Betriebssysteme

W.-P. de Roever K. Baukus M. Steffen

Wintersemester 2000/01

Lektion 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 Einschalten[1]

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

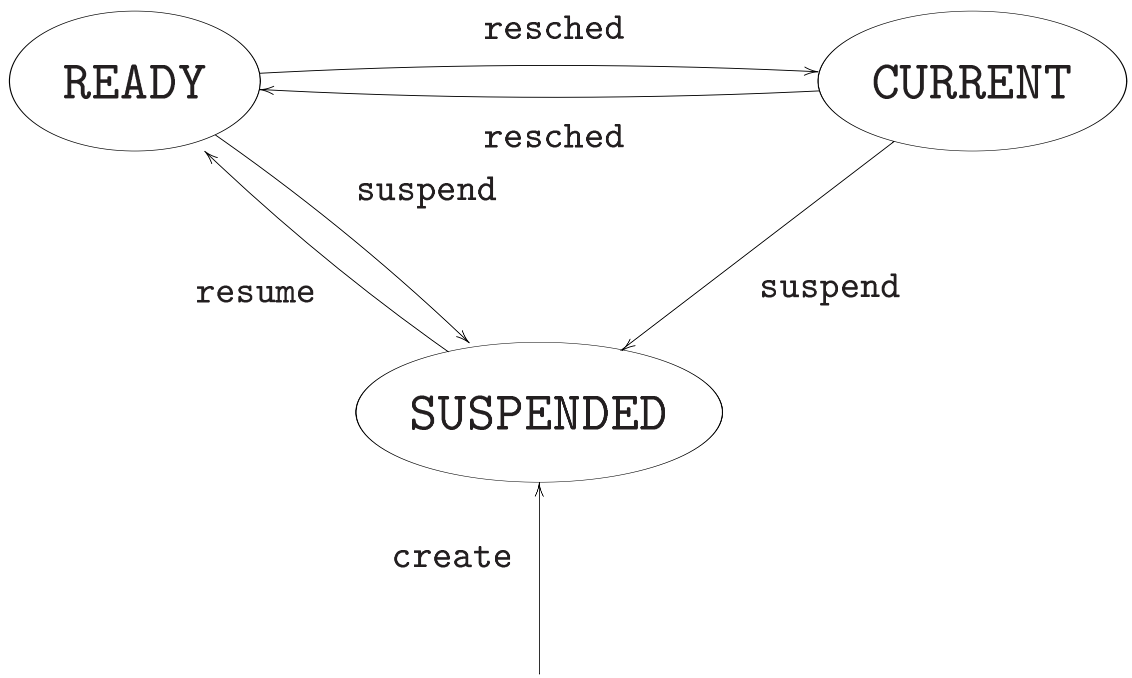
Prozeßzustand: `suspended`

- 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]
- Siehe Anhang 2.2 von [Com83]: system calls (angelehnt an Unix)
- Aufgaben
  1. Schnittstelle zwischen Anwenderprogrammen und Systemkern, also die Dienste des Betriebssystems[4]
  2. Schutz des System

[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
- Rückgabewert: Priorität zum Zeitpunkt des Aufrufs $\Rightarrow$Retten des Wertes
- Abfangen von
  1. falscher PID
  2. Zustand nicht suspendiert

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 Anschalten der Interrupts
- Priorität: Nach dem wieder Aufwachen[5]
- *Neu*: 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`

```c
/* 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        */
}
```

Lektion 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.

