5 More Process Management

Resumption, Suspension, Process Creation & Termination

- To temporarily stop a process from executing and later start it explicitly, a new process state is introduced : **suspended** animation, abbreviated to "**suspended**"
- This process state is, e.g., used in paging systems, to install pages referenced in user memory.
  - During this operation the process is suspended.

5.1 Process Suspension And Resumption

Suspend, resume, create are the operations associated with this new process state.

- **suspend :** removes a process from the ready queue or stops a currently executing process; parameter: process id.
- **resume :** moves a process from the suspended state to the ready list explicitly.
  - The currently executing process can suspend itself by calling *suspend(getpid())*.
- **create :** creates a new, independent, process as if it had been suspended during running, so *ctxsw* can return to it
  - This mechanism is called **pseudo call**.

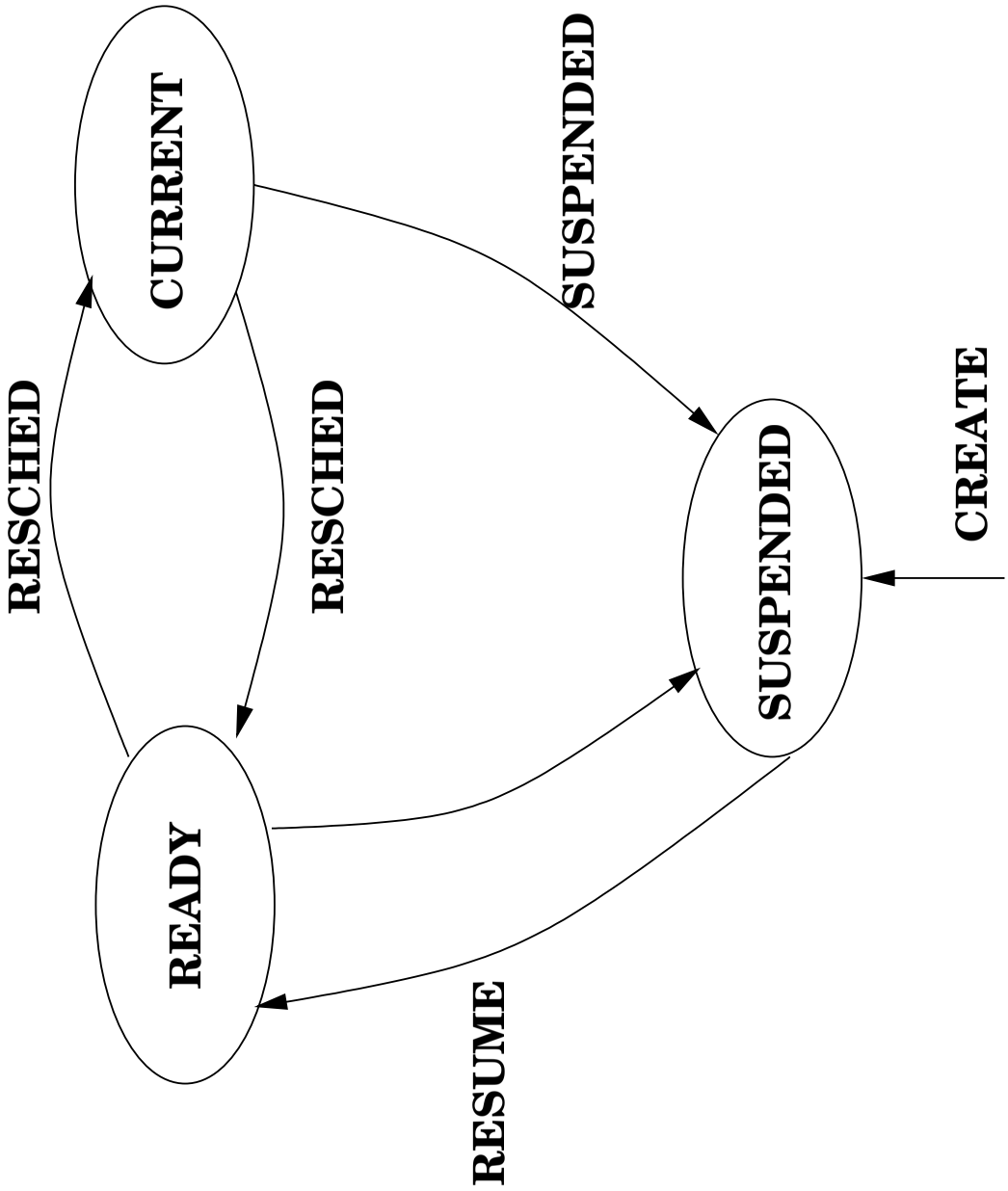

Figure 5.1: Transitions among the current, ready, and suepended states

5.1.1 implementation of Resume

```
/*------------------------------------------------------------------------
 * resume -- unsuspend a process, making it ready; return
 * the priority
 *------------------------------------------------------------------------
 */
SYSCALL resume(pid)
        int     pid;
{
        char    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, RESCHYES);
        restore(ps);
        return(prio);
}
```

The need for turning off when accessing the central data structure of Xinu

- Resume calls **resched**, and hence changes:
  - the process table (i.e., the entry for the resumed process)
  - the q-structure (since within the ready list is implemented)
- When a process would be simultaneously changing the process table - this situation is, in general, called **race condition**
- The final state of the process table is not clear.

  Comparison:

  ```
  {x=  } x:=1 || x:=2 {x= ?}
  ```

- To prevent such situation from happening:
  this ascertaining is prohibited by **disabling interrupts** while accessing the process table (or other certain data structure of the operating system).

5.2 System Calls

- The precaution *resume* takes, to verify that a process is legal, make it a general purpose routine that can be invoked by (unexperienced) user.
  - resume is a **system call**
- System calls define the exterior of the operating system by providing an interface through which the user accesses all system services
