Echtzeitbetriebssysteme

(Echtzeit-)Betriebssysteme

???

Echtzeitbetriebssysteme

Aufgaben und Anforderungen

Aufgaben eines Betriebssystems:

- die Ausführung der Benutzerprogramme,
- die Verteilung der Betriebsmittel (z.B. Speicher, Prozessor, Dateien), zu ermöglichen, steuern und überwachen.

Echtzeitbetriebssysteme

Aufgaben und Anforderungen

Betriebssystem stellt dem Benutzer die Sicht einer *virtuellen Maschine* zur Verfügung, die einfacher zu benutzen ist als die reale Hardware:

- **Aus Sicht eines Benutzers steht der Rechner ihm allein zur Verfügung.**
- **Einfacher, standardisierter Zugriff auf Ressourcen wie Speicher, Geräte, Dateien (Speichermanagement, Gerätetreiber, Dateisystem).**

Echtzeitbetriebssysteme

Aufgaben und Anforderungen

- **Zeitverhalten**
  - **Schnelligkeit**
  - **Bei einem RTOS insbesondere die Realisierung kurzer Antwortzeiten**
  - **Zeitlicher Determinismus (z.B. Speicherverwaltung/Garbage Collection ist problematisch)**
- **Geringer Ressourcenverbrauch**
  - **Hauptspeicher**
  - **Prozessorzeit**

Aufgaben und Anforderungen

- **Zuverlässigkeit und Stabilität**
  - **Programmfehler dürfen Betriebssystem und andere Programme nicht beeinflussen**

    VxWorks: standardmäßig bis 5.x keine getrennten Prozessadressräume!!!

    Linux: Treiber/Kernelmodule laufen im Kernel-Adressraum

    QNX: Mikrokernel-Architektur: sogar Treiber haben eigenen Prozessadressraum!!!

    OS9: optional!! gemeinsamer Prozessadressraum
- **Sicherheit**
  - **Dateischutz, Zugangsschutz**

Aufgaben und Anforderungen

- **Portabilität, Flexibilität und Kompatibilität von Systemkomponenten**
  - **Erweiterbarkeit von Systemen**
  - **Einhalten von Standards (z.B. POSIX)**
  - **Möglichkeit, für andere BS geschriebene Programme zu portieren (d.h. anpassen, übersetzen, auszuführen)**
- **Skalierbarkeit**
  - **Hinzunehmen oder Weglassen von BS-Komponenten ermöglichen**
  - **Geringer (Programm-, Daten-)Speicherbedarf („Footprint“) bei kleinen Anwendungen**
  - **Komfort und umfassende Funktionalität bei großen Anwendungen**

Aufgaben und Anforderungen

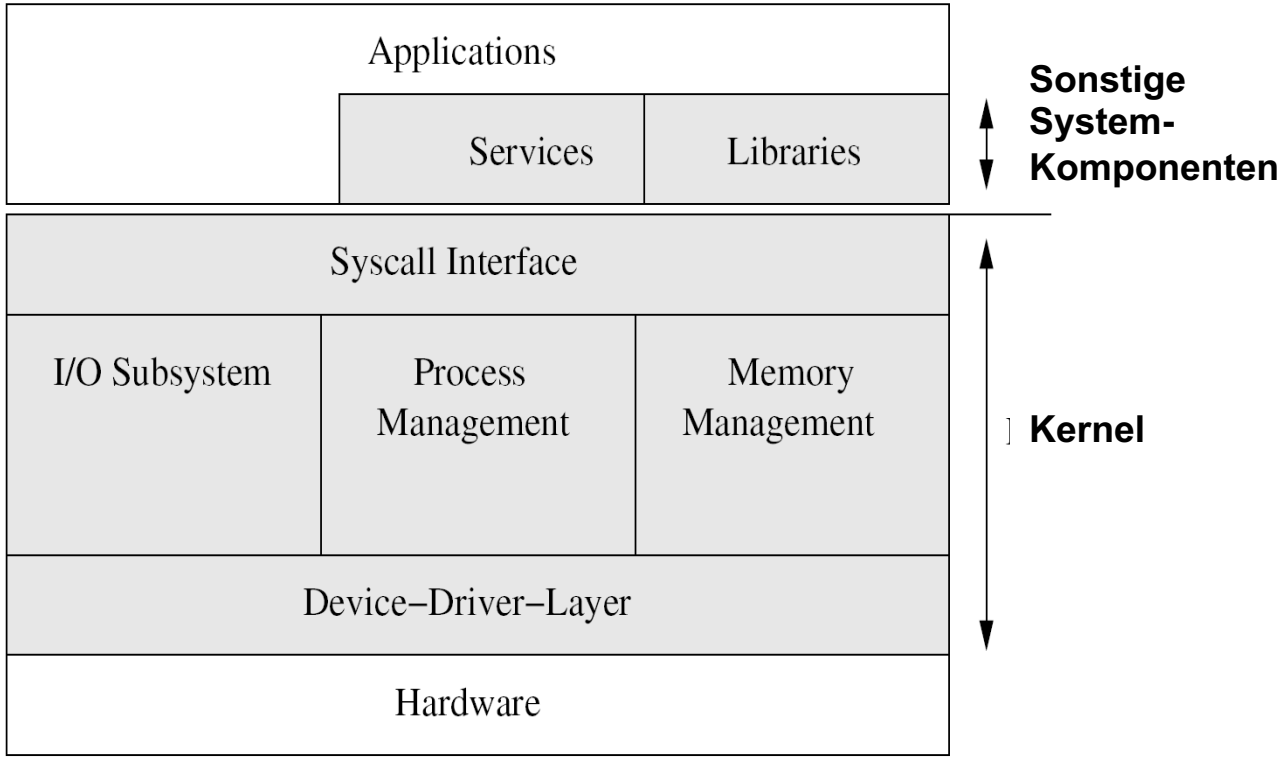
Ein Echtzeitbetriebssystem hat insbesondere die Aufgabe:

