4 Scheduling and Context Switching

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 juggling act: consists of
  1. stopping the current computation
  2. saving informations needed for restarting it later and
  3. 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 information**, **chooses** another process to run and **returns control** to that process. It describes also the basic **data structure**, the **process table**, holding all information about processes.

4.1 Process Table

- Information about processes are kept in the **process table** `proctab`, an array of `pentry`-structure:
  - There's **one entry for each process** in this table.
  - Because only one process is **running** at a time, one of these entries is **out-of-date**, since corresponding with the currently active process.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 a new process will change the contents of the **machine registers**, so this must be saved. Also: information used to control the process and account for its resource.
- Processes are referenced by their **process id**, witch is the index of the saved-state-info in `proctab`; e.g. `pbase`, `pstklen`, and `plimit` are used to **free memory space** when process completes.

Process Table in proc.h

```
#ifndef     NPROC                   /* set the number of processes          */
#define     NPROC      30           /*  allowed if not already done         */
#endif

/* miscellaneous process definitions */

#define     PNREGS     9            /* size of saved register area          */
#define     PNMLEN     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               / see */
        short  pregs[PNREGS];       /* saved regs. R0-R5,SP,PC,PS <--|  pg. */
        short  psem;                /* semaphore if process waiting   \ 24  */
        short  pmsg;                /* message sent to this process         */
        short  phasmsg;             /* nonzero iff pmsg is valid            */
        short  pbase;               /* base of run time stack               */
        short  pstklen;             /* stack length                         */
        short  plimit;              /* lowest extent of stack               */
        char   pname[PNMLEN];       /* process name                         */
        short  pargs;               /* initial number of arguments          */
        short  paddr;               /* initial code address                 */
};

extern  struct pentry proctab[];
extern  int    numproc;             /* currently active processes  \  see   */
extern  int    nextproc;            /* search point for free slot   | page  */
extern  int    currpid;             /* currently executing process /  196   */
```

4.2 Process States

- Process states are needed for **checking validity** of **operations performed on a process**.
- The following process states are defined in `proc.h`:

```
/* process state constants */

#define  PRCURR   '\001'  /* process is currently running  -- chapter  4 */
#define  PRFREE   '\002'  /* process slot is free          -- chapter  4 */
#define  PRREADY  '\003'  /* process is on ready queue     -- chapter  4 */
#define  PRRECV   '\004'  /* process waiting for message   -- chapter  7 */
#define  PRSLEEP  '\005'  /* process is sleeping           -- chapter 10 */
#define  PRSUSP   '\006'  /* process is suspended          -- chapter  5 */
#define  PRWAIT   '\007'  /* process is on semaphore queue -- chapter  6 */
```

4.3 Selecting a Ready Process

- A process is classified as **ready** (`PRREADY`) when it is **eligible for CPU** but **not currently executing**.
- The **single process served by the CPU** is classified as **current** (`PRCURR`).
- Switching context involves **selecting a process** from those that are ready or current and **giving control** to selected process.
- The software implementing the **policy to select a process** from among those that are ready or current is called **scheduler**.

Selecting a Ready Process: Xinu's Policy

- XINU's policy for selecting a process is:

  At any time, the highest priority process eligible for CPU service is executing!

  Among processes with equal priority, scheduling is round-robin – i.e., all processes with equal priority are selected one-after-another so that **all members of this set are executed before any member has a second opportunity**.

- **Priorities** are **positive integers** kept in `pprio`-field of process table entry.
- All ready processes appear in the **ready list ordered by priority** (a priority queue in the q-structure). Global variables `rdyhead` and `rdytail` point to head and tail of this q-structure.
- In XINU, the **current process** is **not on ready list**, its process identifier is given by the global variable `CURRPID`.

Selecting a Ready Process: Xinu's Policy

- What happens to **currently executing process** during a context switch? – Often, it **remains eligible to use CPU**, even when control is temporarily passed to another process (e.g., as a result of round-robin scheduling). Then 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` (see page 69) or in `kill` (see page 71).
- When `resched` prepares to **switch context**, it checks `pstate`-field of current process and makes it ready for execution by CPU, only if its `pstate` is `PCURRENT`.

