Betriebssysteme

M. Steffen

Wintersemester 2003/04

Abschnitt I

Prozeßmanagement

Literatur: In der Hauptsache Kapitel 5 aus [Com83]. Allgemeine Weisheiten zum Thema aus verschiedenen anderen Büchern.

Inhalt: Prozeßzustände · `suspend`/`resume` · Kreierung und Terminierung von Prozessen · Systemaufrufe · `kill` und `create` · Pseudocall

Einfrieren und Entfrosten

- transparentes Einfrieren/Entfrosten eines Prozesses
  - wie beim Kontextwechsel zwischen `CURRENT` u. `READY`: Retten aller Daten potentiell wichtig
  - auf Benutzerwunsch ⇒Systemaufruf (*system call*)
  - Einfrieren „bis auf expliziten Widerruf“ ⇒neuer Zustand SUSPENDED
- Abfragen von Fehlerbedingungen ⇒`SYSERR`
- Retten des Zustands/(potentieller) Kontextwechsel (resched)
- Ausschalten der Interrupts und wieder “Zurück”-schalten[1]

[1] wie `disable/enable/restore` implementiert ist, kommt später in `kernel.h`. Die Datei einthält auch die Definition von `OK/SYSERR`.

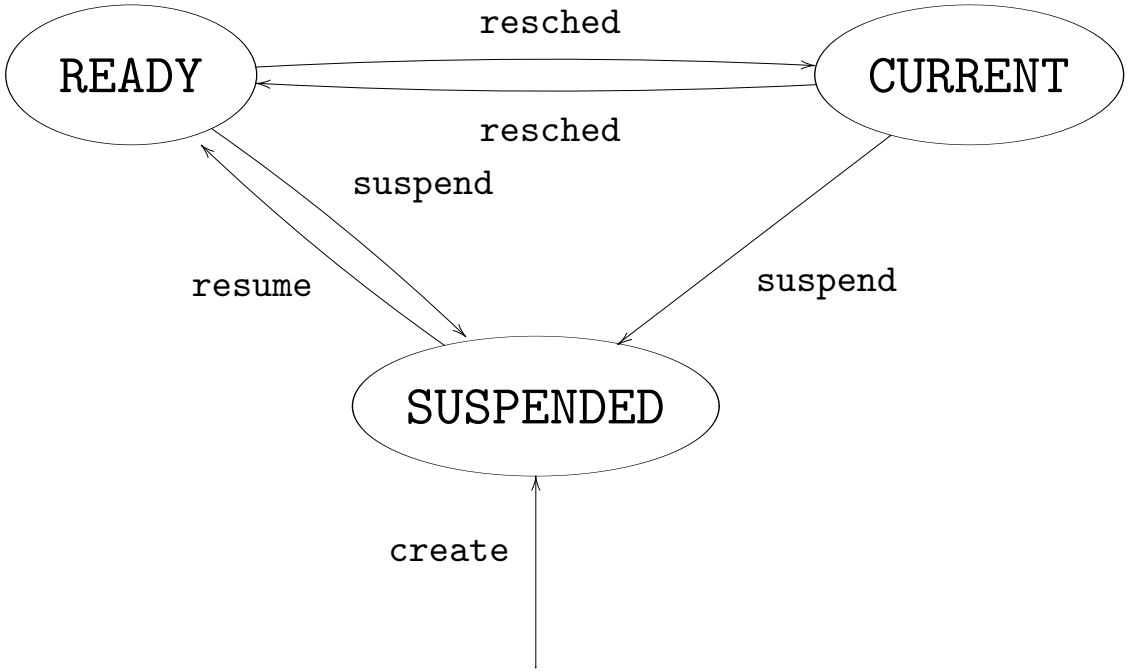
Prozeßzustand: `suspended`

- Prozeß: ein jeweils genau einem von möglichen Zuständen
- Insgesamt in Xinu: 6 Zustände
- Bisher (die beiden fundamentalen für Timesharing/Multiprogramming):
  - `CURRENT` Laufend
  - `READY` Lauffähig, aber gerade nicht dran
- Heute: Suspendiert

