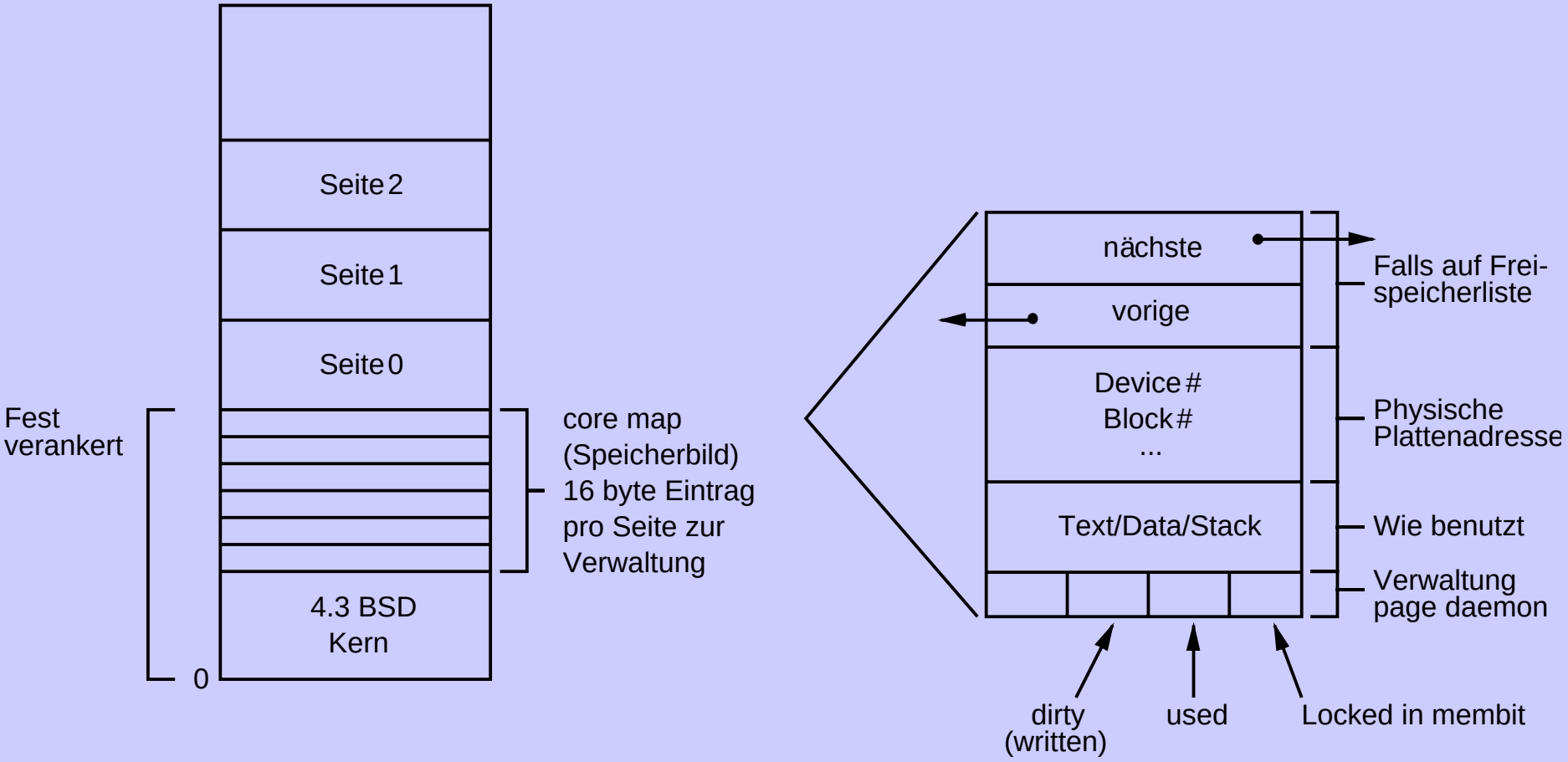
1. Durchlauf:
Alle „benutzt“ Bits werden zurückgesetzt



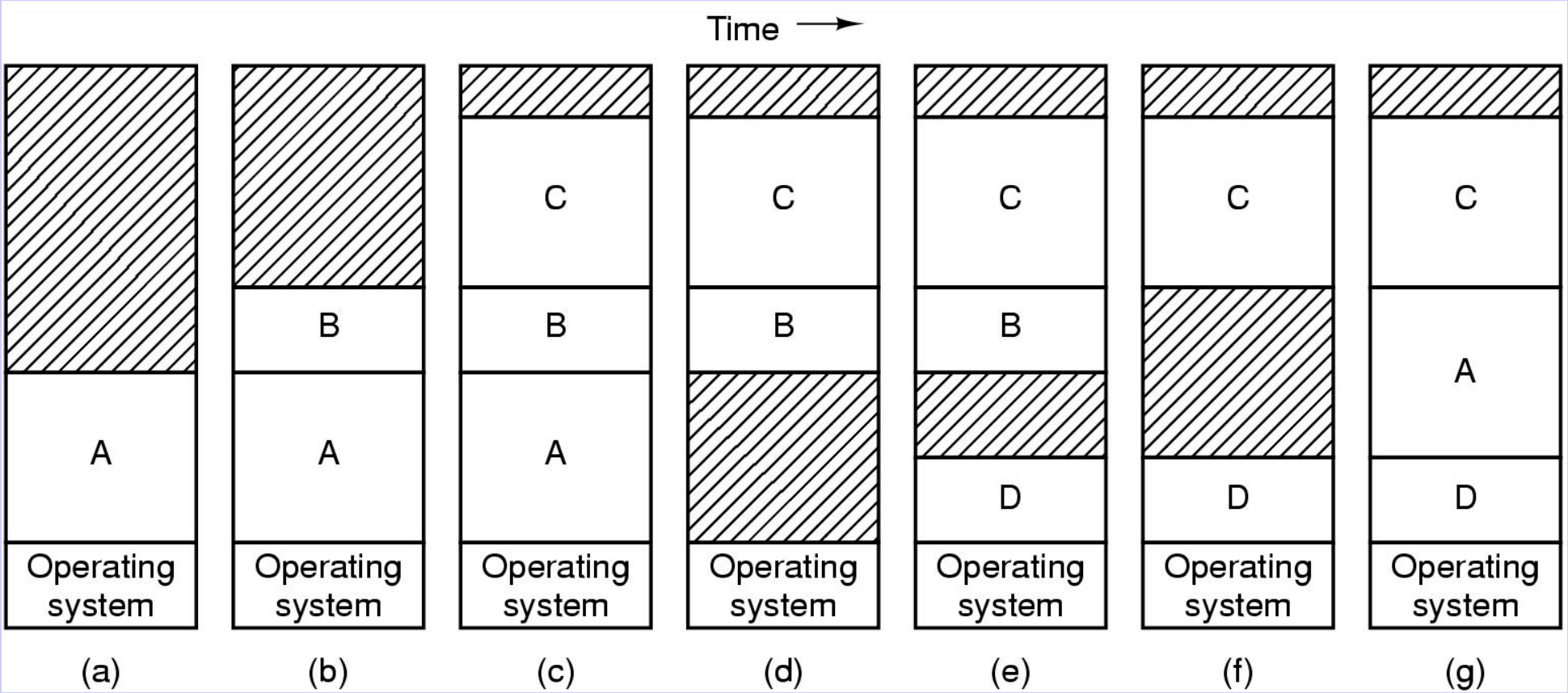


2. Durchlauf:
Alle unbenutzten Seiten werden freigesetzt.
(Schmutzige Seiten werden auf Platte zurückgeschrieben)

Seiten- (Kachel-)Einteilung 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[]);
```

argv is an array of argument strings passed to the new program. *envp* is an array of strings, conventionally of the form **key=value**, which are passed as environment to the new program. Both *argv* and *envp* must be terminated by a null pointer. argv[ ] and env[ ] can be accessed by the called program's main function, when defined as **int main(int argc, char *argv[ ], char *envp[ ])**. In case of failure, exec returns -1, else it does not return!

