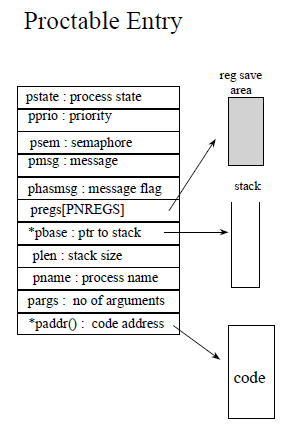
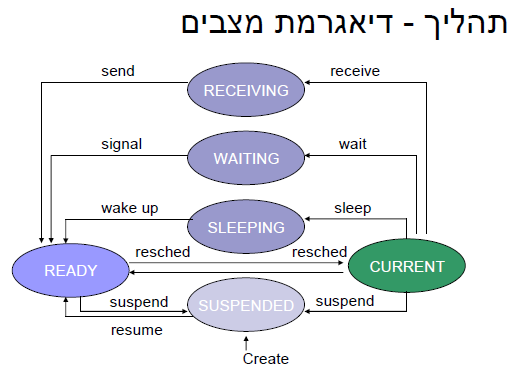
**PROC.H**

****

****

**struct pentry**

**{**

**char pstate;**

**/\* process state:**

**#define PRCURR '\01'--process is currently running**

**#define PRFREE '\02'--process slot is free**

**#define PRREADY '\03'--process is on ready queue**

**#define PRRECV '\04'--process waiting for message**

**#define PRSLEEP '\05'--process is sleeping**

**#define PRSUSP '\06'--process is suspended**

**#define PRWAIT '\07'--process is on semaphore queue**

**\*/**

**int pprio; /\* process priority \*/**

**int psem; /\* semaphore if process waiting \*/**

**int pmsg; /\* message sent to this process \*/**

**int phasmsg; /\* nonzero iff pmsg is valid(1)\*/**

**char \*pregs; /\* saved environment \*/**

**char \*pbase; /\* base of run time stack \*/**

**word plen; /\* stack length in bytes \*/**

**char pname[PNMLEN+1]; /\* process name \*/**

**int pargs; /\* initial number of arguments \*/**

**int (\*paddr)(); /\* initial code address \*/**

**};**

**extern struct pentry proctab[];**

**extern int numproc; /\* currently active processes \*/**

**#ifndef NPROC /\* set the number of processes \*/**

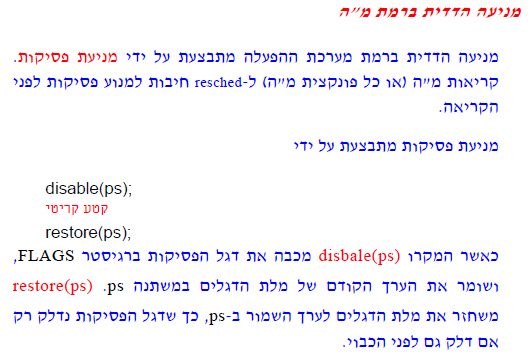
**#define NPROC 30 /\* allowed if not already done** \*/

**initialize.c**

**struct pentry proctab[NPROC]; /\* process table \*/**

**struct sentry semaph[NSEM]; /\* semaphore table \*/**

**struct qent q[NQENT]; /\* q table (see queue.c) \*/**



**מאחר ש-disable שומר את המצב הנוכחי של הדגלים ו- restore מחזיר את אותו המצב, ניתן להשתמש בהם בצורה מקוננת, תוך כדי שמירה על הכללים הבאים:**

**1. לא יתקיימו שתי קריאות ל-disable מקוננות עם אותו פרמטר ל-disable. (כלומר, לא לדרוס את ערך הדגלים השמור כבר במשתנה מסוים).**

**2. הפונקציה הקוראת ל-disable, היא הקוראת ל- restore בסיום הקטע הקריטי.**

**לכן, ניתן לראות שקריאות מ"ה ב- XINU "עטופות" בצירוף: disable(ps)/restore(ps) ו- ps הוא משתנה מקומי מסוג int.**

המקרו enable():

**enable מדליק את דגל הפסיקות (קורא ל-sti).**

**מקרו זה מאפשר להדליק את דגל הפסיקות ללא קשר לערכו האחרון בעת כיבויו, אולם אנו** לא נשתמש **בו ב-XINU.**

למשתמש אסור לקרוא ל- disable/restore/enable. במערכות הפעלה מתקדמות יש הגנה כדי שמשתמש **לא יוכל** לקרוא לפונקציות כאלה.