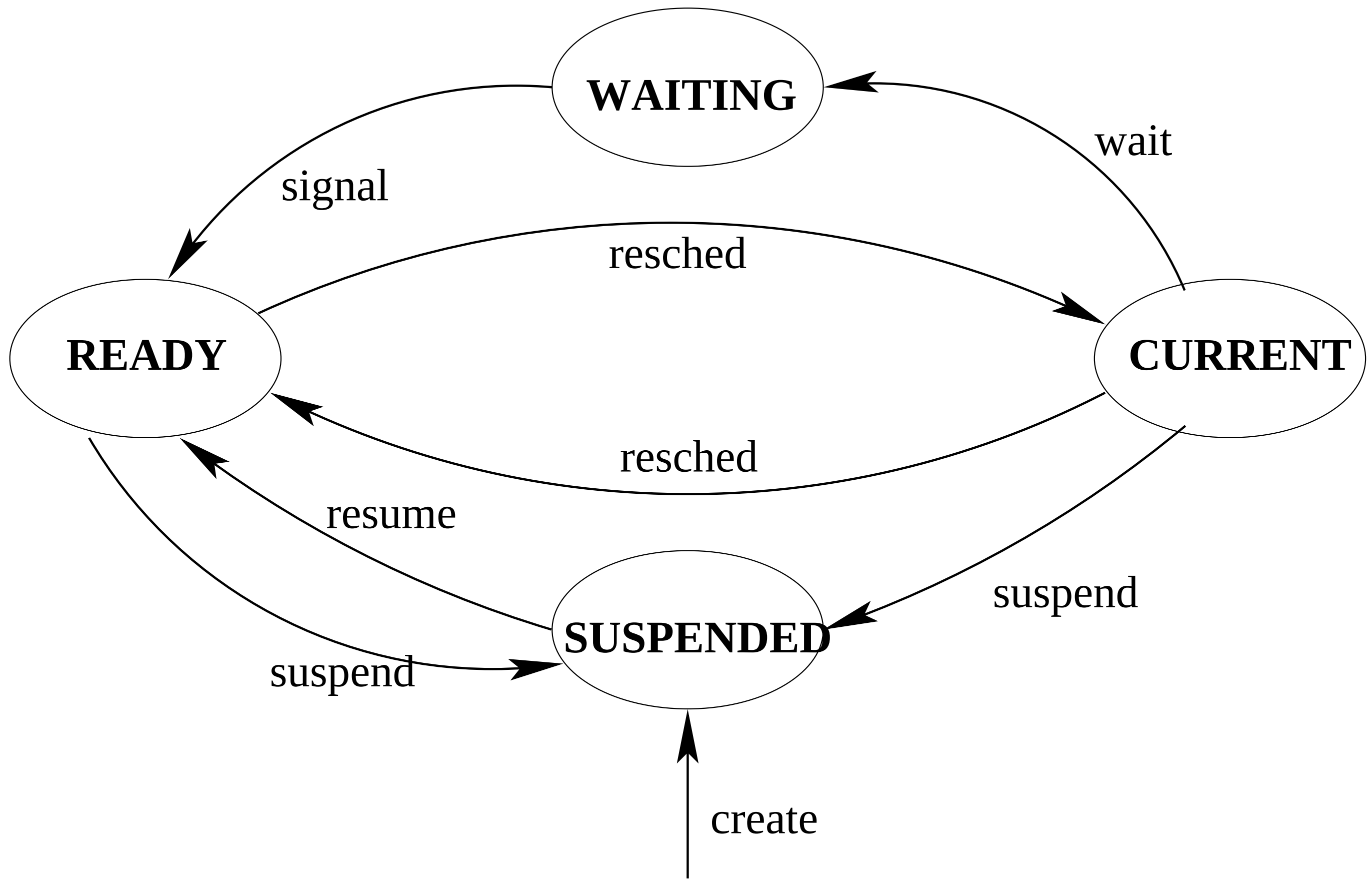
Low-Level Koordinierung

- `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

```
/* sem.h - isbadsem */

#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);
}
```

Lektion III

Message passing

Literatur: Kapitel 7 aus [Com83]. Zusätzlich noch Abschnitt 2.2 aus [Tan87].