Execution of a File

- Steps in Exec
  1. reserve resources for new contents of virtual memory
  2. release old memory resources
  3. set up new memory map
- Setting up the memory map
  - After step 2 only a u.-structure and kernel stack remains
  - allocate a *vmspace*-structure and 4 *vm_map_entry* structures
    1. Copy-on-write, fill-from-file entry for text segment
       - (write to text occurs while debugging)
    2. Copy-on-write, fill-from-file entry for initialized data
    3. Anonymous zero-fill-on-demand entry for uninitialized data
    4. Anonymous zero-fill-on-demand entry for stack segment

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

Signale

- UNIX Signale sind ein reines Software-Konzept
  - modelliert nach Vorbild Interrupt + Interrupt-Handler
  - Prozess schickt Signal durch Systemaufruf
  - BS liefert Signal an Empfänger aus
  - BS aktiviert Signal-Handler des Empfängers
- Signale informieren einen Prozess über **asynchrone** Ereignisse
  - User (Software-)Ereignisse: Prozess verschickt Signal (via BS)
  - System-Ereignisse:
    - Software-Ereignis: Interrupt-Handler verschickt Signal (z.B. aufgrund Ctrl-C oder Ctrl-Z auf Tastatur, segmentation violation)
    - Hardware-Ereignis: Hardwarefehler (z.B. Bus-Error, illegal instruction, floating point exception)

Signale

Signal	Beschreibung	Aktion (default)
SIGABRT	Abnormal Program Termination	T
SIGALRM	Ablaufen eines Timers	T
SIGFPE	Floating Point Exception	T
SIGHUP	Abbruch Terminal-Verbindung (hangup)	T
SIGILL	Illegale Instruktion	T
SIGINT	Interrupt-Taste	T
SIGKILL	Prozess-Termination	T
SIGPIPE	Pipe ohne Leseprozess	T
SIGSEGV	Ungült. Speicherzugr. (segment. viol.)	T
SIGTERM	Default-Signal, Termination	T
SIGUSR1	Benutzerdef. Signal 1	T
SIGUSR2	Benutzerdef. Signal 2	T
SIGCHLD	Term/Stopp eines Kindes	I
SIGCONT	Starten eines gestoppten Prozesses	I
SIGQUIT	Quit-Taste	T
SIGSTOP	Stoppen eines Prozesses	S
SIGSTP	Stopp-Taste	S
SIGTTIN	Backg.-Proz. Liest vom Terminal	S
SIGTTOU	Backg.-Proz. Schreibt auf Terminal	S

T=Termination, S=Stop, I=Ignorieren **man –s 5 signal**

Signals

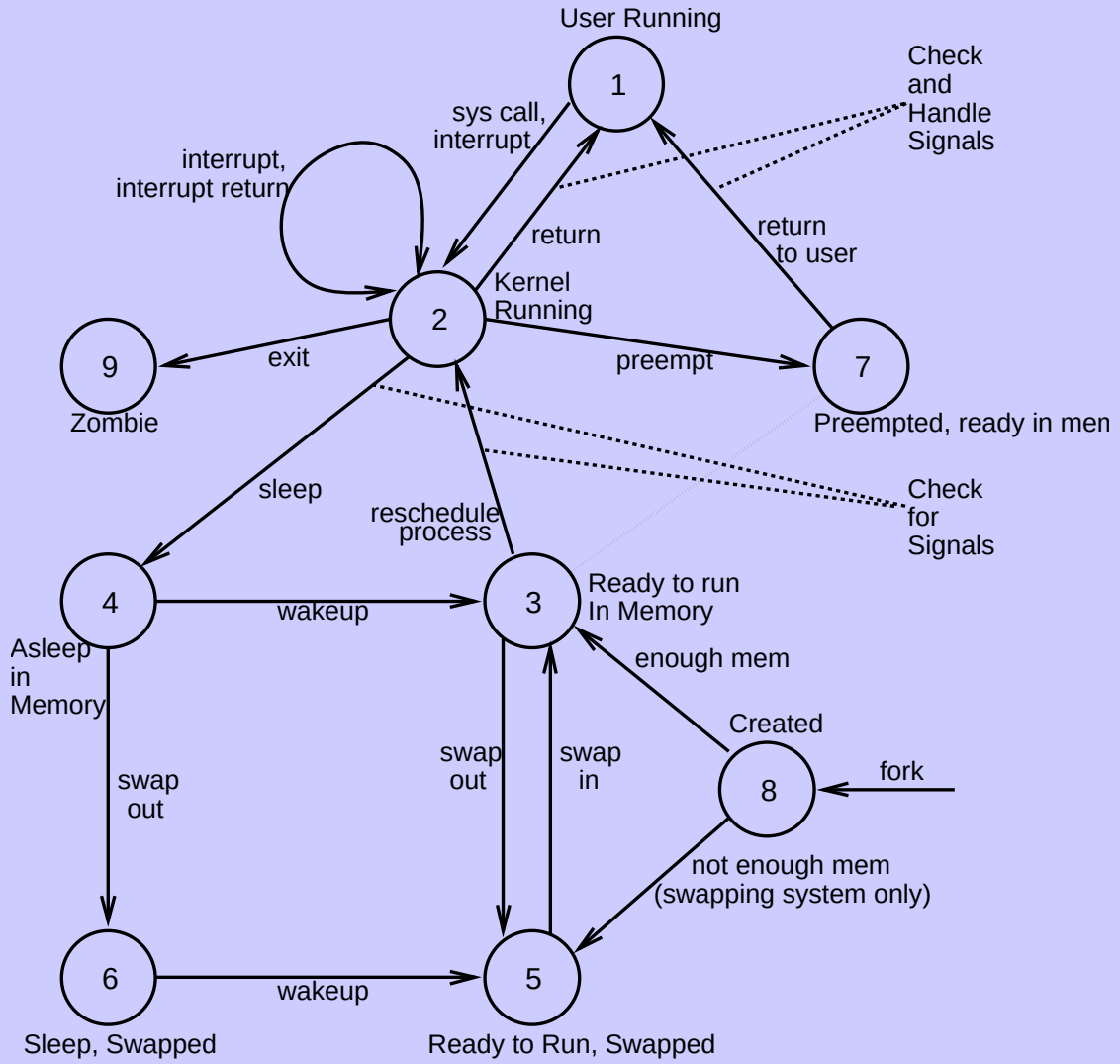
- UNIX Signals notify a process that a particular event has occurred (signal is *posted* to process)
  - hardware event (illegal instruction, floating point exception, …)
  - software event (stop request from terminal, …)
  - event notification by another process (kill() system call)
- Signals modelled **in Software** after HW-interrupts/traps

Hardware machine	**Software virtual machine**
instruction set	set of system calls
restartable instructions	restartable system calls
interrupts / traps	signals
interrupt / trap handlers	signal handlers
blocking interrupts	masking signals
interrupt stack	signal stack

Posting / Delivering / Handling Signals

- Steps in sending a signal
  1. Signal is generated by one process (or the operating system)
  2. Signal is *posted* by the operating system to another process