- **den deterministischen Ablauf zu garantieren**
  - **Scheduling**
  - **IPC und Synchronisation**
  - **Speichermanagement: kein Swapping/Paging, keine Garbage Collection**
- **die Angabe und das Einhalten von Zeitbedingungen zu ermöglichen (Zeitdienste, späteres Thema)**
- **Unter obigen Bedingungen die Ressourcenverteilung sicherzustellen**
- **Erweiterbarkeit (Prozessanbindung über neue Hardware) einfach gewährleisten**

Aufbau und Struktur

Betriebssystem besteht aus

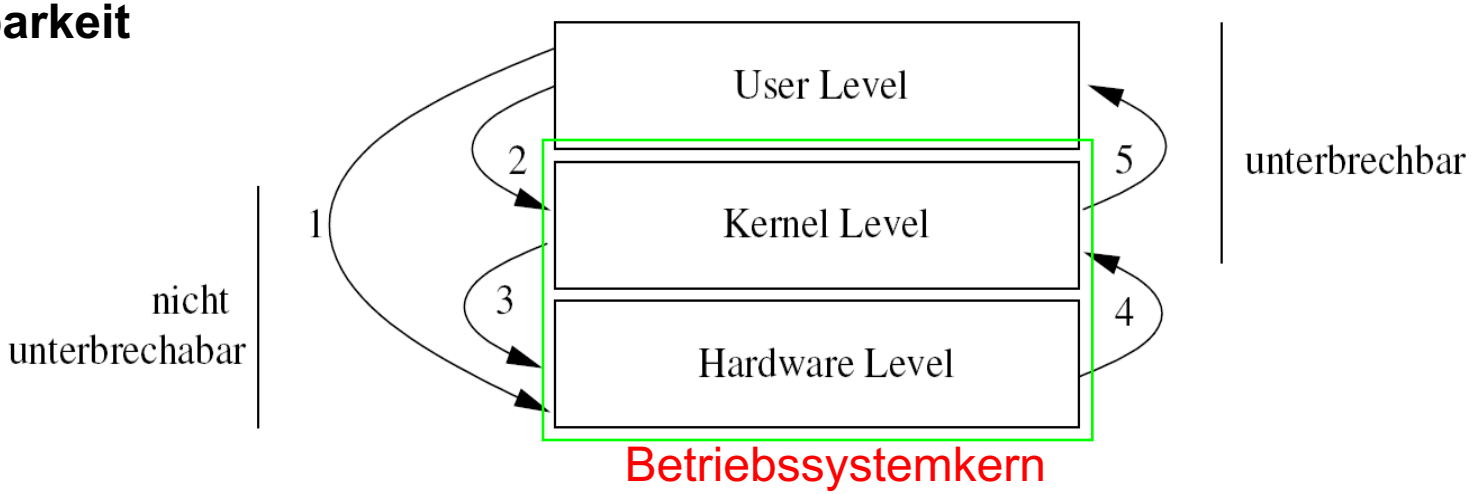
- **auf BS-Kern aufbauenden Systemkomponenten (Dienstprogramme, Werkzeuge, ...)**
- **Betriebssystemkern**



Echtzeitbetriebssysteme

Aufbau und Struktur

Unterbrechbarkeit



Achtung: BS-Dienste werden (fast) bei jedem BS über SW-Interrupts (SVC: *supervisor call*, *system call*) angefordert

1 HW–Interrupt

2 SW–Interrupt (Systemcall)

3 HW–Interrupt (während eines Systemcalls)

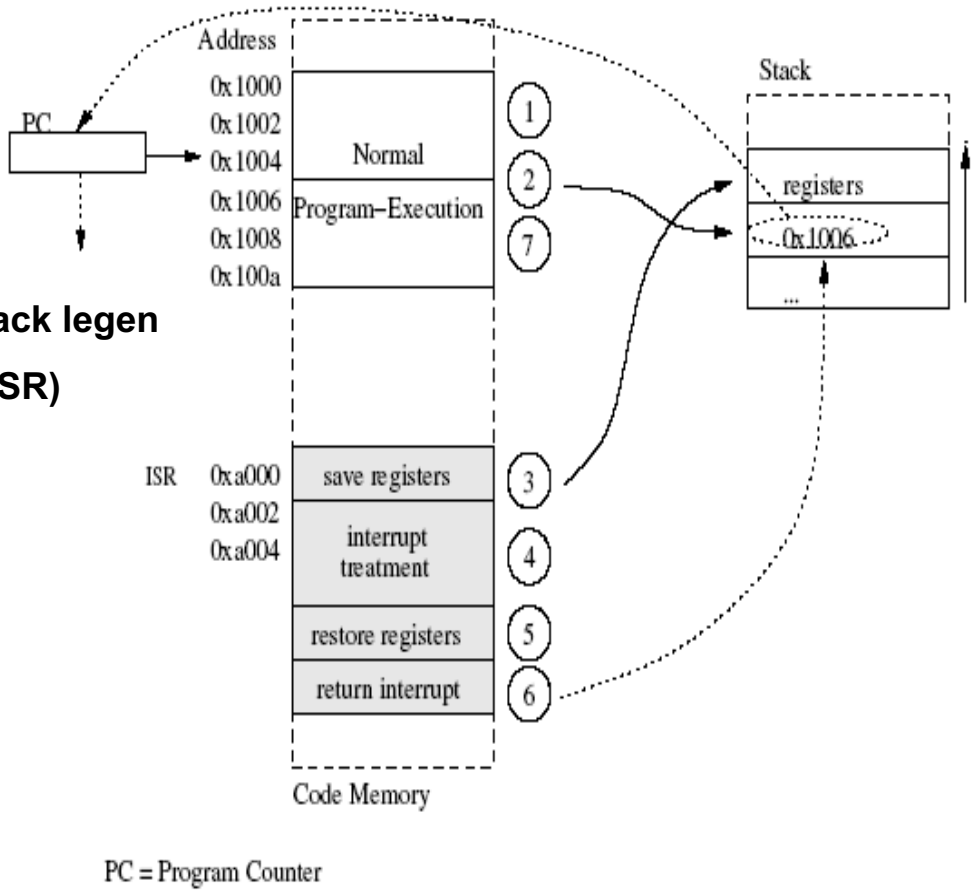
4 HW–Interrupt beendet (Scheduler wird aufgerufen)