Selecting a Ready Process: `resched`

The XINU scheduler `resched`

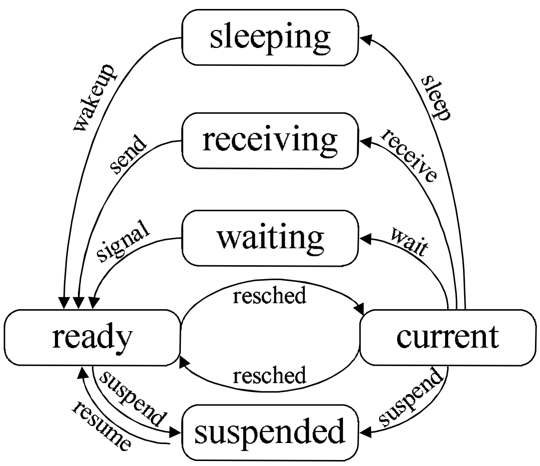
- selects a process to run
- changes process table for that entry
- removes new process from ready list and makes it current
- updates `CURRPID`
- resets preemption counter (to give the new process maximal amount of slot time to run, see pages 73 and 125 – it's set to `QUANTUM` = 10)
- calls `ctxsw(optr <- pregs, nptr <- pregs)` to save and reset the machine registers

Selecting a Ready Process: resched

```
/*------------------------------------------------------------------------
 * 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 PRREADY.
 *------------------------------------------------------------------------
 */
int   resched()
{
        register struct pentry  *optr;   /* pointer to old process entry */
        register struct pentry  *nptr;   /* pointer to new process entry */

        /* 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 = PRREADY;
              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;               /* reset preemption counter     */
#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 unter XINU **on this exact moment**. See process state diagram on pg. 128:



`sleep10(n)`	pg. 129
`wakeup()`	pg. 133
`receive()`	pg. 97
`send(...,...)`	pg. 96
`wait(sem)`	pg. 85
`signal(sem)`	pg. 86
`sdelete(sem)`	pg. 89
`suspend(pid)`	pg. 69
`resume(pid)`	pg. 67
`kill(pid)`	pg. 71

- So `resched()` is a **normal procedure**, and calling it results in executing it **like a normal procedure**.

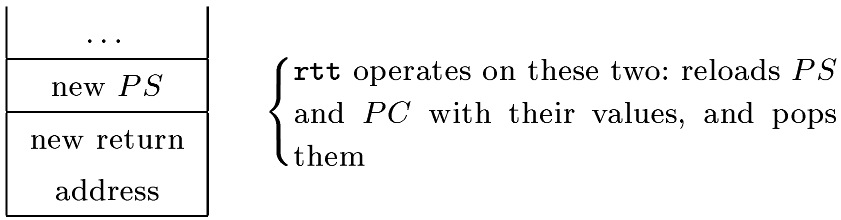
- However, `ctxsw` is not a normal procedure call in that after executing it one doesn't return (immediately) to the next statement in `resched()`.