  `SUSPENDED`

  = temporäres, benuztergesteuertertes Gestopptsein ($\neq$ `READY`)
- Aber: keine Suspendedqueue

Prozeßzustände (bislang)



Systemaufrufe

- Systemaufruf[2]: vom Benutzer aufrufbar
- i.d.R. mit besonderen Privilegien ausgestattet[3]
- Aufgaben
  1. Schnittstelle zwischen Anwenderprogrammen und Systemkern, also die Dienste des Betriebssystems[4]
  2. Schutz des System
  3. Siehe Anhang 2.2 von [Com83]: system calls (angelehnt an Unix)

[2] *system call*, auch *monitor call*, *supervisor call*

[3] Stichwort: "dual mode" Operationen, sofern die HW das unterstützt

[4] Darum auch manchmal SPI genannt *system programmers interface* um es vom API zu trennen (*application programmers interface*)

Systemaufrufe: Klassifizierung

- Systemaufrufe: alles was der Benutzer vom System verlangt
- grobe Klassifizierung
  - Prozeßmanagement und Signale, Interprozeßkommunikation
  - File/management, Verzeichnis- und Filesystemmanagement
  - Schutzfunktionen
  - Zeitmanagement oder allg. Informationmanagement
  - I/O (z.B. Öffnen, Schließen, Terminalattribute, Geschwindigkeiten ...)
  - (Speichermanagement)

Entfrosten mit `resume`

- [Com83] Seite 56
- Abfangen von
  1. falscher PID
  2. Zustand nicht suspendiert
- im Wesentlichen: Verwendung von `ready`
- zeitweises Ausschalten der Interrupts (Systemaufruf)
- Rückgabewert: Priorität zum Zeitpunkt des Aufrufs ⇒Retten des Wertes

Code von `resume` (Auszug)

```
[....]
SYSCALL resume(pid)
        int     pid;
{
  disable(ps);
  if (isbadpid(pid) || (pptr = &proctab[pid])->pstate != PRSUSP) {
                restore(ps);
                return(SYSERR);
  }
  prio = pptr->pprio;
  ready(pid, RESCHYES);
  restore(ps);
  return(prio);
}
```

Einfrieren mittels `suspend`

- dualer Systemaufruf zu `resume`
- ähnliche (aber mehr) Fehlbedienungen müssen abgefangen werden
- Aus- und "Rück"-schalten der Interrupts
- Priorität: Nach dem wieder Aufwachen[5]
- *anders als beim Suspendieren*: Aufruf an sich selbst: Selbstsuspendierung
  - „Parameterübergabe" an `resched`: Setzen des eigenen Prozeßzustandes auf das gewünschte `SUSPENDED`

[5] Macht das einen Unterschied?

```
SYSCALL suspend(pid) [..]
{
  disable(ps);
  if (isbadpid(pid) || pid==NULLPROC ||
      ((pptr= &proctab[pid])->pstate!=PRCURR && pptr->pstate!=PRREADY)) {
    restore(ps);
    return(SYSERR);
  }
  if (pptr->pstate == PRREADY) {
    dequeue(pid);
    pptr->pstate = PRSUSP;
  } else {
    pptr->pstate = PRSUSP;
    resched();
  }
  prio = pptr->pprio;
  restore(ps);
  return(prio);
}
```

Terminierung durch Tötung

- Terminieren:
  - Prozeß verschwindet ganz aus dem System
  - Freigabe seiner Ressourcen
    1. Pid, Prozeßtabelle
    2. Stack
    3. Entfernen aus den Queues (Ready-Queue, Semaphore-Queue, Sleepqueue)
    4. Zusätzlich (nicht im Code von Comer) Schließen offener Devices/Connections/Sessions etc.

- Aufräumen = Zugriff auf gemeinsame Daten ⇒Gegenseitiger Ausschluß ⇒Interrupts ausschalten!

