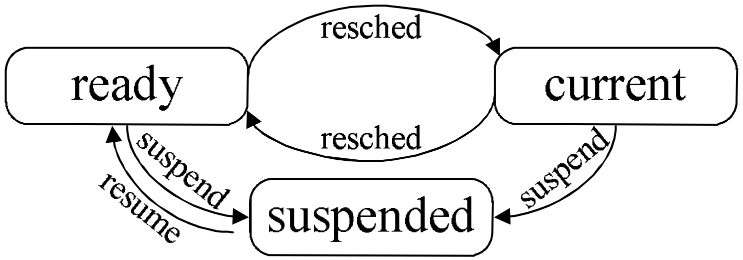
5 More Process Management: Resumption, Suspension, Process Creation & Termination

5.1 Process Suspension and Resumption

- 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 programs in **primary** (code) memory from **secondary** (disk) memory. During this operation the process is **suspended**.

Process Suspension and Resumption



- `suspend` removes a process from the ready list 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 stopped during running**, so `ctxsw` can return to it; this mechanism is called **pseudocall**.

Process Suspension and Resumption:

resume

```
/* resume.c - resume */

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

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

Process Suspension and Resumption:

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     */
{
        STATWORD ps;
        struct  pentry  *pptr;          /* pointer to proc. tab. entry  */
        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) {
                pptr->pstate = PRSUSP;
                dequeue(pid);
        }
        else {
                pptr->pstate = PRSUSP;
                resched();
        }
        prio = pptr->pprio;
        restore(ps);
        return(prio);
}
```

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

- `resume` calls `resched`, and hence changes the **process table** (i.e., the entry for the resumed process), the q-**structure** (since within it the ready list is implemented).
- What happens when another process would be simultaneously changing the process table? **The final state of the process table is not clear!** This situation is, in general, called **race condition**:

```
{ x = 0 }    x := 1 || x := 2     { x = ? }
```

- To prevent such situation from happening: this uncertainly 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) users. So `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. They should protect the internal system from illegal use.
- There is one process table in the system, shared by all its processes. **How can a process be sure** that no other process is trying to change the process table **at the same time**?
  - 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!

System calls

- How can `resched()` be called? See process table on page 128: by calling `sleep`/`wakeup`, `receive`/`send`, `wait`/`signal`, `suspend`/`resume`.
- However this kind of call presupposes that `resched()` is called **in parallel to the current process**.
- Interleaving 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**. For then the real-time clock interrupt is temporarily postponed.
- To prevent Interrupts `resume` invokes procedure `disable(ps)` to set the process priority high enough (`034i`) to prevent interrupts. To restore the situation before `disable(ps)` was called, one calls `restore(ps)`. But does this necessarily imply that the **process reacts to interrupts again**?

System calls

- As we will see in chapter 9, **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.

  (E.g., to make room in a character buffer when reading characters from a terminal, see chapters 11 and 12, or to read in characters to an output buffer when the output buffer is empty and characters should be printed.)

- To **prevent inconsistencies**, the shared data structures should be **left in a valid sate** before calling **any routines that switches context**. For example, the following states are **invalid**:
  - If `resched()` is called in between `pptr->pstate != PRCURR` and `pptr->pstate != PRREADY`, this might result in setting state to `PRCURR`
  - `resched` occurring in between `dequeue(pid)` and `pptr->pstate = PRSUSP` could result in `enqueue(pid)`
  - 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**.
- First, `kill` checks its argument `pid` to ensure it corresponds to a **valid active process**. Then it **decrements** `numproc`, it **frees the 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 the process no longer appears on the ready list, it will **never regain control of CPU**.
- A process can **kill itself** by calling `kill(getpid())`. Its state is made **PRFREE** and `resched()` is called – this process can not be started again because it does no longer appear on the ready list.
- If the last active process is killed, the procedure `xdone()` is called.