5 Scheduler übergibt CPU einer Task auf User–Ebene

Prozessmanagement

Unterbrechung (ohne BS)

1. Die CPU arbeitet ein Programm ab.
2. Interrupt während der Programmabarbeitung
   Abarbeitung aktuellen Befehls beenden.
   Befehlszähler und Registerinhalte auf den Stack legen
   Befehlszähler auf Interrupt Service Routine (ISR)
3. ISR rettet von ihr benötigte CPU-Register
4. Eigentliche Interrupt-Behandlung
5. Gerettete CPU-Register restaurieren
6. Befehl “Return from Interrupt”
   Auf Stack abgelegte Register
   (Flags, Program Counter) restaurieren
7. Normalen Programmablauf fortsetzen



PC = Program Counter

Prozessmanagement

Unterbrechung (*interrupt*)

Zwei Arten von Interrupts: *Softwareinterrupts* und *Hardwareinterrupts*.

- **Über Softwareinterrupts (Systemcalls) fordern Benutzerprogramme Dienste des Betriebssystems an**
- **Über Hardwareinterrupts fordern Hardwarekomponenten (Systemuhr, Platten, Modem usw.) Dienste des Betriebssystems an.**

Echtzeitbetriebssysteme

Prozessmanagement

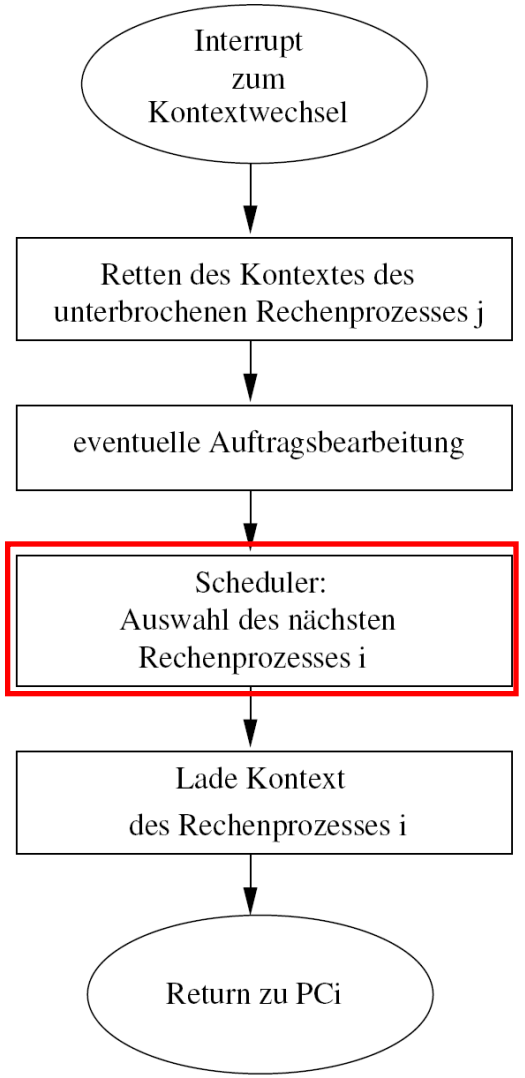
Unterbrechung (Multitasking-BS) mit präemptivem Scheduling

Präemptiv (RT-BS):

Rechnerkern wird bei Interrupt der aktuell rechnenden Task entzogen, wenn höherpriore Task auf Interrupt reagieren muss