- Systemaufruf kill
  - Beachte: Selbstmord möglich

kill-Code (zweigeteilt)

```
SYSCALL kill(pid) [....]
{
   disable(ps);
   if (isbadpid(pid) || (pptr= &proctab[pid])->pstate==PRFREE) {
      restore(ps);
      return(SYSERR);
   }
   if (--numproc == 0)                          /* letzer Prozess?          */
      xdone();                                  /* message: "und tschuess"  */
   freestk(pptr->pbase, pptr->pstklen);         /* Memory management S. 105 */
    [..........]                                /* Reaktion per Zustand     */
     }
 restore(ps);
 return(OK);
}
```

```
SYSCALL kill(pid) [...]
{
  disable(ps);  [....]                          /* Siehe vorige Folie */
    switch (pptr->pstate) {
    case PRCURR:          pptr->pstate = PRFREE;      /* suicide */
      resched();                                      /* wichtig */

    case PRWAIT:          semaph[pptr->psem].semcnt++;
    case PRSLEEP:
    case PRREADY:         dequeue(pid);             /* aus Q entfernen */

    default:      pptr->pstate = PRFREE;
    }
    restore(ps);
    return(OK);
}
```

Kerneldeklarationen

- zentrale, OS-weite Systemkonstanten und Routinen
- Beispiele: Abkürzungen für
  - Register
  - Typen
  - Standardwerte ...
- Inline-Prozeduren (wg. Effizienz)
- `asm`: direkte Assemblerbefehle
  - `mfps`/`mtps` *Move from/to processor status*: Kopiere `PS` in das angegebene Register, kopiere ein Wort in das `PS`
  - Beachte: *Prozessorstatus*: Oktal 0340 bedeutet: Interrupts aus

Funktionen in `kernel.h`

```
/* kernel.h – disable, enable, halt, restore, isodd, min */
   .......

#define isodd(x)          (01&(int)(x))
#define min(a,b)          ((a) < (b) ? (a) : (b))
#define disable(ps)       asm("mfps ~ps");asm("mtps $0340")
#define restore(ps)       asm("mtps ~ps") /* restore interrupt status     */
#define enable()          asm("mtps $000")/* enable interrupts            */
#define pause()           asm("wait")     /* machine "wait for interr."   */
#define halt()            asm("halt")     /* machine halt instruction     */
```

Prozeßkreierung

- neuer Prozeß mit Systemaufruf create
- Zuweisung aller benötigten Ressourcen, Daten = Parameter des Systemaufrufs
  - (neue) Pid, Eintrag in Prozeßtabelle
  - Stackbereich etc (Siehe Kapitel 8. ca. S. 105)
  - Priorität
  - Name
  - Argumente
- der erzeugende Prozeß soll i.d.R. weiterlaufen ⇒der neue kann nicht sofort `CURRENT` werden
- ⇒ möglichst transparent in das System „einfädeln“ ⇒

als sei er nur gestoppt

Prozeßkreierung (2)

- nicht alle Prozesse werden vom Benutzer einzeln getötet ⇒Problem: wie bereitet man den

  natürlichen Tod

  eines Prozesses vor?

- Antwort: in dem der Prozeß beim letzten Return nicht “irgendwohin” returned, sondern anständig weggeräumt wird.

Pseudocall

⇒ Beachtung der C-Konventionen, Simulation eines Prozeduraufrufs

- Prozeß bekommt Stack = Teil des globalen Stackbereichs ⇒: sein initialer muß bereits den Konventionen gehorchen

Pseudocall

mit passend gefälschten

1. Rücksprungadresse
2. Argumente

sodaß der Prozeß am Ende in den vorbereiteten (Frei-)Tod returniert
⇒Returnadresse auf `userret`

Prozedur `userret`

```
/* userret.c - userret */

#include <conf.h>
#include <kernel.h>

/*------------------------------------------------------------------------
 * userret  --  entered when a process exits by return
 *------------------------------------------------------------------------
 */
userret()
{
        kill( getpid() );
}
```