---------------------------------------------------------------------

**kernel.h**

**#define disable(x) (x)=sys\_disabl() /\* save interrupt status \*/**

**#define restore(x) sys\_restor(x) /\* restore interrupt status \*/**

**#define QUANTUM 1 /\* clock ticks until preemption \*/**

**---------------------------------------------------------**

**eidi.asm**

**---------------------------------------------------------**

**; \_sys\_disabl -- return interrupt status and disable interrupts**

**; int sys\_disabl()**

**\_sys\_disabl proc near**

**pushf ; put flag word on the stack**

**cli ; disable interrupts!**

**pop ax ; deposit flag word in return register**

**ret**

**\_sys\_disabl endp**

**---------------------------------------------------------------------**

**; \_sys\_restor -- restore interrupt status**

**; void sys\_restor(ps)**

**; int ps;**

**\_sys\_restor proc near**

**push bp**

**mov bp,sp ; C calling convenion**

**push [bp+4]**

**popf ; restore flag word**

**pop bp**

**ret**

**\_sys\_restor endp**

---------------------------------------------------------------------

**create.c**

**/\* newpid -- obtain a new (free) process id \*/**

**LOCAL newpid()**

**{**

**int pid; /\* process id to return \*/**

**int i;**

**for (i=1 ; i<NPROC ; i++)**

**{**

**/\* check all NPROC slots \*/**

**if (proctab[i].pstate == PRFREE)**

**return i;**

**}**

**return(SYSERR);**

**}**

**---------------------------------------------------------**

**/\*create--create a process to start running a procedure\*/**

**SYSCALL create(procaddr,ssize,priority,namep,nargs,args)**

**int (\*procaddr)(); /\* procedure address \*/**

**word ssize; /\* stack size in words \*/**

**short priority; /\* process priority > 0 \*/**

**char \*namep; /\* name (for debugging) \*/**

**int nargs; /\* number of args that follow \*/**

**int args; /\* arguments (treated like an array) \*/**

**{**

**int pid; /\* stores new process id \*/**

**struct pentry \*pptr;/\*pointer to proc table entry \*/**

**int i; /\* loop variable \*/**

**int \*a; /\* points to list of args \*/**

**char \*saddr; /\* start of stack address \*/**

**int \*sp; /\* stack pointer \*/**

**int ps; /\* saved processor status \*/**

**disable(ps);**

**ssize = roundew(ssize); /\* #define** **[roundew](http://www.ugrad.cs.ubc.ca/spider/xinu/xinu.sun3/xref/extra/ref-r" \l "roundew)(x) (WORD \*)( (3 + (WORD)(x)) & ~03 ) \*/**