Inhalt: Message passing · Prozeß-Prozeß-Messagepassing · zugehörige Xinu-Systemaufrufe

Kommunikation in Betriebssystemen

- Interprozeßkommunikation: wichtiges Mittel in Betriebssystemen
- Unix-Beispiel: shell-pipeline: IPC zwischen Benutzerprozessen
- Verschiedene Mechanismen
  - Semaphoren zur Koordination/Synchronisation ([Com83], Kapitel 6)
  - Monitore [Hoa74] [Han75]:
    - Mechanismus für gegenseitigen Ausschluß
    - high-level, d.h. Programmiersprachkonstrukte, eine Art Abstrakter Datentyp oder Objekt
    - kein Systemaufruf
  - Datenübertragung mittels gemeinsamen Variablen: *shared variables*
  - message passing
    - nicht nur Synchronisation, sondern auch Datenaustausch

Message passing

- Form der Interprozeßkommunikation/Prozeßsynchronisation/-koordination
- alternativ: shared variables
- anders als Synchronisation durch Semaphore: kann unsynchronisiert sein
- Implementiert durch Systemaufrufe:[6] `send` und `receive`
- Verschiedene Varianten möglich
  - Nachrichten von Prozeß zu Prozeß (direkt)
  - über eine getrennte Datenstruktur: Mailboxing (indirekt) (Beispiel: Unix-Pipelines)[7]

[6] Wie Semaphore

[7] einziger Unterschied: Message-Grenzen werden nicht bewahrt. Man kann weiter unterscheiden zwischen Mailboxen, die einem Prozeß zugehörig sind oder aber unabähngig sind.

Message passing (cont'd)

- Senden und Empfangen können blockierend oder nicht-blockierend sein
- Rendez-vouz: Senden *und* Empfangen sind blockierend.
- Kapazität der Verbindungen (Pufferlänge); was passiert, wenn der Puffer voll ist?
- Nachrichten fester oder variabler Länge
- ein oder mehrere Prozesse als Empfänger möglich?
- spezifizierter Sende/Empfangsprozeß
- ...

Message passing in Xinu

- Zwei Formen des Nachrichten Austauschs
  1. direkt von Prozeß zu Prozeß
  2. über Rendezvous-Punkte (siehe Kapitel 14 in [Com83])
- drei Systemaufrufe
  1. `send`: asynchron
  2. `receive` (synchron)
  3. `recvclr` (asynchron)
- Empfangspuffer der Größe 1 (= ein Wort),
  - nur die erste wird empfangen
  - alle anderen gehen verloren, da Sender nicht blockiert!
- neuer Prozeßzustand: Empfangend (PRRECV)

- Übergabeort: reservierter Platz im Empfangangsprozeß (also in der Prozeßtabelle)
  - im Sender geht nicht, denn der kann sterben
  - direkt im Speicherbereich des Empfängers: zu gefährlich

```
/*  proc.h      vgl. S. 55   */
...

struct   pentry {
         char      pstate;            /* Zustand  */
         ...
         short     pmsg;              /* 1 message sent to this process */
         short     phasmsg;           /* nonzero => msg is valid        */
         ...
};
...
```

Senden

- Interrupts ausgeschaltet
- benutzt den Prozeßidentifikator, um auf den entsprechenden Prozeßtabelleneintrag zugreifen zu können
- Fehler falls
  - Prozeßid nicht paßt (ungültig oder nicht vergeben)
  - Empfangspuffer bereits voll
- Ansonsten
  1. Übergabe der Message
  2. Setzen des Flags (`phasmsg`)[8]
  3. falls Empfänger auf Message wartet ⇒ Reschedulen mittels `ready`[9]

[8] Eine Art *biff*: /bif/ vt. To notify someone of incoming mail. From the BSD utility '`biff(1)`', which was in turn named after a friendly golden Labrador who used to chase frisbees in the halls at UCB while 4.2BSD was in development (it had a well-known habit of barking whenever the mailman came). No relation to BIFF. (Lexicon of Hacker Jargon)

[9] [Com83], Seite 62. `ready` setzt den Zustand des Prozesses auf `PREADY` und ruft, falls das zweite Argument auf `true` gesetzt ist wie in diesem Fall, `reschedule` auf. Dabei ist `RESCHEDYES` in `hernel.h` auf Seite 72 definiert. Das Aktivieren und Deaktivieren der Interrupts wird ebenfalls dort gemacht.