Code von `create`

- Code: aufgeteilt in 3 Phasen
  1. Stackbereich reklamieren (vgl. Kapitel 8: Memory Management)
  2. Eintragen der Daten in die Prozeßtabelle, und zwar
     - gemäß der Parameter oder
     - Defaultwerte (definiert meist in `kernel.h`): insbesondere: Zustand = `SUSPENDED`
     - Vorbereitung des Pseudocalls
  3. Rückgabewert: Pid

```
SYSCALL create(procaddr,ssize,priority,name,nargs,args)
        int             *procaddr;      /* procedure address                */
        int             ssize;          /* stack size in words              */
        int             priority;       /* process priority > 0             */
        char            *name;          /* name (for debugging)             */
        int             nargs;          /* number of args that follow       */
        int             args;           /* arguments (treated like an       */
                                        /* array in the code)               */
{
    int     pid;                        /* stores new process id            */
    struct          pentry  *pptr;      /* pointer to proc. table entry     */
    int     i;
    int     *a;                         /* points to list of args           */
    int     *saddr;                     /* stack address                    */
    char    ps;                         /* saved processor status           */
    int     INITRET();
    [......]
}
```

```
SYSCALL create(procaddr,ssize,priority,name,nargs,args)[..] { [...]
 ssize = roundew(ssize);
 if ( ssize < MINSTK || ((saddr=getstk(ssize)) == SYSERR ) ||
                (pid=newpid()) == SYSERR || isodd(procaddr) ||
                priority < 1 ) {
                restore(ps);
                return(SYSERR);
 }
 numproc++;
 pptr = &proctab[pid];
 pptr->pstate = PRSUSP;          /* Wir starten SUSPENDED!! */
 for (i=0 ; i<PNMLEN && (pptr->pname[i]=name[i])!=0 ; i++)
   ;
 pptr->pprio = priority;         /* Sonstige Initialisierung */
 pptr->pbase = (short)saddr;     /* gemaess den Parametern   */
 pptr->pstklen = ssize;
 pptr->psem = 0;                 /* Semaphore: nix           */
 pptr->phasmsg = FALSE;          /* Messages:  keine         */
 pptr->plimit = (short)((unsigned)saddr - ssize + sizeof(int));
 pptr->pargs = nargs;
 .......
```

```
SYSCALL create(procaddr,ssize,priority,name,nargs,args) [...]
{

  for (i=0; i<PNREGS; i++)                    /* Register initialisieren      */
    pptr->pregs[i]=INITREG;

  pptr->pregs[PC] = pptr->paddr = (short)procaddr;  /* hier geht's los */
  pptr->pregs[PS] = INITPS;                         /* = 0             */

  a = (&args) + (nargs-1);                  /* point to last argument       */
  for ( ; nargs > 0 ; nargs--)              /* machine dependent; copy args */
    *saddr-- = *a--;                        /* onto created process' stack  */
  *saddr = (int)INITRET;                    /* push on return address       */
                                            /* INITRET = Selbstmordadresse  */
  pptr->pregs[SP] = (int)saddr;             /* Stackpointer vorbereiten     */
  restore(ps);                              /* und fertig                   */
  return(pid);                              /* R"uckgabewert: neue Id       */
}
```

Abschnitt II

Semaphoren

Literatur: Kapitel 6 aus [Com83]. Zusätzlich noch Abschnitt 2.2 aus [Tan87].

Inhalt: Semaphore · Prozeßkoordinierung · zugehörige Xinu-Systemaufrufe

Prozeßkoordinierung

Prozeßkoordinierung ist notwendig

- um Aktionen von Prozessen zu synchronisieren
- und um den Zugriff auf gemeinsame Resourcen zu regeln.

In Kapitel 1: Producer-Consumer-Problem, Mutual Exclusion

Die einfachste Möglichkeit zur Prozeßkoordinierung bieten Semaphore

Semaphore

- Jede Semaphore s ist im Prinzip ein integer-Wert.
- System Call `wait(s)` reduziert s um 1.
- System Call `signal(s)` erh"oht s um 1.
- Wird s beim Aufruf von `wait(s)` negativ wird der Prozeß angehalten.
- Beim Aufruf von `signal(s)` wird ein Prozeß der auf s wartet aufgeweckt.