**if ( ssize < MINSTK || priority < 1 ||**

**(pid=newpid()) == SYSERR ||**

**((saddr=getstk(ssize)) == NULL ) ) {**

**/\*#define getstk(n) getmem(n)\*/**

**restore(ps);**

**return(SYSERR);**

**}**

**numproc++;**

**pptr = &proctab[pid];**

**pptr->pstate = PRSUSP;**

**for (i=0 ; i<PNMLEN ; i++)**

**pptr->pname[i] = (\*namep ? \*namep++ : ' ');**

**pptr->pname[PNMLEN]='\0';**

**pptr->pprio = priority;**

**pptr->phasmsg = 0; /\* no message \*/**

**pptr->pbase = saddr;**

**pptr->plen = ssize;**

**sp = (int \*) (saddr+ssize); /\* simulate stack pointer \*/**

**sp -= 4; /\* a little elbow room \*/**

**pptr->pargs = nargs;**

**a = (&args) + nargs; /\* point past last argument \*/**

**for ( ; nargs > 0 ; nargs--) /\* machine dependent; copy args \*/**

**\*(--sp) = \*(--a); /\* onto created process' stack \*/**

**\*(--sp) = (int)INITRET; /\* push on return address \*/**

**\*(--sp) = (int)procaddr; /\* simulate a context switch \*/**

**--sp ; /\* 1 word for bp \*/**

**\*(--sp) = INITF; /\* FLAGS value \*/**

**sp -= 2; /\* 2 words for si and di \*/**

**pptr->pregs = (char \*)sp; /\* save for context switch \*/**

**pptr->paddr = procaddr;**

**restore(ps);**

**return(pid);**

**}**

## 

כלל**: תהליך יכול להופיע, בכל זמן נתון, בתור אחד לכל היותר. מספר התהליכים והתורים השונים מוגבל.**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Key | Next | Prev |
| 0 |  |  | 5 |
| 1 | 25 | 33 | 3 |
| 2 |  |  |  |
| 3 | 14 | 1 | 32 |
| 4 |  |  |  |
| 5 |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
| NPROC-1 |  |  |  |
| NPROC |  |  |  |
|  |  |  |  |
| Head at 32 | MININT | 3 | -1 |
| Tail at 33 | MAXINT | -1 | 1 |
|  |  |  |  |
|  |  |  |  |

**q.h**

**---------------------------------------------------------**

**#define NQENT NPROC+NSEM+NSEM+4/\*for ready&sleep \*/**

**struct qent**

**{**

**/\* one for each process plus two for each list \*/**

**Int qkey; /\*key on which the queue is ordered \*/**

**int qnext; /\* pointer to next process or tail \*/**

**int qprev; /\*pointer to previous process or head \*/**

**};**

**extern struct qent q[];**

**extern int nextqueue;**

**/\* inline list manipulation procedures \*/**

**#define isempty(list) (q[(list)].qnext >= NPROC)**

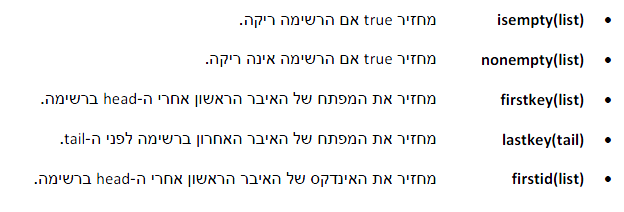
**#define nonempty(list) (q[(list)].qnext < NPROC)**