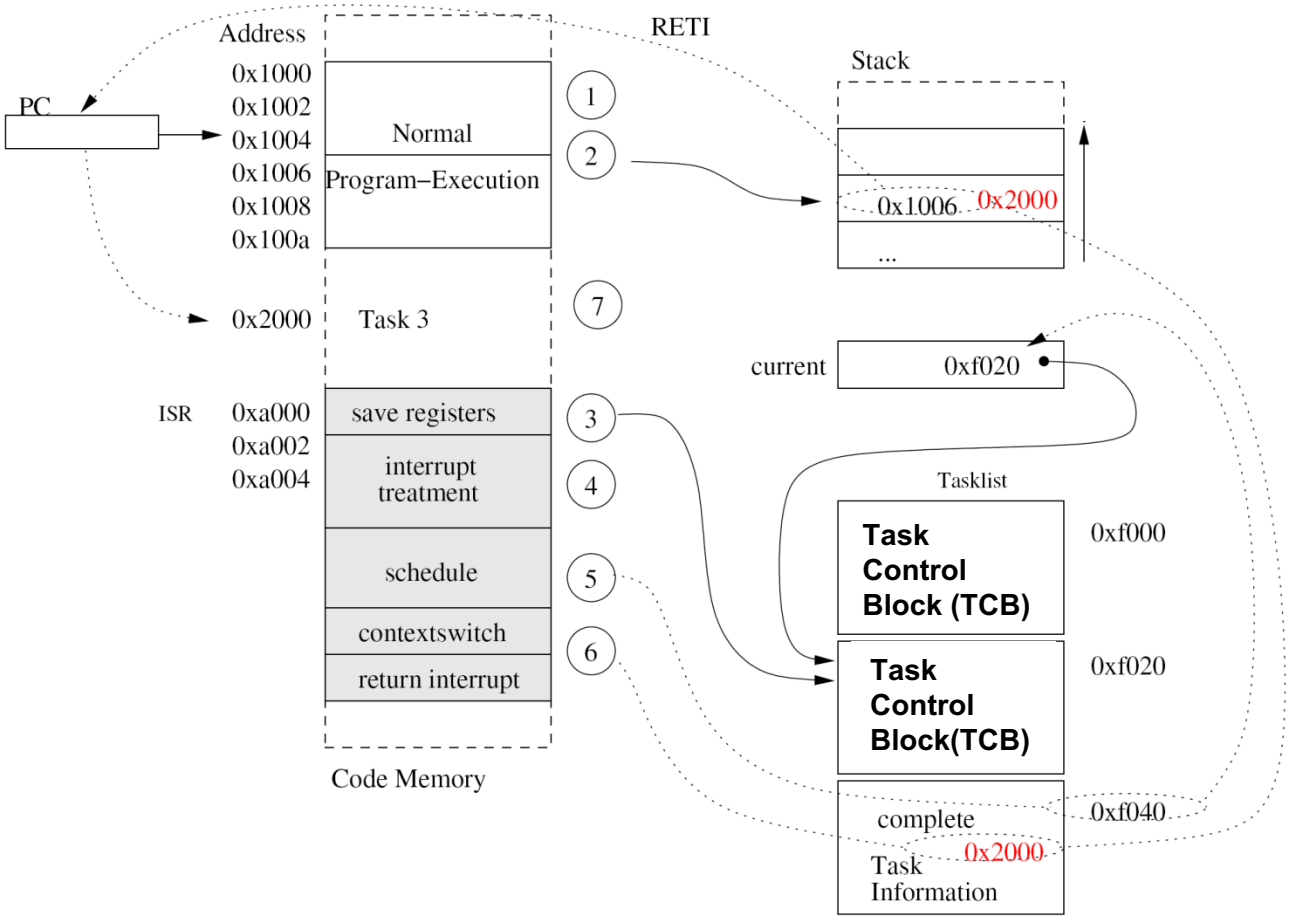
Scheduling nicht-präemptiv (Standard-BS):

Scheduling nur bei System Call oder zeitgesteuert, nicht bei jedem Interrupt



Prozessmanagement

Unterbrechung Multitasking-BS mit präemptivem Scheduling



Prozessmanagement

Task Control Block

Beinhaltet vollständige Information, die zu einer Task gehört

Prozessmanagement

Task Control Block

Beinhaltet vollständige Information, die zu einer Task gehört

- **Prozess ID (→ eineindeutig im Gegensatz zu Prozessnamen)**
- **Priorität**
- **Task-Zustand (+Bedingungen, auf die die Task wartet)**
- **Zeit-Quantum („erzeugte Prozessorlast“; Summe, in letzter Zeiteinheit)**
- **Maschinenzustand (Register, Stack, ...)**
- **Verwaltungsdaten für Betriebsmittel (Filedeskriptoren,...)**
- **Speicherabbildungstabellen virtueller Speicher (Prozessadressraum) -> realer Speicher (code, data, stack)**

Echtzeitbetriebssysteme

OS9 Processdescriptor:

```
struct prdsc {
   u_int32 p_sync;                  /* process descriptor sync code    */
   process_id  p_id,          /* process id                          */
        p_pid;                /* parent's process id                 */
   owner_id p_owner;          /* group/user numbers                  */
   u_int32 p_rsrv1[4];        /* reserved space          */
   Mh_exec p_pmodul;          /* primary module pointer */
   u_int32 p_pdsize;          /* size of process descriptor          */
   u_int16 p_prior,                 /* priority                                */
        p_age;                /* age                                 */
   u_int32 p_sched,                 /* process scheduling constant*/
        p_state,              /* process status bit flags            */
        p_queueid,            /* current queue id                    */
        p_preempt,            /* system-state preemption flag, 0 = switchable*/
        p_srstat,                   /* process service request capability status
        */
        p_srmask;                   /* process service request mask    */
```

