4.1 Context switching

What is Context switching ?

Operating system achieves **the illusion of concurrent processing** by rapidly switching one processor among several **computations**

Context switching at the heart of the processor juggler act :

consists of stopping the current computation, saving informations needed for restarting it later, and starting a possibly different one.

Why is this job difficult ?

The CPU can not be stopped at all- it continues to execute code until it switches to another process.

This chapter describes basic context switch mechanism showing :

- how a process saves its state informations
- chooses another process to run
- return control to that process.

It describes also the basic data structure, the **process table**, holding all information about processes.

4.2 Process Table

Information about processes kept in process table **proctab**, an array of structure **pentry** :

- One entry for each process in this table
- Because only one process running at the time,

  one of these entries is out of date, since corresponding with the currently active process.
- The other entries correspond to processes witch are temporarily halted.

Exactly what information must be saved in proctab ?

- All values that will be destroyed when another process runs.
- E.G., since in Xinu each process has its own segment of the stack, no copy of the stack needs to be saved.
- However new process will change the contents of the machine registers, so this must be saved.
- Also: information used to control process and account for its resource.
- Processes are referenced by their process **id**, witch is the index of the saved-state-info in **proctab**
- pbase, pstklen, plimit used ti free memory space when process completes.

```
/* proc.h - isbadpid */

/* process table declaration and defined contents             */

#ifndef NPROC                 /* set the number of processes */
#define NPROC           10    /* allowed if not already done */
endif

/* process state contents */

#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*/

/* miscellanous process definetions */

#define PNREGS           9    /* size of saved register area  */
#define PNMEN            8    /* length of process "name"     */
#define NULLPROC         0    /* id of the null process; it   */
                              /* is always eligible to run    */
#define isbadpid(x)      (x<=0 || x>=NPROC)
```

```
/* process table entry */

struct pentry  {
       char    pstate;          /* process state: PRCURR, etc  */
       short   pprio;           /* process priority            */
       short   pregs[PNREGS];   /* saved regs. R0-R5, SP,PC,PS */
       short   psem;            /* semaphore if process waiting*/
       short   pmsg;            /* message sent to this waiting*/
       short   phasmsg;         /* nonzero if pmsg is vaild    */
       short   pbase;           /* base of run time stack      */
       short   pstklen;         /* stack length                */
       short   plimit;          /* lowest extent of stack      */
       char    pname[PNMLEN];   /* process name                */
       short   pargs;           /* initail number of arguments */
       short   paddr;           /* initial code address        */
};

extern struct pentry proctab[];
extern int    numproc;          /* currently active processes  */
extern int    nextproc;         /* search for free slot        */
extern int    currpid;
```

4.3 Process State

- Needed for checking validity of operations performed on a process.

4.4 Selecting A Ready Process

- A process is classified as **ready** when it is eligible for CPU but not currently executing.
- The single process served by the CPU is classified as **current**.
- Switching context involves :
  - selecting a process from those that are ready or current
  - giving control to selected process

Scheduler :

Software implementing the policy to select a process from among those that are ready or current is called scheduler.

Xinu policy :

- At any time, the highest priority process eligible for CPU service is executing.
- Among processes with equal priority scheduling is **Round Robin** :
  - all processes with equal priority are selected one-after-one so that all members of this set are executed before any member has a second opportunity.

Priorities :

- kept in **pprio**-field of process table entry, are positive integers.
- All ready processes appear in the ready list ordered by priority .

- **In Xinu :** current process not on ready list, it its process id gives by global variable **currpid**.
- What happens to current executing process during a context switch ?
  - After, it remains eligible to use CPU, even when control is temporarily passed to another process, its current process state changes to **PRREADY** and it is moved to ready list for later CPU service.
- How dose the rescheduler, **resched**, decide whether to move current process to ready list ?
  - If current process is not eligible to use CPU, the system routines assign to its **pstate**-field a desired next state before calling **resched**:
    - See e.g. in suspend, pg. 69[comer]
    - See e.g. in kill, pg. 71[comer]
  - When **resched** prepares to switch context, it checks **pstate**-field of current process and makes it ready for executing CPU, only if its **pstate** is **PCURRENT**.

Resched :

- selects a process to run
- changes process table for that entry
- removes new process from ready list and makes it current
- update **currpid**
- rests preemption counter
- calls *ctxsw*

  `(optr <-- pregs, nptr <--pregs)`

  to save and reset the machine registers.

See code on the next slide!.

```
/* resched.c - resched */

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

/*------------------------------------------------------------------------
 * resched -- reschedule processor to highest priority ready
 * process
 *  Notes:    Upon entry, currpid gives current process id.
 *            Proctab[currpid].pstate gives correct NEXT state
 *            for current process if other than PRCURR.
 *------------------------------------------------------------------------
 */
int    resched()
{                               /* pointer to old process entry */
       register struct pentry *optr;
                                /* pointer to new process entry */
       register struct pentry *nptr;

/*no switch needed if current process priority higher than next*/
       if(((optr= &proctab[currpid])->pstate == PRCURR) &&
           (lastkey(rdytail)<optr->pprio))
       return(OK);
```

```
        /* force context switch */

        if(optr->pstate == PRCURR){
       optr->pstate = PREADY;
       insert(currpid,rdyhead,optr->pprio);
        }

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

        nptr = &proctab[ (currpid = getlast(rdytail)) ];
        nptr->pstate = PRCURR;     /* mark it currently running */
#ifdef RTCLOCK
        preempt  = QUANTUM;
#endif
        ctxsw(optr->pregs,nptr->pregs);

                    /* The OLD process returns here when resumed. */
        return(OK);
}
```

Which processes call resched ?