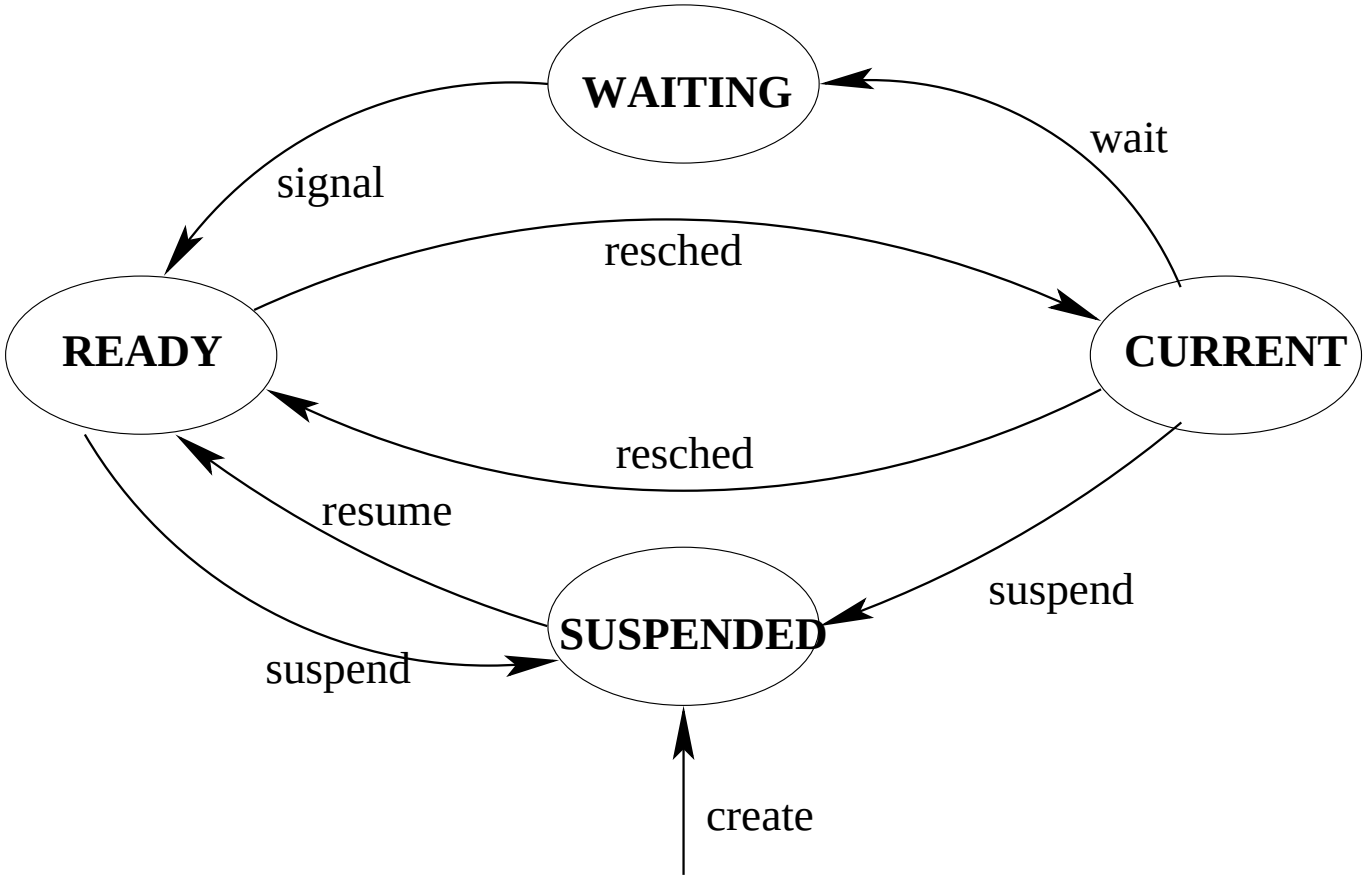
Low-Level Koordinierung

- `ready(s)` und `resume(s)` benutzen das Ausschalten der Interrupts, um sich zu koordinieren.
- Ausschalten der Interrupts ist aber i.A. nicht adequat:
  - nur ein Prozeß ist aktiv und dessen Möglichkeiten sind eingeschränkt.