     a. add signal to the list of pending signals in the process table entry
     b. if signal is ignored (masked), or process sleeps non-interruptable, return
     c. make process runnable or cause it to be rescheduled
  3. Signal is *delivered*: 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 process.
  4. Signal is *handled*
  5. Normal execution resumes at the point of interruption

Prozess-Zustände in UNIX



Delivering signals

- Signals are delivered in the context of the receiving proc.
- A process checks for pending signals at least once each time it enters the system (by calling *cursig()*)
- Delivering a signal (through cursig())
  1. call *issignal()* to find first unmasked pending signal
  2. if there is no handler, perform default action and return
  3. mask current signal (and maybe others)
  4. Determine signal stack and save execution context there
  5. save argument list and set next instruction to a call of *signal-trampoline code;* return to user mode
  6. trampoline code runs and calls signal handler
  7. trampoline code calls kernel to restore original execution context

System calls: Posting and catching signals

- int kill (pid_t pid, int sig)
  - kill(2) schickt Signal sig an Prozess mit ID pid (auch: Signal an Prozessgruppe)
- Signal abfangen (*catch*)
  - Spezifikation eines Signalhandler
    - signal (int sig, handler)
  - Spezifikation einer Aktion: default action / ignoring the signal / catching with handler
    - int sigaction (int sig, sigaction *act)
- **SIGKILL** und **SIGSTOP** können nicht abgefangen werden.

Ignoring signals: masking

- Signals can be *masked* from delivery
  - sigprocmask() system call manipulates signal mask
    - *adds* / *deletes* signals in set, or *replaces* mask
    - int sigprocmask (int how, sigset_t *set)
- If a posted signal is masked
  - the signal is recorded in the set of pending signals
  - but no action is taken until the signal is *unmasked*

Weitere Signalfunktionen

- sigaltstack()
  - spezifiziere alternativen (user-defined) signal stack
- alarm (int secs);
  - schickt SIGALARM nach **secs** Sekunden an aufrufenden Prozess
- pause();
  - Blockiert den aufrufenden Prozess bis zu einem Signal
- sleep (int secs);
  - Blockiert **secs** Sekunden bzw. bis zu Signal
- sigsuspend()
  - blockiert bis zum Empfang eines Signals

Probleme mit Signalen

- Signalhandler asynchron zum normalen Programmfluss
  - Kann überall gestartet werden, wo Programm unterbrochen werden kann
    - MMU unterbricht mitten in Instruktion!
    - Datenstrukturen i.a. inkonsistent!
    - System-Calls durch Signale unterbrechbar!
- Signale nur mit 1 Bit gespeichert
  - Eintreffen wird nicht gezählt → Sign. kann verloren gehen
  - Nur 1 Bit Information wird übertragen
- Bearbeitungsreihenfolge unspezifiziert (keine zeitliche oder sonstige Priorität)

→ Probleme werden mit POSIX 4 z.T. behoben