Senden — send

```
SYSCALL send(pid, msg)
    ....    {   ....
        struct  pentry  *pptr;          /* receiver's proc. table addr. */

        disable(ps);
        if (isbadpid(pid) || ( (pptr= &proctab[pid])->pstate == PRFREE)
            || pptr->phasmsg) {
                restore(ps);
                return(SYSERR);
        }
        pptr->pmsg = msg;               /* deposit message              */
        pptr->phasmsg = TRUE;
        if (pptr->pstate == PRRECV)     /* if receiver waits, start it  */
                ready(pid, RESCHYES);
        restore(ps);
        return(OK);
}
```

Asynchrones Empfangen

- ähnlich dem synchronen Empfangen
- falls Nachricht vorhanden: Rückgabe
- falls keine Nachricht vorhanden ist: sofort Rückkehr mit `OK`

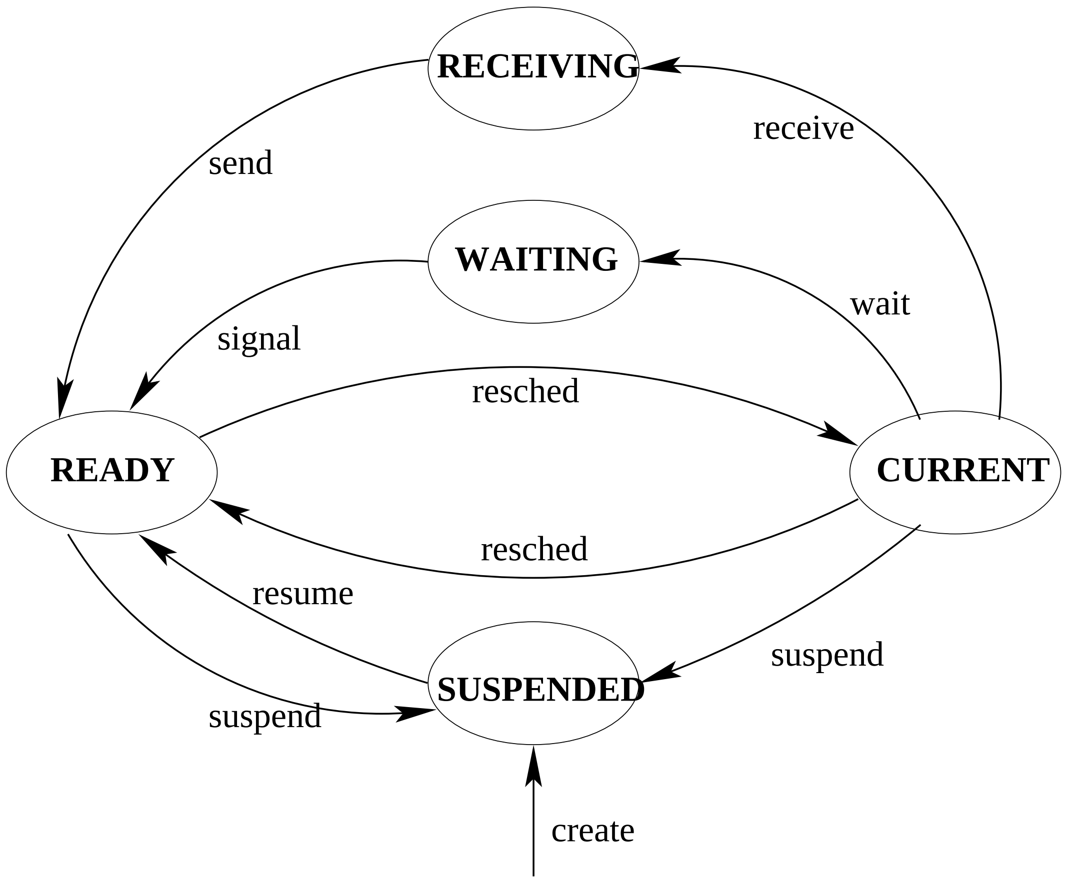
Asynchrones Empfangen — `recvclr`

```
SYSCALL recvclr()
{
        char    ps;
        int     msg;

        disable(ps);
        if ( proctab[currpid].phasmsg ) {        /* existing message?    */
                proctab[currpid].phasmsg = FALSE;
                msg = proctab[currpid].pmsg;
        } else
                msg = OK;
        restore(ps);
        return(msg);
}
```