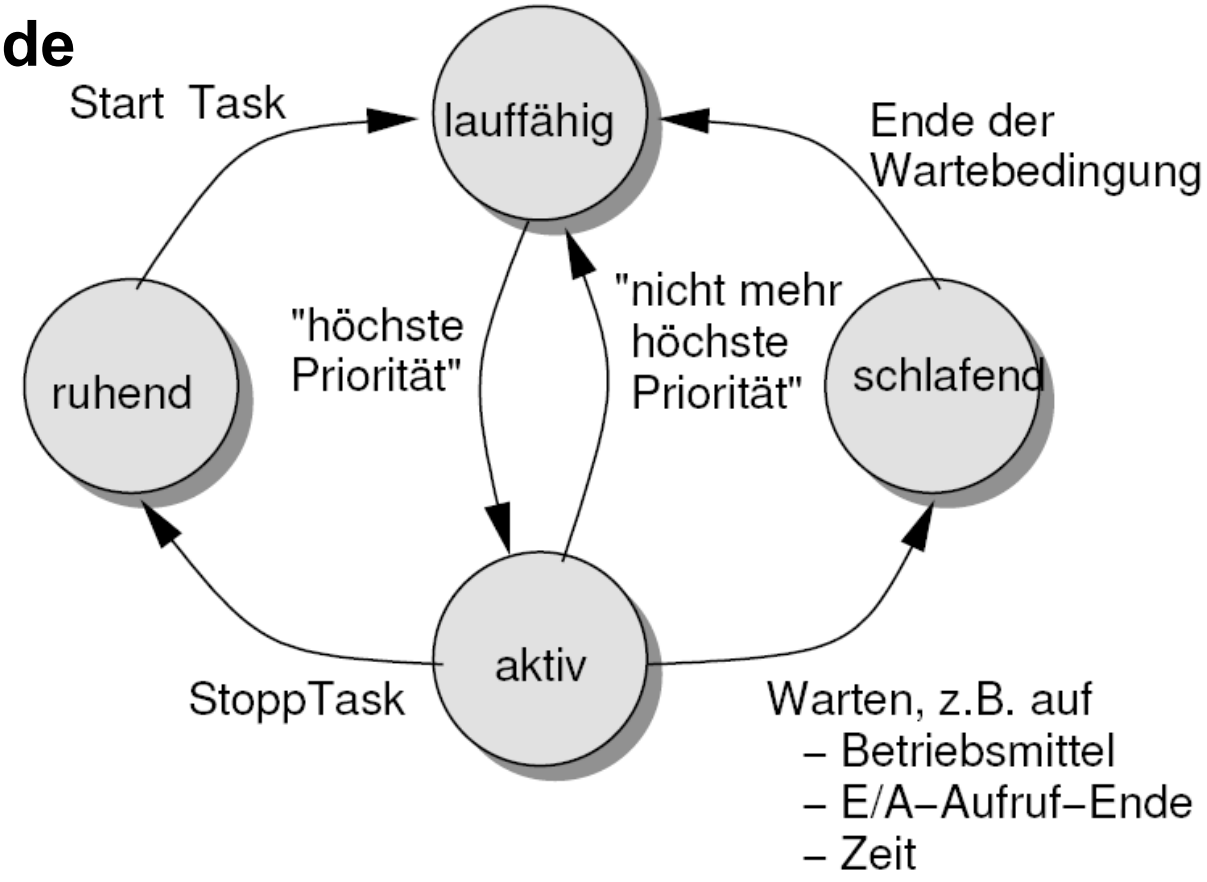
Echtzeitbetriebssysteme

```
.....
error_code p_status;    /* exit (error) status of condemned process */
 u_int32 p_timbeg,      /* time when forked in seconds from system ref. date*/
         p_uticks,      /* user state ticks elapsed                  */
         p_sticks;      /* system state tick elapsed                 */
Regs
         p_sp;          /* system stack pointer   */
u_char
         *p_usp,        /* user stack pointer     */
         *p_excpsp;     /* system state exception recovery stack*/

Mod_dir
        p_mdir,         /* process' current module directory */
        p_altmdir,      /* process' alternate module directory */
        p_smdir;        /* process' current shared module directory */
….
```

Prozessmanagement

Task-Zustände



Prozessmanagement

Tasks und Threads

???

Prozessmanagement

Tasks und Threads

Auch: Leichtgewichtige Prozesse (*light weight processes*)