**#define firstkey(list) (q[q[(list)].qnext].qkey)**

**#define lastkey(tail) (q[q[(tail)].qprev].qkey)**

**#define firstid(list) (q[(list)].qnext)**

**#define EMPTY -1/\* equivalent of null pointer \*/**

****

****

****

**insert.c**

**#include <conf.h>**

**#include <kernel.h>**

**#include <q.h>**

**int insert(proc, head, key)**

**int proc; /\* process to insert \*/**

**int head; /\* q index of head of list \*/**

**int key; /\* key to use for this process \*/**

**{**

**int next; /\* runs through list \*/**

**int prev;**

**next = q[head].qnext;**

**while(q[next].qkey<key) /\*tail has MAXINT as key \*/**

**next = q[next].qnext;**

**q[proc].qnext = next;**

**q[proc].qprev = prev = q[next].qprev;**

**q[proc].qkey = key;**

**q[prev].qnext = proc;**

**q[next].qprev = proc;**

**return(OK);**

**}**

**queue.c**

**#include <conf.h>**

**#include <kernel.h>**

**#include <q.h>**

**int enqueue(item, tail)**

**int item; /\* item to enqueue on a list \*/**

**int tail; /\* index in q of list tail \*/**

**{**

**Struct qent \*tptr; /\* points to tail entry \*/**

**Struct qent \*mptr; /\* points to item entry \*/**

**tptr = &q[tail];**

**mptr = &q[item];**

**mptr->qnext = tail;**

**mptr->qprev = tptr->qprev;**

**q[tptr->qprev].qnext = item;**

**tptr->qprev = item;**

**return(item);**

**}**

**int dequeue(item)**

**int item;**

**{**

**Struct qent \*mptr; /\* pointer to q entry for item \*/**

**mptr = &q[item];**

**q[mptr->qprev].qnext = mptr->qnext;**

**q[mptr->qnext].qprev = mptr->qprev;**

**return(item);**

**}**

**getitem.c**

**#include <conf.h>**

**#include <kernel.h>**

**#include <q.h>**

**int getfirst(head)**

**int head; /\* q index of head of list \*/**

**{**

**int proc; /\* first process on the list \*/**

**if ((proc=q[head].qnext) < NPROC)**

**return( dequeue(proc) );**

**else**

**return(EMPTY);**

**}**

**int getlast(tail)**

**int tail; /\* q index of tail of list \*/**

**{**

**int proc; /\* last process on the list \*/**

**if ((proc=q[tail].qprev) < NPROC)**

**return( dequeue(proc) );**

**else**

**return(EMPTY);**

**}**

**newqueue.c**

#include <conf.h>

#include <kernel.h>

#include <q.h>

**int newqueue()**

**{**

**Struct qent \*hptr; /\* address of new list head \*/**

**Struct qent \*tptr; /\* address of new list tail \*/**

**int hindex, tindex; /\* head and tail indexes \*/**

**hptr = &q[ hindex=nextqueue++ ];/\* nextqueue is global variable \*/**

**tptr = &q[ tindex=nextqueue++ ];/\* giving next used q pos. \*/**

**hptr->qnext = tindex;**

**hptr->qprev = EMPTY;**

**hptr->qkey = MININT;**

**tptr->qnext = EMPTY;**

**tptr->qprev = hindex;**

**tptr->qkey = MAXINT;**

**return(hindex);**

**}**

**resched.c**

**#include <conf.h>**

**#include <kernel.h>**

**#include <proc.h>**

**#include <q.h>**

**int resched()**

**{**

**register struct pentry \*optr;/\* pointer to old process entry \*/**

**register struct pentry \*nptr;/\* pointer to new process entry \*/**

**optr = &proctab[currpid];**

**if ( optr->pstate == PRCURR )**

**{**

**/\* no switch needed if current prio. higher than next \*/**

**/\* or if rescheduling is disabled ( pcxflag == 0 ) \*/**

**if ( sys\_pcxget() == 0 || lastkey(rdytail) < optr->pprio**

**|| ( (lastkey(rdytail) == optr->pprio) && (preempt > 0) ) )**

**return;**

**/\* force context switch \*/**

**optr->pstate = PRREADY;**

**insert(currpid,rdyhead,optr->pprio);**

**} /\* if \*/**

**else if ( sys\_pcxget() == 0 )**

**{**

**kprintf("pid=%d state=%d name=%s",**

**currpid,optr->pstate,optr->pname);**

**panic("Reschedule impossible in this state");**

**} /\* else if \*/**

**/\* remove highest priority process at end of ready list \*/**

**nptr = &proctab[ (currpid = getlast(rdytail)) ];**

**nptr->pstate = PRCURR; /\* mark it currently running \*/**

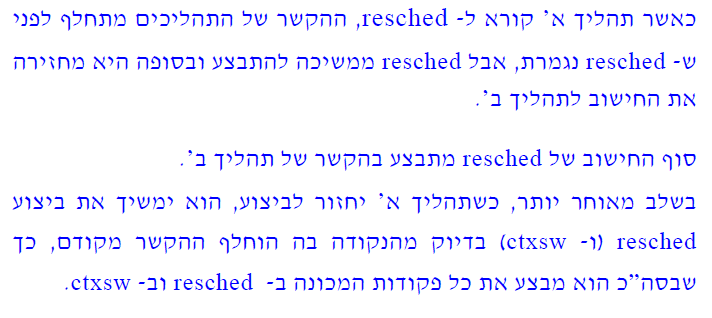
**preempt = QUANTUM; /\* reset preemption counter \*/**