- System calls should protect the internal system from illegal sue.
- There is one process table in the system, shared by all its processes.
- A process can be sure that other process is trying to change the process table at the same time, so :
  - It should not call *resched()*, because rescheduling switches control to another process and this change the process table.
  - The system should not react to interrupts, since interrupt routines can call *resched* as well

5.2.1 Implementation of Suspend

```
/* suspend.c - suspend */

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

/*------------------------------------------------------------------------
 *  suspend  --  suspend a process, placing it in hibernation
 *------------------------------------------------------------------------
 */
SYSCALL suspend(pid)
        int     pid;            /* id of process to suspend   */
{
        struct  pentry  *pptr;/* pointer to proc. tab. entry*/
        char    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);
}
```

Calling resched()

- This kind of call presuppose that *resched* is called in parallel to the current process.
- Inter leaning the instructions of this concurrent (parallel) process in between the instructions of the concurrent process, implies that, e.g., the real-time clock procedure has interrupted the current process.
- However, this is impossible when the current process is deaf to interrupts.
  - the real-time clock interrupt is for then temporarily protected.
- To prevent Interrupts :
  - resume invokes procedure *disable(ps)* to set the process priority heigh to prevent interrupts.
  - to restore the situation before *disable(ps)* was called, one calls *restore(ps)*

Examples (invalid, inconsistent sate):

As wee will see later, high level interrupt routines enable the occurrence of other interrupts by calling *resched()*, if they need the CPU to switch temporarily to a new context, i.e., a process with interrupt enabled.

To prevent inconsistencies, the shared data structures should be left in a valid sate before calling any routines switches context

1. If *resched()* is called in between

   `pptr->pstate != PRCURR`

   `pptr->pstate != PRREADY`

   hight result in setting

   `pptr->pstate = PRCURR`

   leaving the system in an invalid system state ....

2. *Resched* occurring in between *dequeue(pid)* and

   `pptr->pstate = PRSUSP`

   could result in *enqueue(pid)* .....