```
int resched() {
...
  ctxsw(optr->pregs,nptr->pregs);
  return(OK);
}
```

- **So when is the return address in `resched()` reached?**

Context Switching: `ctxsw`

- The 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 is stored in `proctab` and stack.
- On the 11/2 the `rtt` instruction pops **both** *PS* **and** *PC* **from the stack**, and reload them **in one step**:



- So after saving registers 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, and, after having pushed **new values of** *PS* **and** *PC* **on stack** it resets the *PS* and *PC* in one go by calling `rtt`.

Context Switching: `ctxsw`

- Convention for calling procedures (see pg. 37):

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

- So when calling `ctxsw(optr<-pregs,nptr<-pregs)` (i.e. after `jsr pc, _ctxsw` was called in translation of `resched()`) the stack looks like this:

Stack		
...		
Address of save area with **new** registers and PS	$\leftarrow$	$SP+4$
Address of save area with **old** registers and PS	$\leftarrow$	$SP+2$
return address of `return(OK);`	$\leftarrow$	SP

Context Switching: `ctxsw`

All values of the **nine registers** ($R_0, \ldots, R_5, SP, PC, PS$) **of the old process are saved**:

- $R_0 :=$ `*`$2(SP)$ – where `*`x is the address of the area that x points to, i.e., the value of R_0 is stored in R_0-position in **old register-save-area**.
- $R_0 := 2(SP)$; $R_0 := R_0 + 2$ – so R_0 points to the R_1-position in old register-save-area.
- R_1 to R_5 are saved in old register-save-area.
- $SP := SP + 2$ – **after return from hibernation** stack pointer should point to stack segment with **parameters of `ctxsw`** since these are popped in calling procedure's translation. So the stack pointer is now **prepared** for later return from hibernation, i.e. SP no longer points to return address of `return(OK)` but to address of `ctxsw`'s arguments.
- Now the prepared stack pointer is saved; the return address of `return(OK)` (now contents of $SP - 2$, this leads to `-(sp)`) is put to old register-save-area; finally `mfps` puts the processor status word PS to old register-save-area.

$\Rightarrow$ Now the values of SP, PC and PS we need when **returning to `return(OK)`** are all **assembled in the old register-save-area**!

Context Switching: `ctxsw`

The result of saving all nine registers:

…		
R_0 ⋮ R_5		
old $SP + 2$	←	saved SP
contents of old SP	←	saved PC
old PS	←	saved PS

Now the **registers of the new process are restored**:

- $R_0 := SP + 4$ – so R_0 holds address of **new register-save-area**
- Now the values of R_1 to R_5 are restored.
- SP is restored (now we have actually switched stacks)
- PS and PC are loaded from new register-save-area and pushed onto the new stack
- R_0 is restored from new register-save-area
- **`rtt`** – this instructions pops PS and PC of the new stack (and thereby SP is reset correctly) all at once!

– only now context switch has taken place –

It is interesting to note that alle processes call **`resched()`** to perform context switching, and **`resched()`** calls `ctxsw`, so all suspended processes resume at the same place – just after the call to `ctxsw`. However, each process has its own stack of procedure calls, so the return from **`resched()`** will take them in various directions.

Context Switching: `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 consists of: the 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. of R1
      mov   r1,(r0)+            / Save registers R1-R5 in successive
      mov   r2,(r0)+            /  locations of the old process
      mov   r3,(r0)+            /  register save area.  (r0)+ denotes
      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 area
      mov   4(r0),r2            /   for the new process.
      mov   6.(r0),r3           / NOTE: dot following a number makes it
      mov   8.(r0),r4           /   decimal; all others are octal
      mov   10.(r0),r5
      mov   12.(r0),sp          / Have now actually switched stacks
      mov   16.(r0),-(sp)       / Push new process PS on new process stack
      mov   14.(r0),-(sp)       / Push new process PC on new process stack
      mov   (r0),r0             / Finally, load R0 from new area
      rtt                       / Load PC, PS, and reset SP all at once
```

4.4 Null Process

- `resched()` resumes that at least one process is available on the ready queue – it does not bother to verify whether ready list is empty, i.e.:
  `resched()` can only switch 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; its code contains of an infinite loop.

- Because user processes have all priority greater than 0, the scheduler switches to the null process **only when no user process ready to run** – so the null process is currently running when other processes **deadlock**. But even then the **real time clock interrupt routine** decreases real time clock until new process, if any, get ready to execute when their time expires. These are then put on ready queue, after which `resched()` is called.

4.5 Making A Process Ready

- Making a process **eligible for CPU service** occurs so often that a special procedure has been designed to do so: `ready(pid,resched)`.
- In principle putting a process on the ready list should result in a **call to `resched()`** 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 (for instance see process wakeup in realtime interrupt processing). Then all these processes are put **in one go** on the ready list **without rescheduling** after which `resched()` is called **once** to make sure that the process with the highest priority is running.
- This construction is made possible by providing `ready(...,...)` with a **boolean** argument `RESCHEDNO`/`RESCHEDYES` determining whether `resched()` is called.

Making A Process Ready

```
/* ready.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 struct pentry  *pptr;

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