Synchrones Empfangen

- ähnlich dem asynchronen Empfangen,
- Unterschied: falls keine Nachricht vorhanden: Übergang in den neuen Empfangszustand



Synchrones Empfangen — `receive`

```
SYSCALL receive()
{
        struct  pentry  *pptr;
        int     msg;
        char    ps;

        disable(ps);
        pptr = &proctab[currpid];
        if ( !pptr->phasmsg ) {         /* if no message, wait for one */
                pptr->pstate = PRRECV;
                resched();
        }
        msg = pptr->pmsg;               /* retrieve message            */
        pptr->phasmsg = FALSE;
        restore(ps);
        return(msg);
}
```

Lektion IV

Memory Management

Literatur: Kapitel 8 aus [Com83].

Inhalt: Speicher · Speicheralloziierung · Speicherfreigabe

Memory Management

- Hauptspeicher ist essentiell zur Programmausführung.
- Das OS vergibt freien Speicher und sammelt ihn wieder ein.
- Wenn die Nachfrage den tatsächlichen Speicher überschreitet, führt das OS *multiplexing* durch:
  - *swapping* lagert ganze Prozesse auf sekundären Speicher aus,
  - *paging* hält nur die gerade benutzten Speicherbereiche eines Prozesses (*pages*) im Hauptspeicher.
  - OS mit paging geben den Prozessen häufig einen eigenen virtuellen Adreßbereich.

Effiziente Speicherverwaltung setzt Hardwareunterstützung voraus.

Memory Management in Xinu

- Die 11/2 Hardware unterstützt weder paging noch werden die Adreßräume der Prozesse voreinander geschützt.
- Das Xinu System und alle Prozesse benutzten den gleichen Adreßraum.
- Der Xinu Programmcode und die globalen Variablen bleiben im laufenden System unberührt.
- Laufende Programme erfordern dynamisch Stack für Prozeduraufrufe nd deren lokale Variablen und Heap für dynamisch wachsende Datenstrukturen.

  Speicher wird hier zunächst als reichlich vorhanden vorausgesetzt und bei Bedarf vergeben. Ist kein Speicher mehr vorhanden muß der Prozeß selbst entscheiden, wie zu reagieren ist. (Dies setzt voraus, das die Prozesse nicht sofort allen verfügbaren Speicher anfordern.)

Low-Level Memory Management in Xinu

- `getstk` and `freestk` sind für die Ver- und Freigabe von Prozeßstackbereichen verantwortlich.
- `getmem` and `freemem` verwalten Blöcke von freiem Speicher.
- Da nur `create` and `kill` `getstk` and `freestk` aufrufen, wird der belegte Stack bei Prozeßterminierung wieder freigegeben.
- Heap muß hingegen vom Prozeß vor seiner Terminierung wieder freigegeben werden.
- Auch dann kann der freie Speicher ausgeschöpft oder stark fragmentiert werden, so daß kein neuer Speicher mehr vergeben werden kann.