3. Interrupts occurring in between

   `pptr->pstate = PRSUSP`

   and *resched* :might result in

   `pptr->pstate = PRCURR`

5.3 Process Termination

- *Suspend* freezes processes, but leaves them in the system so they can be resumed later.
- System call *kill* stops a process immediately and removes it from the system completely.
- See code for *kill* below
  - First *kill* checks its argument, **pid** to ensure it corresponds to a valid active process
  - it then decrements **numproc**
  - free stack space of this process (by calling procedure *freestk*)
  - unlinks the process from the ready list (*dequeue*)
  - and frees the process table entry (by assigning its state field **PRFREE**).
- Since process no longer appears on ready list, it will never regain control of CPU.
- A process can kill itself by calling *kill(getpid())*
  - if it is the last active process the *xdone()* is called
- its state is made **PRFREE** and *resched()* is called - this process can not be started again because it dose not longer appear on the ready list.

```
/* kill.c - kill */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <sem.h>
#include <mem.h>
/*------------------------------------------------------
 * kill -- kill a process and removes it from the system
 *------------------------------------------------------
 */
SYSCALL kill(pid)
        int pid;            /* process to kill             */
{
        struct pentry *pptr;/* points to proc.tab. for pid */
        char   ps;          /* saved processor status      */

        disable(ps);
        if(isbadpid(pid) ||
                        (pptr= &proctab[pid]->pstate==PRFREE){
                restore(ps);
                return(SYSERR);
        }
        if(--numproc == 0)
                xdone();
```

```
        freestk(pptr->pbase, pprt->pstklen);
        switch (pptr->pstate){

        case PRCURR:    pptr->pstate = PRFREE;/* suicide   */
                resched();
        case PRWAIT:    semph[pptr->psem].semcnt++;
        case PRSLEEP:                       /* fall through*/
        case PRREADY:   dequeue(pid);
                                /* fall through */
        default:        pptr->pstate = PRFREE;
        }
        restore(ps);
        return(OK);
}
```

```
/* xdone.c - xdone */

/*------------------------------------------------------
 * xdone --  print system completion message as last
 * process exits .
 *------------------------------------------------------
 */
xdone()
{
       printf("\n\nAll user process have completed.\n\n");
}
```

5.4 Kernel Declaration

The code below defines in-line procedures *disable* and *restore* used above, as well as other variables and symbolic constants used throughout Xinu :

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

/* Symbolic constants user throughout Xinu */

typedef char       Bool;     /* Boolean type                  */
#define FALSE      0         /* Boolean constants             */
#define TRUE       1
#define NULL       (char *)0 /*Null pointer for linked lists*/
#define SYSCALL    int       /* System call declaration     */
#define LOCAL      struct    /* Local procedure declaration */
#define INTPROC    int       /* Interrupt procedure         */
#define PROCESS    int       /* Process declaration         */
#define RESCHYES   1         /* tell ready to reschedule    */
#define RESCHNO    0         /* tell ready not to reschedule*/
#define MININT     0100000   /* minimum integer (-32768)    */
#define MAXINT     0077777   /* maximum integer             */
#define SP         6         /* reg. 6 is stake pointer     */
#define PC         7         /* reg. 7 is program counter   */
#define PS         8         /* proc. status in 8th reg. loc*/
#define MINSTK     40        /* minimum process stake size  */
#define NULLSTK    300       /* process 0 stake size        */
```

```
#define DISABLE    0340      /* PS to disable interrupts    */
#define OK         1         /* returned when system call ok*/
#define SYSERR    -1         /*returned when sys. call fails*/
#define INITARGC   1         /* initial process argc        */
#define INITSTK    200       /* initial process stack       */
#define INITPRIO   20        /* initial process priority    */
#define INITNAME   "main"    /* initial process name        */
#define INITRET    userret   /* processes return address    */
#define INITPS     0         /* initial process PS          */
#define INITREG    0         /* initial register contents   */
#define AUANTUM    10        /* clock ticks until preemption*/

/* misc. utility inline functions */

#define isodd(x)     (01&(int)(x))
#define disable(ps)  asm("mfps ~ps");asm("mtps $0340")
#define restore(ps)  asm("mtps ~ps")/*restore inturrupt status*/
#define enable()     asm("mtps $000")/* enable interrupts   */
#define pause()      asm("wait")/* machine "wait for interr.*/
#define halt()       asm("halt")/* machine halt instruction */

extern  int   rdyhead, rdytail;
extern  int   preempt;
```

5.5 Process Creation

System call *create* creates a new, independent process . So it

- lays down an exact image of the process as if it has been stopped while running, so *ctxsw* can switches to it.
- finds a free slot in the process table
- allocates space for the process' stake
- and fills in the process table entry.

See code below