Ziel: Aufwand für Kontextwechsel minimieren

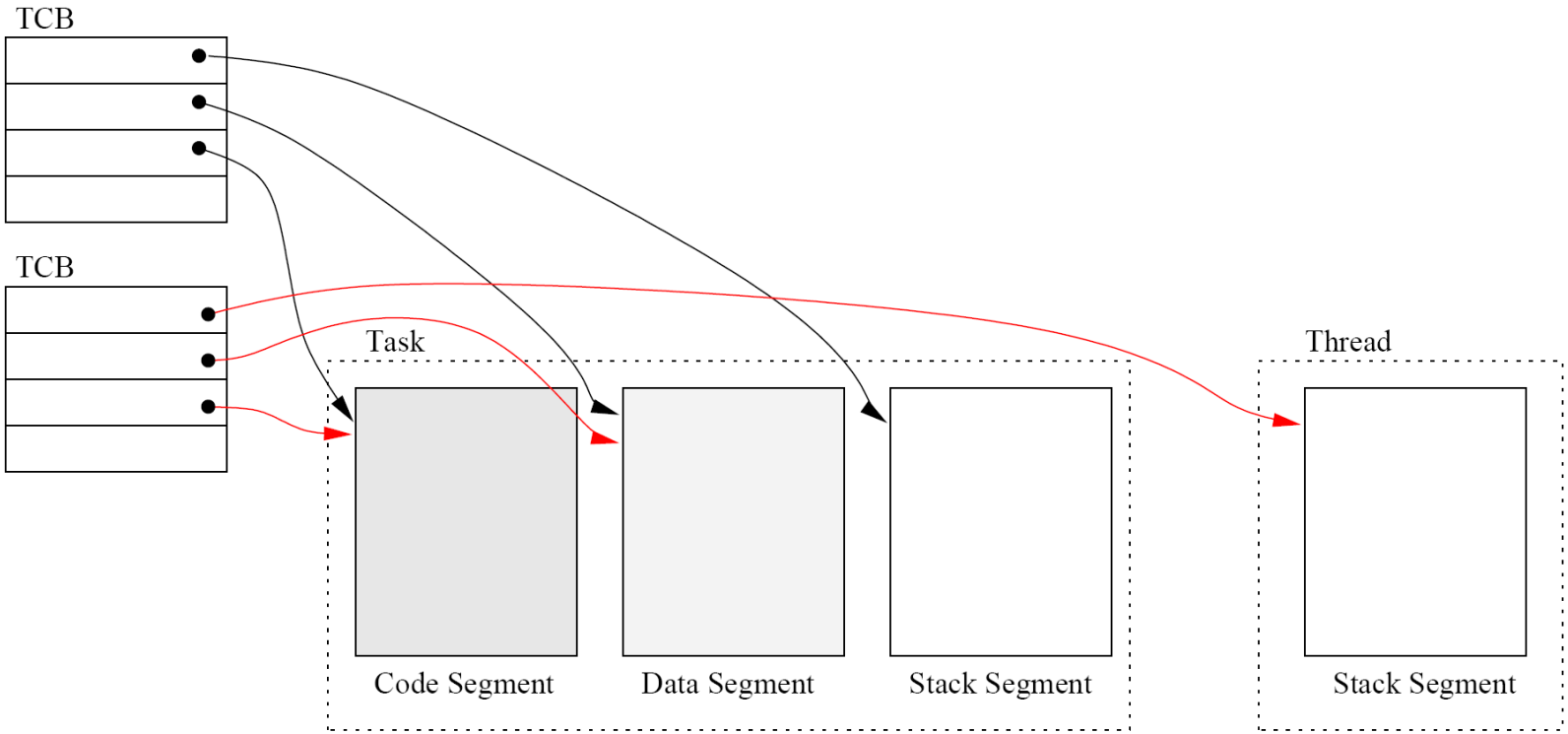
Mehrere Ablaufeinheiten (Threads) teilen sich fast den kompletten Task-Kontext

Lediglich Stack (incl. PC) und Thread-Status ist unterschiedlich

- **Effizient zu erzeugen und zu schedulen**
- **Gemeinsamer (kein getrennter) Prozessadressraum**
- **Gemeinsame Betriebsmittel (Files, Devices, ...)**
- **Gemeinsamer „globaler Speicher“ wird oft aus Effizienzgründen (und Faulheitsgründen) verwendet (missbraucht)**

Prozessmanagement

Tasks und Threads



Echtzeitbetriebssysteme

Prozessmanagement: Tasks und Threads: Task (Prozess) erzeugen Linux…

```
#include <stdio.h>

void parent()
{
printf(''The parent process has ID %d\n'', getpid() );
}

void child()
{
printf(''The child process has ID %d\n'', getpid() );
return;
}

int main( int argc, char **argv )
{
if( fork()!=0 ) {
parent();
wait();
} else {
child();
}
printf(''Exit process %d\n'', getpid() );
exit( 0 );
}
```

Prozessmanagement: Tasks und Threads: Task (Prozess) erzeugen OS9

```
Includes: <const.h> ,<process.h> ,<errno.h>,<types.h> ,<cglob.h>,<stdlib.h> ,<const.h>

main() {
error_code myerr;
u_int32    priority = 0;/* gleiche Priorität wie der Aufrufer */
u_int32    paths =3;    /* die ersten drei offenen Pfade vererben*/
u_int32    edata = 0;   /* kein zusätzlicher Platz */
process_id child_id;    /* enthält child-procID nach os_exec() Aufruf */
char       orphan =0;   /* normales child!! */
u_int32    typelang;    /* Type des zu ladenden Moduls */
static char     * arguments [ ] = { "meinprogramm", "erstesArg", "zweitesArg", NULL};
                    /* Aufruf via mshell waere: meinprogramm erstesArg zweitesArg */
typelang = mktypelang (MT_PROGRAM, ML_OBJECT);

if ( (myerr = _os_exec (_os_fork, priority, paths, arguments[0], arguments, _environ,
    edata, &child_id, typelang, orphan) ) != SUCCESS) exit (myerr);

    oder kürzer:

if ( (myerr = _os_exec (_os_fork, 0, 3, arguments[0], arguments, _environ, 0,
     &child_id, mktypelang (MT_PROGRAM, ML_OBJECT), 0) ) != SUCCESS) exit (myerr);
…}
```

Prozessmanagement

Tasks und Threads: Thread erzeugen (Linux, OS9…)

```
#include <stdio.h>
#include <pthread.h>

void *ThreadFunction( void *args )
{
printf( "%s - I am fine!\n", (char *)args );
pthread_exit( NULL );
}

main( int argc, char **argv )
{
pthread_t MyThread;
char *Hello="How are you, Thread?";

if( pthread_create( &MyThread, NULL, ThreadFunction, Hello )!= 0 ) {
fprintf(stderr,"creation of thread failed\n");
exit( -1 );
}
/* Now the thread is running concurrent to the main task */
printf("Here is the main task!\n");
/* waiting for the termination of MyThread */
pthread_join( MyThread, NULL );
}
```