Implementierung

- Xinu Implementierung vermeidet busy waiting durch Einführung eines neuen Prozeßzustandes **PRWAIT**.

- Zu jeder Semaphore gehört eine eigene Warteschlange.



Semaphoreninformationen

```
#ifndef NSEM
#define NSEM              35        /* number of semaphores, if not defined */
#endif
#define SFREE    '\01'              /* this semaphore is free               */
#define SUSED    '\02'              /* this semaphore is used               */
struct  sentry  {                   /* semaphore table entry                */
        char    sstate;             /* the state SFREE or SUSED             */
        short   semcnt;             /* count for this semaphore             */
        short   sqhead;             /* q index of head of list              */
        short   sqtail;             /* q index of tail of list              */
};
extern  struct  sentry  semaph[];
extern  int     nextsem;

#define isbadsem(s)     (s<0 || s>=NSEM)
```

Semaphores are identified by their index in the global semaphore table, **semaph**

System Call — wait

```
SYSCALL wait(sem)
    ...             {    ....
        register struct sentry  *sptr;
        register struct pentry  *pptr;

        disable(ps);
        if (isbadsem(sem) || (sptr=&semaph[sem])->sstate==SFREE) {
                restore(ps);
                return(SYSERR);
        }
        if (--(sptr->semcnt) < 0) {
                (pptr = &proctab[currpid])->pstate = PRWAIT;
                pptr->psem = sem;
                enqueue(currpid,sptr->sqtail);
                resched();
        }
        restore(ps);
        return(OK);
}
```

System Call — signal

```
SYSCALL signal(sem)
register int     sem;{
        register struct sentry  *sptr;
        char    ps;

        disable(ps);
        if (isbadsem(sem) || (sptr=&semaph[sem])->sstate==SFREE) {
                restore(ps);
                return(SYSERR);
        }
        if ((sptr->semcnt++) < 0)
                ready(getfirst(sptr->sqhead), RESCHYES);
        restore(ps);
        return(OK);   }
```

A nonnegative semaphore count means that the queue is empty; a semaphore count of negative n means that the queue contains n waiting processes.

Generierung von Semaphoren

```
SYSCALL screate(count)
        int     count;                  /* initial count (>=0)          */
{
        char    ps;
        int     sem;

        disable(ps);
        if ( count<0 || (sem=newsem())==SYSERR ) {
                restore(ps);
                return(SYSERR);
        }
        semaph[sem].semcnt = count;
        /* sqhead and sqtail were initialized at system startup */
        restore(ps);
        return(sem);
}
```

Neue Semaphor-Warteschlange

```
LOCAL   newsem()
{
        int     sem;
        int     i;

        for (i=0; i<NSEM; i++) {
                sem=nextsem--;
                if (nextsem < 0)
                        nextsem = NSEM-1;
                if (semaph[sem].sstate==SFREE) {
                        semaph[sem].sstate = SUSED;
                        return(sem);
                }
        }
        return(SYSERR);
}
```

Löschen von Semaphoren

```
SYSCALL sdelete(sem)
   ....     {   .....
        struct   sentry   *sptr;              /* address of sem to free        */

        disable(ps);
        if (isbadsem(sem) || semaph[sem].sstate==SFREE) {
                restore(ps);
                return(SYSERR);
        }
        sptr = &semaph[sem];
        sptr->sstate = SFREE;
        if (nonempty(sptr->sqhead)) {     /* free waiting processes        */
                while( (pid=getfirst(sptr->sqhead)) != EMPTY)
                        ready(pid,RESCHNO);
                resched();
        }
        restore(ps);
        return(OK);
}
```

Literatur

[Com83] Douglas Comer. *Operating System Design, The Xinu Approach*. Prentice Hall, 1983.

[Tan87] Andrew S. Tanenbaum. *Operating Systems: Design and Implementation*. Prentice Hall, 1987.