**ctxsw(&optr->pregs,&nptr->pregs);**

**/\* The OLD process returns here when resumed. \*/**

**return;**

**}**

****

**ctxsw.asm**

**include dos.asm**

**dseg**

**; null data segment**

**endds**

**pseg**

**public \_ctxsw**

**;-------------------------------------------------------------------------**

**; \_ctxsw -- context switch**

**;-------------------------------------------------------------------------**

**; void ctxsw(opp,npp)**

**; char \*opp, \*npp;**

**;---------------------------------------------------------------------**

**; Stack contents upon entry to ctxsw:**

**; SP+4 => address of new context stack save area**

**; SP+2 => address of old context stack save area**

**; SP => return address**

**; The addresses of the old and new context stack save areas are**

**; relative to the DS segment register, which must be set properly**

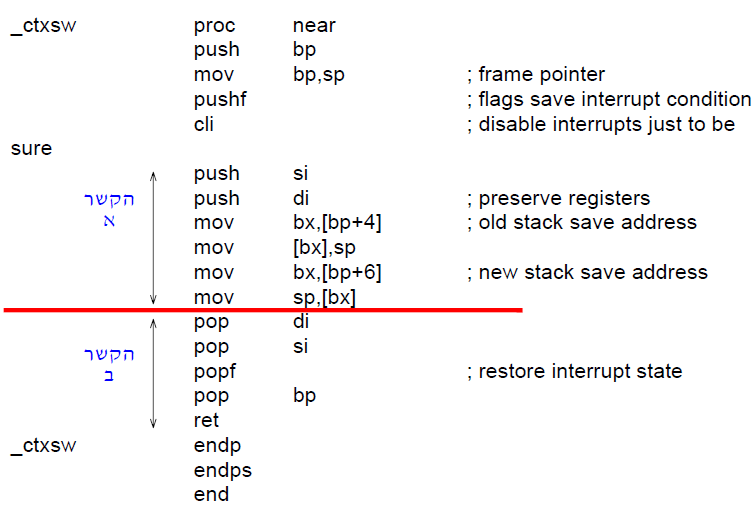
**; to access the save/restore locations.**