POSIX.1-2017 API (Threads)

```
int pthread_create( pthread_t * thread,                  /* OUT */
                    pthread_attr_t * attr,               /* IN  */
                    void *(*start_routine)(void *),      /* IN  */
                    void * arg);                         /* IN  */
```

Erzeugt neuen Thread (Einstiegsfunktion: `start_routine` mit Argument `arg`)

Thread wird nebenläufig mit aufrufender Thread abgearbeitet

- Beendigung des Threads
- Aufruf von `pthread_exit`

Beendigung von `start_routine`

Attribute

- Scheduling (Art, Parameter)
- Stack (Größe, Adresse)
- Sollen andere Threads auf Beendigung dieser Thread warten können?

POSIX.1-2017 API (Threads)

```
void pthread_exit( void *retval );     /* IN  */
```

Beendigung einer Thread mit Ergebnis `retval`

Alternative zu Beenden der `start_routine`

Cleanup-Handler aufrufen (Ressourcen freigeben: Speicher, Filedeskriptoren,...)

POSIX.1-2017 API (Threads)

```
int pthread_join (  pthread_t thread ,          /* IN  */
                    void ** thread_return ); /* OUT */
```

Aufruf blockiert, bis der Thread, auf den gewartet wird, sich beendet

Ergebnis des sich beendenden Threads in steht in `thread_return`

POSIX.1-2017 API (Threads)

```
int pthread_detach( pthread_t th );
```

Wenn keiner auf einen Thread warten will, kann der sich bei ihrer Beendigung gleich komplett aufräumen (Thread Deskriptor, Stack)

Mit `pthread_detach` sagt man ihr das.

Echtzeitbetriebssysteme

• **OS9: procs dient zur Anzeige der laufenden Tasks im System. Ohne weitere Parameter zeigt procs die Tasks des eingeloggten Benutzers an. procs -e zeigt alle im System befindlichen Tasks an:**

$ procs -e

Id	PId	Thd	Grp.Usr	Prior	MemSiz	Sig	S	CPU Time	Age	Module & I/O
2	5	1	0.0	128	38.00k	0	w	0.02	???	mshell <>>>term
3	0	1	0.0	128	103.50k	0	s	0.05	???	inetd <>>>nil
4	0	1	0.0	128	16.25k	0	e	0.02	???	spf_rx
5	0	1	0.0	128	6.25k	0	w	0.01	???	sysgo <>>>term
6	0	1	0.0	128	6.50k	0	s	0.00	???	ndpio <>>>nil
7	9	1	0.0	128	66.50k	0	*	0.08	0:00	procs <>>>term
8	0	1	0.0	128	26.25k	0	s	0.02	???	spfndpd <>>>nil
9	2	1	0.0	128	38.00k	0	w	0.13	???	mshell <>>>term

Id	Process ID
PId	Parent process ID
Thd	Thread count
Grp.usr	Owner of the process (group and user)
Prior	Initial priority of the process
MemSiz	Amount of memory the process is using
Sig	Last pending signal value for the process/exit status for dead process
CPU Time	Amount of CPU time the process has used
Age	Elapsed time since the process started

S	Process status: * Currently executing w Waiting s Sleeping a Active e Waiting on event m Waiting for an mbuf z Suspended process
Module & I/O	Process name and standard I/O paths: < Standard input > Standard output >> Standard error output If several of the paths point to the same pathlist, the identifiers for the paths are merged.

Echtzeitbetriebssysteme

- **OS9:procs -a zeigt weitere Informationen an:**

```
Id Pld Thd Aging  F$calls I$calls Last     Read Written Module & I/O
 2   5   1    128      81     18  F_WAIT      0       0  mshell <>>>term
 3   0   1    128     147     45  F_SLEEP     0       0  inetd <>>>nil
 4   0   1    128       0      0  ???         0       0  spf_rx
 5   0   1    128      20      0  ???         0       0  sysgo <>>>term
 6   0   1    128      12      2  F_SLEEP     0       0  ndpio <>>>nil
 7   9   1    128      92     23  F_GPRDSC    0     457  procs <>>>term
 8   0   1    128      84     14  I_SETSTAT   0       0  spfndpd <>>>nil
 9   2   1    128     131    184  F_WAIT     51      68  mshell <>>>term
```

- **procs –x zeigt detailliertere Informationen an**

- **procs –t zeigt Threadinformationen zu einem Prozess an**

Aging	Age of the process based on the initial priority and how long it has waited for processing
F$calls	Number of service request calls made
I$calls	Number of I/O requests made
Last	Last system call made
Read	Number of bytes read
Written	Number of bytes written

```
$ procs –t 7
Id  Pld Thd Grp.Usr  Prior  MemSiz Sig S  CPU Time  Age Module & I/O
 7   9   2     0.0    128   17.75k   0 a     22.23 0:01 ptest <>>>term
10   7   -     0.0    128       -    0 a     22.25 0:01
```

$ kill <TaskID> terminiert sofort die Task TaskID

Echtzeitbetriebssysteme

- **OS9: Interaktives Starten von Tasks I**

Die *mshell* startet nach Booten des Praktikumssystems auf der Console oder nach dem Login des Benutzers via Telnet als User Super, Passwort user und kreiert folgende *enviroment*- Variable