Process Termination: kill

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

#include "conf.h"
#include "kernel.h"
#include "proc.h"
#include "sem.h"
#include "mem.h"

/*------------------------------------------------------------------------
 * kill  --  kill a process and remove it from the system
 *------------------------------------------------------------------------
 */
SYSCALL kill(pid)
        int     pid;                    /* process to kill              */
{
        struct  pentry  *pptr;          /* points to proc. table for pid*/
        char    ps;                     /* saved processor status       */

        disable(ps);
        if (isbadpid(pid) || (pptr= &proctab[pid])->pstate==PRFREE) {
                restore(ps);
                return(SYSERR);
        }
        freestk(pptr->pbase, pptr->pstklen);
        switch (pptr->pstate) {
          case PRCURR:  pptr->pstate = PRFREE;  /* suicide */
                        resched();

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

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

5.4 Kernel Declaration

- The code in `kernel.c` defines in-line procedures `disable` and `restore` used above, as well as other variables and symbolic constants used throughout Xinu.
- Constants like `INITSTK`, `INITPRIO` and `INITNAME` are used on page 196 when creating the null process:

  > *Once initialization is complete and a process has been created to execute the user's program, the null process just falls into an infinite loop, giving **`resched`** a process to schedule when no user processes are ready to run.*

Kernel Declarations

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

/* Symbolic constants used 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       static         /* 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 resch.    */
#define MININT      0100000        /* minimum integer (-32768)    */
#define MAXINT      0077777        /* maximum integer             */
#define SP          6              /* reg. 6 is stack pointer     */
#define PC          7              /* reg. 7 is program counter   */
#define PS          8              /* proc. status in 8th reg. loc */
#define MINSTK      40             /* minimum process stack size  */
#define NULLSTK     300            /* process 0 stack size        */
#define DISABLE     0340           /* PS to disable interrupts    */
#define OK           1             /* system call ok              */
#define SYSERR      -1             /* system call failed          */

/* Initialization constants */

#define INITARGC    1              /* initial process arcs        */
#define INITSTK     500            /* 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 QUANTUM     10             /* clock ticks until preemption */

/* Miscellaneous 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 interrupt 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` in `resched` can switch to it; it fills a free (unused) slot in the **process table** and allocates space for the **process' stack**. The initial process stack looks like this:

...
arg_n
...
arg_1
returnaddress

- The return address is used **after** the created process has terminated, so it contains the address of procedure `userret` which calls `kill(getpid())`.
- We refer to the initial process stack as a **pseudo-call** because `create` carefully pushes values on it to **simulate a procedure call** (in C: **arguments and a return address**). The process begins executing the code, obeying the calling conventions for accessing arguments and allocating local variables – so **it behaves exactly as if it had been called from another procedure**.

Process Creation: create

```
/*------------------------------------------------------------------------
 *  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->plimit = (short)(saddr - ssize + 1);
        pptr->pargs = nargs;
        for (i=0 ; i<PNREGS ; i++)
                pptr->pregs[i]=INITREG;
        pptr->pregs[PC] = pptr->paddr = (short)procaddr;
        pptr->pregs[PS] = INITPS;
        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       */
        pptr->pregs[SP] = (int)saddr;
        restore(ps);
        return(pid);
}
```

Process Creation: newpid and userret

```
/*------------------------------------------------------------------------
 * newpid  --  obtain a new (free) process id
 *------------------------------------------------------------------------
 */
LOCAL   newpid()
{
        int     pid;                    /* process id to return         */
        int     i;

        for (i=0 ; i<NPROC ; i++) {     /* check all NPROC slots        */
                if ( (pid=nextproc--) <= 0)
                        nextproc = NPROC-1;
                if (proctab[pid].pstate == PRFREE)
                        return(pid);
        }
        return(SYSERR);
}


/* userret.c - userret */

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

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

5.6 Utility Procedures

- Following code shows the implementation of the system calls: `getpid`, `getprio` and `chprio`
- `getpid` allows a process to obtain its process id – `userret` shows one reason why a procedure needs to know the process id of the process executing it.
- `getprio` allows a process to obtain a process' scheduling priority, and `chprio` allows a process to change a process' priority.

Utility Procedures: getprio and getpid

```
/* 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 currently executing process
 *------------------------------------------------------------------------
 */
SYSCALL getpid()
{
        return(currpid);
}
```