Answer: The process which is executed under Xinu on this exact moment. See : process state diagram below.

- Also *resched()* is a normal procedure, and calling it results in executing it like a normal procedure.
- *int resched()*{ ... *ctxsw(optr-¿pregs,nptr-¿pregs);*
  return address : *return(OK)*}
- **ctxsw** is not a normal procedure call. SO after executing it one dosen't return *immediately* to the return address.

ctxsw(...,...)

- Code of *ctxsw(,)* is machine dependent because machine registers should be saved by it.
- PC must be changed last to give the CPU the opportunity executing the new process (and not earlier), once information about old process stored in proctab und stack.
- **On the 11/2:**
  - the *rtt* instruction pops both PS and PC from the stack, and reload them **in one step**.
  - After saving regs associated with the old process in the register save area of that process
  - *ctxsw* switches stack to new process stack, restore the values of new registers
  - after having pushed new values of PS and PC on stack
  - it resets the PS and PC in one go by calling *rtt*

ctxsw

Calling procedure pushes the call's actual arguments on the stack in reverse order, pushes the return address on the stack, and then branches to the called routine.

```
/* ctxsw.s - ctxsw */

/*------------------------------------------------------------
/* ctxsw -- actually perform context switch, saving/loading
/* registers
/*------------------------------------------------------------
   / The stack contains three items upon entry to this routine :
   /
   /     SP+4 => address of 9 word save area with new
   /     registers + PS
   /     SP+2 => address of 9 word save area for old
   /     registers + PS
   /     SP   => return address
   /
   / The saved state contains of : the saved values of R0-R5
   / upon entry, SP+2, PC equal to  the return  address, and
   / the PS (i.e., the PC and SP are saved as if the calling
   / process had returned to its caller).

        .globl _ctxsw    / declare the routine name global
_ctxsw:                   / entry point to context switch
        mov     r0,*2(sp) / Save old R0 in old register area
        mov     2(sp),r0  / Get address of old register area
        add     $2,r0     / in R0;  increment to saved pos.ofR1
        mov     r1,(r0)+  / Save registers R1-R5 in successive
```

```
mov     r2,(r0)+  / locations of the old process
mov     r3,(r0)+  / register save area. (r0)+ denores
mov     r4,(r0)+  / indirect reference and, as a side
mov     r5,(r0)+  / effect, incrementing r0 to next word.
add     $2,sp     / move sp beyond the return address,
                  / as if a return had occurred.
mov     sp,(r0)+  / save stack pointer
mov     -(sp),(r0)+/ Save caller's return address as PC
mfps    (r0)    / Save processor status beyond registers
mov     4(sp),r0 / Pick up address of new registers in R0
                  / Ready to load registers for the new
           / process and abandon the old stack.
mov     2(r0),r1   / Load R1-R5 and Sp from the saved are
mov     4(r0),r2   / for the new process.
mov     6.(r0),r3  / Note: dot following a number makes i
mov     8.(r0),r4  / decimal; all others are octal
mov     10.(r0),r5
                  / Have now actually switched stack
mov     12.(r0),sp
                  / Push new process PS on new process s
mov     16.(r0),-(sp)
                   / Push new process PC on new process
mov     14.(r0),-(sp)
mov     (r0),r0     / Finally, load R0 from new area
rtt                 / Load PC, PS, and reset SP all at on
```

4.5 Null Process

- *resched()* resumes that at least one process is available on the ready queue
  it dose not bother to verify whether ready list is empty, so :

  Resched can only switches context from one process to another, so at least one process must always remain on ready queue.

- To ensure that a ready process always exists
  - Xinu creates an extra process, the **null process**, when it initializes the system.
  - It has pid zero and *pprio* zero; it is code contains of an infinite loop
- Because user processes have all priority ¿ 0,
  - the scheduler switches to the null process only when no user process ready to run
- So null process current running when processes deadlock
- When the real time clock interrupt routine decreases runtime clock until new process, if any, get ready run when their time expires.
- These are then put on ready queue, after which *resched* is called.

4.6 Making A Process Ready

Making a process eligible for CPU service accuse so often that a special procedure has been designed to do so

ready(pid,resched)

- In principle putting a process on the ready list schould result in a call to reshed to make sure that the process with the highest priority is running
- However, sometimes this results in a too heavy overhead in execution time when many processes are put on the ready queue
- Then all these processes are put in one go on the ready list without rescheduling after which *resched()* is called to make sure that the process with the highest priority is running.
- Thus construction is made possible by providing *read(..,..)* with a **boolean** argument **RESCHEDNO/RESCHEDYES** determining whether *resched(...,resch)* is called.
  See the code below !

```
/* read.c - ready */
#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <q.h>

/*------------------------------------------------------------------------
 * ready -- make  a process eligible for CPU service
 *------------------------------------------------------------------------
 */
int     ready (pid, resch)
        int     pid;            /* id of process to make ready */
        int     resch;          /* reschedule afterward        */
{
        register structer *pptr;

        if (isbadpid(pid))
        return(SYSERR);
        pptr = &proctab[pid];
        pptr->pstate = PREADY;
        insert(pid,rdyhead.pptr->pprio);
        if (resch)
        resched();
        return(OK);
}
```