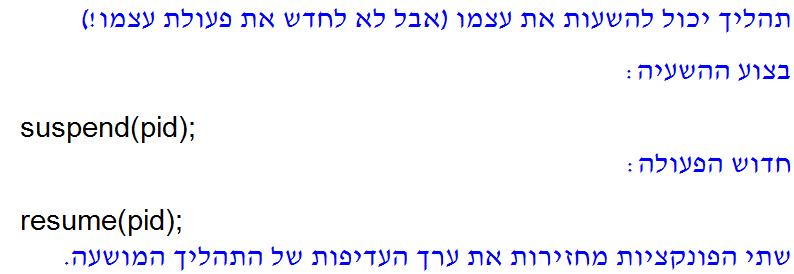
**;**

**; The saved state consists of the current BP, SI and DI registers,**

**; and the FLAGS register**

**;---------------------------------------------------------------------**

****

****

**suspend.c**

#include <conf.h>

#include <kernel.h>

#include <proc.h>

**SYSCALL suspend(pid)**

**int pid; /\* id of process to suspend \*/**

**{**

**Struct pentry \*pptr;/\*pointer to proc. tab. entry \*/**

**int ps; /\* saved processor status \*/**

**int prio; /\* priority returned \*/**

**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);**

**}**

**resume.c**

**#include <conf.h>**

**#include <kernel.h>**

**#include <proc.h>**

**SYSCALL resume(pid)**

**int pid;**

**{**

**int ps; /\* saved processor status \*/**

**struct pentry \*pptr;/\*pointer to proc. tab. entry \*/**

**int prio; /\* priority to return \*/**

**disable(ps);**

**if (isbadpid(pid) || (pptr = &proctab[pid])->pstate != PRSUSP) {**

**restore(ps);**

**return(SYSERR);**

**}**

**prio = pptr->pprio;**

**ready(pid);**

**resched();**

**restore(ps);**

**return(prio);**

**}**

**ready.c**

#include <conf.h>

#include <kernel.h>

#include <proc.h>

#include <q.h>

**int ready (pid)**

**int pid; /\* id of process to make ready \*/**

**{**

**register struct pentry \*pptr;**

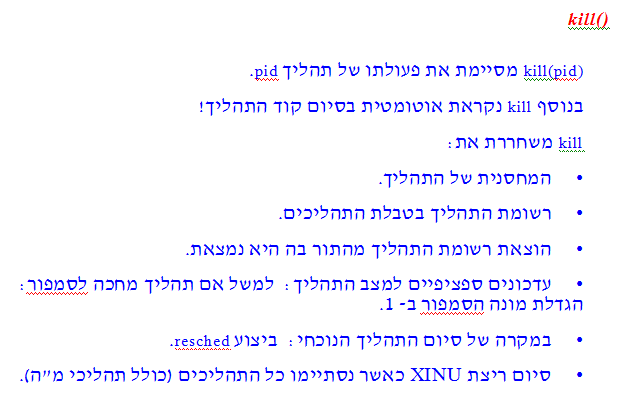
**pptr = &proctab[pid];**

**pptr->pstate = PRREADY;**

**insert(pid,rdyhead,pptr->pprio);**

**return(OK);**

**}**

****

**kill.c**

**#include <conf.h>**

**#include <kernel.h>**

**#include <proc.h>**

**#include <sem.h>**

**#include <mem.h>**

**#include <q.h>**

**#include <sleep.h>**

**SYSCALL kill(pid)**

**int pid; /\* process to kill \*/**

**{**

**Struct pentry \*pptr;/\*points to proc.table for pid\*/**

**int ps; /\* saved processor status \*/**

**int pstate;**

**disable(ps);**

**if (isbadpid(pid) || (pptr = &proctab[pid])->pstate==PRFREE) {**

**restore(ps);**

**return(SYSERR);**

**}**

**if (--numproc == 0)**

**xdone();**

**freestk(pptr->pbase, pptr->plen);**

**pstate = pptr->pstate;**

**pptr->pstate = PRFREE;**

**if (pstate == PRCURR)**

**resched();**

**else**

**if (pstate == PRWAIT)**

**{**

**semaph[pptr->psem].semcnt++;**

**dequeue(pid);**

**} /\* PRWAIT \*/**

**else**

**if (pstate == PRREADY)**

**dequeue(pid);**

**else**

**if (pstate == PRSLEEP)**

**{**

**if(q[pid].qnext < NPROC )/\*qnext is not tail \*/**

**q[q[pid].qnext].qkey += q[pid].qkey;/\*Account for the**

**pid delay \*/**

**dequeue(pid);**

**/\* The sleep queue may now be empty, or have a new first \*/**

**if ( slnempty = nonempty(clockq) )**

**sltop = &firstkey(clockq);**

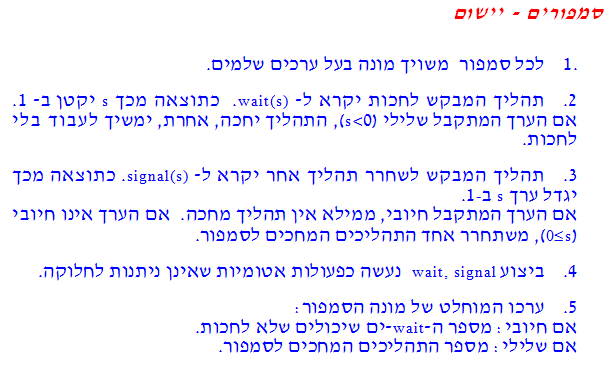
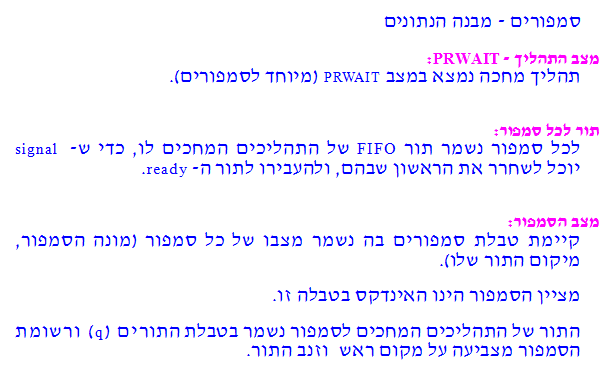
**} /\* PRSLEEP \*/**

**else; /\* default: PRSUSP, PRRECV \*/**

**restore(ps);**

**return(OK);**

**} /\* kill \*/**

** **

**sem.h**

**#ifndef NSEM**

**#define NSEM 45 /\*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)**

**---------------------------------------------------------** **wait.c**

**#include <conf.h>**

**#include <kernel.h>**

**#include <proc.h>**

**#include <q.h>**

**#include <sem.h>**

**SYSCALL wait(sem)**

**int sem;**

**{**

**int ps;**

**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);**

**}**

**signal.c**

**#include <conf.h>**

**#include <kernel.h>**

**#include <proc.h>**

**#include <q.h>**

**#include <sem.h>**

**SYSCALL signal(sem)**

**register int sem;**

**{**

**register struct sentry \*sptr;**

**int ps;**

**disable(ps);**

**if (isbadsem(sem) || (sptr = &semaph[sem])->sstate == SFREE) {**

**restore(ps);**

**return(SYSERR);**

**}**

**if ( sptr->semcnt++ < 0 )**

**{**

**ready(getfirst(sptr->sqhead));**

**resched();**

**}**

**restore(ps);**

**return(OK);**

**}**

**screate.c**

**#include <conf.h>**

**#include <kernel.h>**

**#include <proc.h>**

**#include <q.h>**

**#include <sem.h>**

**SYSCALL screate(count)**

**int count; /\* initial count (>=0) \*/**

**{**

**int 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);**

**}**

**LOCAL newsem()**

**{**

**int sem;**

**for (sem=0 ; sem < NSEM ; sem++) {**

**if (semaph[sem].sstate==SFREE) {**

**semaph[sem].sstate = SUSED;**

**return(sem);**

**}**

**}**

**return(SYSERR);**

**}**