PORT	Der Name des Terminals	*z.B. /t1* ►nur via telnet
HOME	Ihr *home-directory* ►nur via telnet	
SHELL	Die erste Task die durch Ihr *login* gstartet wurde	
USER	login-Name ►nur via telnet	
PROMPT und _SHELLPARAMS		
PWD	Aktuelles Verzeichnis	

Diese können mittels folgender Kommandos manipuliert werden: ***setenv,unsetenv,printenv***

z.B. $ setenv PATH .:/h0/CMDS:/d0/CMDS

Einige eingebaute **shell-Kommandos:**

** <text> ►comment ; chd <path> ►Wechselt Data-Dir; chx <path> ►wechselt Execution-Dir ex <name> ►shell wird durch Prog. <name> ersetzt (gestartet); kill <procID> ►beendet Task procID logout ►abmelden; profile <path> ►Führe in der Datei path enthaltene shell-commands aus w ►Warte auf Beendigung **eines** Kindprozesses; wait ►Warte auf Beendigung aller Kindprozesse setpr <procID> <new-priority> ►Ändere Priorität der Task procID*

Echtzeitbetriebssysteme

• **OS9: Interaktives Starten von Tasks II**
Starten von Tasks (process) vom mshell-Prompt aus:

$ meinprogramm #40k ^130 </dd/TEST/myinput >/dd/TEST/myoutput >>/dd/TEST/myfehler

*Es wird **meinprogramm** als Kindprozess gestartet und danach auf Beendigung gewartet.*
*►Zuerst sucht die shell das Moduldirectory nach dem **Modul meinprogramm** ab, ob es bereits geladen ist, wenn ja wird ggf. nur ein neuer Datenbereich für die Task angelegt und der geladene Code erneut gestartet wenn der Benutzer entsprechend den **Modulpermission**rechten dazu berechtigt ist.*
*►Wenn es nicht geladen ist wird über das **Filesystem** zuerst der execution-path (chx <path>) und dann entlang der PATH-Enviromentvariablen gesucht. Wenn die **Datei meinprogramm** gefunden wurde **und** Benutzer diese Datei **(Filepermissions)** ausführen darf, wird der Programmcode geladen. Der im **Modulkopf** enthaltene **Modulname** wird ins Moduldirectory eingetragen und es wird überprüft ob die im Modulkopf enthaltene **Modulpermission** es dem Benutzer erlaubt dieses Programm auszuführen.*
►Zu allerletzt wird das Data-Dir durchsucht und falls meinprogramm gefunden wird, wird angenommen, dass meinprogramm ein Textfile ist welches shell Kommandos enthält. ►Start einer Kind-shell mit Übergabe der Datei und warten auf Beendigung.
*→ **der #** Modifier alloziert zusätzlichen Daten-Speicher zur Task, wenn weggelassen Größe aus Modulkopf*
*→ **der** ^ Modifier erlaubt es die Startpriorität zu verändern ►ohne Angabe hier 128*
< Input-, > Output-, >> Erroroutput-Umleitung von/in Datei ►default stdin,stdout,stderr ans Terminal... (>+ heißt anhängen an und >- überschreiben einer bereits existierende Datei)

Echtzeitbetriebssysteme

- **OS9: Interaktives Starten von Tasks III**

*( ) wird benutzt um Kommandos zusammenzufassen: $(del *.backup; list stuff >p)&*

Mehrere Kommandos in einer Zeile werden, wenn durch ; getrennt, nacheinander ausgeführt. !Nach Beendigung des letzten Kommandos erscheint der shell-Prompt wieder!!

Mehrere Kommandos in einer Zeile werden, wenn durch & getrennt, parallel ausgeführt
& als letztes Zeichen in der Zeile bedeutet: Ausführung des/der Kommandos im Hintergrund
►shell-prompt erscheint wieder, weitere Eingaben möglich! (& Operator geht vor ; Operator!)
!! Achtung die gestarteten Prozesse erben stdin,stdout und stderr vom Starter!!

Werden die **Kommandos durch ! getrennt** so werden alle Prozesse parallel gestartet. Der Output des ersten Kommandos wird umgeleitet aus Input des zweiten Kommandos .... ►***Filter durch pipes***

Wildcards-Expansion (teilweise!!)
* steht für ein bzw. mehrere beliebige oder kein Zeichen
? genau für ein beliebiges Zeichen

Echtzeitbetriebssysteme

- **OS9: Interaktives Starten von Tasks IV**

Besonders:Eingabe von Control-c ►Wird ein Prozess im Vordergrund gestartet **und** hat noch kein I/O auf das Terminal getätigt **dann und nur dann** kann man mittels ^c den Prozess nachträglich in den Hintergrund befördern und es erscheint der Kommandprompt wieder!!
!!^c wird als Signalnr.3 immer an die Task geschickt die zuletzt I/O am Terminal getätigt hat!!

Besonders: Eingabe von Control-e ►Wird ein Prozess im Vordergrund gestartet **und** hat noch kein I/O auf das Terminal getätigt **dann und nur dann** kann man mittels ^e den Prozess aborten und es erscheint der Kommandprompt wieder!!
!!^e wird als Signalnr.2 immer an die Task geschickt die zuletzt I/O am Terminal getätigt hat!!

Jede Task die gestartet wird, erbt die Group.UserID des Erzeugungerprozesses.
►Durch den login-Prozess wird ein User-Programm gestartet, dieses erhält die GroupUserID des Benutzers ►normalerweise die mshell
►Es gibt Programme die nur mit der SuperuserID (0,0) ausgeführt werden dürfen.