Memory Information

```
#define roundew(x)      (int *)( (3 + (int)(x)) & (~3) )
#define truncew(x)      (int *)( ((int)(x)) & (~3) )
#define freestk(p,len)  freemem((unsigned)(p)                 \
                          - (unsigned)(roundew(len))          \
                          + (unsigned)sizeof(int),            \
                          roundew(len) )

struct  mblock  {
        struct  mblock  *mnext;
        unsigned  mlen;
        };
extern  struct  mblock  memlist;   /* head of free memory list     */
extern  int     *maxaddr;          /* max memory address           */
extern  int     etext;             /* address beyond text segment  */
extern  int     edata;             /* address beyond data segment  */
extern  int     end;               /* address beyond loaded memory */
```

Blocks on the free list are ordered by increasing address.

Heap Speicher anfordern — getmem

```
SYSCALL *getmem( nbytes )
        unsigned nbytes;
{
        char      ps;
        struct    mblock   *p, *q, *leftover;

        disable(ps);
        if  (nbytes==0 ||  memlist.mnext==NULL) {
                restore(ps);
                return( (int *)SYSERR);
        }
        nbytes  = (unsigned)  roundew(nbytes);
        for  (q= &memlist, p=memlist.mnext  ;  p!=NULL  ;  q=p, p=p->mnext)
                if  (  p->mlen  ==  nbytes) {
                        q->mnext  =  p->mnext;
                        restore(ps);
                        return( (int  *)p  ); }
```

```
                else if ( p->mlen > nbytes ) {
                        leftover = (struct mblock *)( (unsigned)p + nbytes );
                        q->mnext = leftover;
                        leftover->mnext = p->mnext;
                        leftover->mlen = p->mlen - nbytes;
                        restore(ps);
                        return( (int *)p );
                }
        restore(ps);
        return( (int *)SYSERR );
}
```

Stack Speicher anfordern — getstk

```
SYSCALL *getstk(nbytes)
        unsigned int nbytes;
{
        char      ps;
        struct    mblock   *p, *q;  /* q follows p along memlist        */
        struct    mblock   *fits, *fitsq;
        unsigned  len;

        disable(ps);
        if (nbytes == 0) {
                restore(ps);
                return( (int *)SYSERR );
        }
        nbytes = (unsigned)roundew(nbytes);
        fits = NULL;
        q = &memlist;
```

```
        for (p = q->mnext ; p != NULL ; q = p,p = p->mnext)
                if ( p->mlen >= nbytes) {
                        fitsq = q;
                        fits = p;
                }
        if (fits == NULL) {
                restore(ps);
                return( (int *)SYSERR );
        }
        if (nbytes == (len = fits->mlen) ) {
                fitsq->mnext = fits->mnext;
        } else {
                fits->mlen -= nbytes;
        }
        fits = ((unsigned)fits) + len - sizeof(int);
        restore(ps);
        return( (int *)fits );
}
```

Speicher freigeben — `freemem`

```
SYSCALL freemem( block , size )
        struct   mblock   *block;
        unsigned size;
{
        char     ps;
        struct   mblock   *p, *q;
        unsigned top;

        if ( size==0 || (unsigned) block >(unsigned) maxaddr
                || (( unsigned) block )<((unsigned)&end))
                return (SYSERR);
        size = (unsigned) roundew( size );
        disable ( ps );
        for ( p=memlist.mnext, q= &memlist ; ( char *) p!=NULL && p<block ;
                q=p, p=p->mnext )
                ;
        if (( top=q->mlen+(unsigned) q)>(unsigned) block && q!= &memlist ||
                (char *) p!=NULL && ( size +(unsigned) block ) > (unsigned) p) {
                restore ( ps );
                return (SYSERR); }
```

```
        if ( q!= &memlist && top == (unsigned)block )
                q->mlen += size;
        else {
                block->mlen = size;
                block->mnext = p;
                q->mnext = block;
                q = block;
        }
        if ( (unsigned)( q->mlen + (unsigned)q ) == (unsigned)p) {
                q->mlen += p->mlen;
                q->mnext = p->mnext;
        }
        restore(ps);
        return(OK);
}
```

Literatur

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

[Han75] P. Brinch Hansen. The programming language Concurrent Pasal. *IEEE Transactions on Software Engineering*, 1:199–207, June 1975.

[Hoa74] Charles A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.

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