```
/* create.c - create, newpid */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <mem.h>
#include <io.h>

/*------------------------------------------------------------------------
 *  create  -  create a process to start running a procedure
 *------------------------------------------------------------------------
 */
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();
disable(ps);
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;
for(i=0 ;i<PNMLEN && (pptr->pname[i]=name[i])!=0 ;i++)
        ;
pptr->pprio = priority;
pptr->pbase = (short)saddr;
pptr->pstklen = ssize;
pptr->psem = 0;
pptr->phasmsg = FALSE;
pptr->plimit = (short)((unsigned)saddr - ssize +
                                        sizeof(int));
*saddr-- = MAGIC;
```

```
        pptr->pargs = nargs;
        for (i=0 ; i<PNREGS ; i++)
                pptr->pregs[i]=INITREG;
        pptr->pregs[PC] = pptr->paddr = (short)procaddr;
        pptr->pregs[PS] = INITPS;
        pptr->pnxtkin = BADPID;
        pptr->pdevs[0] = pptr->pdevs[1] = BADDEV;
        a = (&args) + (nargs-1); /* point to last argument   */
                                 /* machine dependent; copy args*/
        for ( ; nargs > 0 ; nargs--)
                *saddr-- = *a--;/*onto created process' stack*/
        *saddr = (int)INITRET;   /* push on return address   */
        pptr->pregs[SP] = (int)saddr;
        restore(ps);
        return(pid);
}
```

```
/*------------------------------------------------------
 * newpid -- obtain a new (free) process id
 *------------------------------------------------------
 */
SYSCALL   newpid()
{
        int     pid;            /* process id to return */
        int     i;
                                /* check all NPROC slots*/
        for (i=0 ; i<NPROC ; i++) {
                if ( (pid=nextproc--) <= 0)
                        nextproc = NPROC-1;
                if (proctab[pid].pstate == PRFREE)
                        return(pid);
        }
        return(SYSERR);
}
```

```
#include <conf.h>
#include <kernel.h>
/*------------------------------------------------------
 * userret -- entered when a process exits by return
 *------------------------------------------------------
 */
userret()
{
  kill( getpid() );
}
```

5.6 Utility Procedures

Following codes show the implementation of the system calls: *getpid, getprio* and *chprio*

- *Getpid* allows a process to obtain its process id.
  - *userret* shows one reason, how a procedure needs to know the process id of the process executing it.
- *Getprio* allows a process to obtain aprocess' scheduling priotity.
- *Chprio* allows a process to change a process' scheduling priority.

See the implementation below !

```
/* getprio.c - getprio */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
/*------------------------------------------------------------------------
 * getprio -- return the scheduling priority of a given process
 *------------------------------------------------------------------------
 */
SYSCALL getprio(pid)
        int     pid;
{
        struct  pentry  *pptr;
        char    ps;

        disable(ps);
        if (isbadpid(pid) ||
            (pptr = &proctab[pid])->pstate == PRFREE){
                restore(ps);
                return(SYSERR);
        }
        restore(ps);
        return(pptr->pprio);
}
```

```
/* getpid.c -- getpid */

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

/*------------------------------------------------------------
 * getpid -- get the process id of current executing process
 *------------------------------------------------------------
 */
SYSCALL getpid()
{
        return(currpid);
}
```

```
/* chprio.c - chprio */

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

/*------------------------------------------------------------
 * chprio -- change the scheduling priority of a process
 *------------------------------------------------------------
 */
SYSCALL chprio(pid, newprio)
        int    pid;
        int    newprio;             /* newprio > 0       */
{
        int    oldprio;
        struct pentry  *pptr;
        char   ps;


        disable(ps);
        if (isbadpid(pid) || newprio <= 0 ||
            (pptr = &proctab[pid])->pstate == PRFREE){
                restore(ps);
                return(SYSERR);
        }
```

```
        oldprio = pptr->pprio;
        pptr->pprio = newprio;
        restore(ps);
        return(oldprio);
}
```