10 Real-Time Clock Management

10.1 Kinds of clocks associated with a computer

- Three kinds of clocks are associated with a computer:
  1. A **central system clock**, controlling the rate the CPU executes instructions.
  2. A **real-time clock**, pulsing regularly an integral number of times each second, signalling the CPU each time a pulse occurs by posting an interrupt.
  3. A **time-of-day clock** like a wristwatch; the CPU controls this clock.
- Unlike the time-of-day clock, a real-time clock does not contain a counter, does not accumulate interrupts (left to the system), and controls the CPU.
- Responsibility for counting interrupts falls upon the system:

  If the CPU takes too long to service a real-time clock interrupt, or if it operates for more than one clock cycle with interrupts disabled, it will **miss the next real-time interrupt**, consequently:

  ∗ Systems should **service clock interrupts quickly**.

  ∗ Clock interrupts have **highest priority**.

10.2 Slowing down of the clock rate

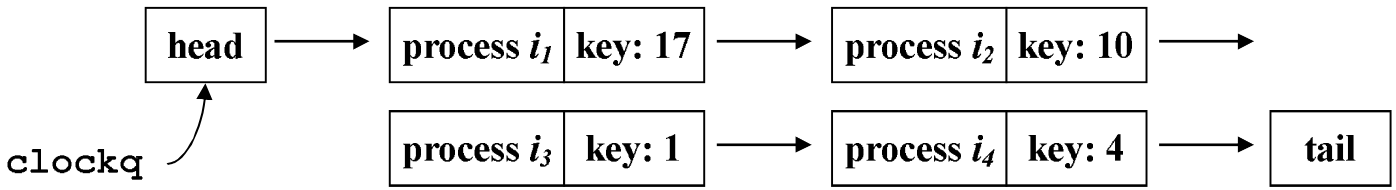
- Even so, the 11/2 processor cannot call a C-procedure on each clock interrupt, or it would spend **most of the time handling them**. So, the **clock rate must be slowed down**.
- How can the clock rate be „effectively " adjusted to prevent the system spending most of the time **handling interrupts**?
  - The clock interrupt handler/dispatcher `clkint` simulates a **slower-rate clock** by dividing the clock rate.
  - The LSI 11/2 real-time clock generates 60 pulses per second. `Clkint` **ignores** 5 clock interrupts in a row before (very quickly!) processing the 6^{th} one.
  - This reduces the **effective clock rate** to **10 pulses per second**, the so-called **tick rate**.
- The **tick rate** is the clock rate assumed by all processes in this chapter.

10.3 The use of real-time clock

- OS's use real-time clocks to compute the **time-slices allotted for the execution of each process**, by scheduling a preemption event.
- The OS also uses the real-time clock to provide processes with **timed delays**: The system maintains a list of processes, ordered by the time they should be **awakened**. When the real-time clock interrupts, it examines this list and wakes up processes for which the **delay has expired**.
- A preemption event is used to prevent processes from running forever. It is set in `resched` by `preempt := QUANTUM`.
- The clock interrupt dispatcher `clkint` **decrements preempt on each tick**, calling `resched` when preempt equals 0.
- `QUANTUM` shouldn't be **too small** (e.g. 2 or 3), causing too much overhead, and not **too large**, slowing down the reaction time. In practice most processes are stopped earlier that `QUANTUM` ticks, by executing `wait` or doing I/O processing.

10.4 Delta List Processing

- To avoid searching through lists, delayed processes are put on the **delta list**, residing in the q-structure. Variable `clockq` contains the q-index of its head.
- At each clock tick, `clkint` examines the **first process** in `clockq`, and calls a high-level interrupt routine `wakeup` to **awake processes when their delay has expired**.
- Processes on the `clockq` are **ordered** by the time at which they should be awakened, with each key telling the number of clock ticks which the process must delay beyond the preceding one on the `clockq`.
- E.g. 4 processes, needing to delay 17, 27, 28 and 32 ticks are ordered in the delta list as follows:



- Procedure `insertd(pid, head, key)` inserts process `pid` in delta-list `head`, given its key `key`.

Delta List Processing

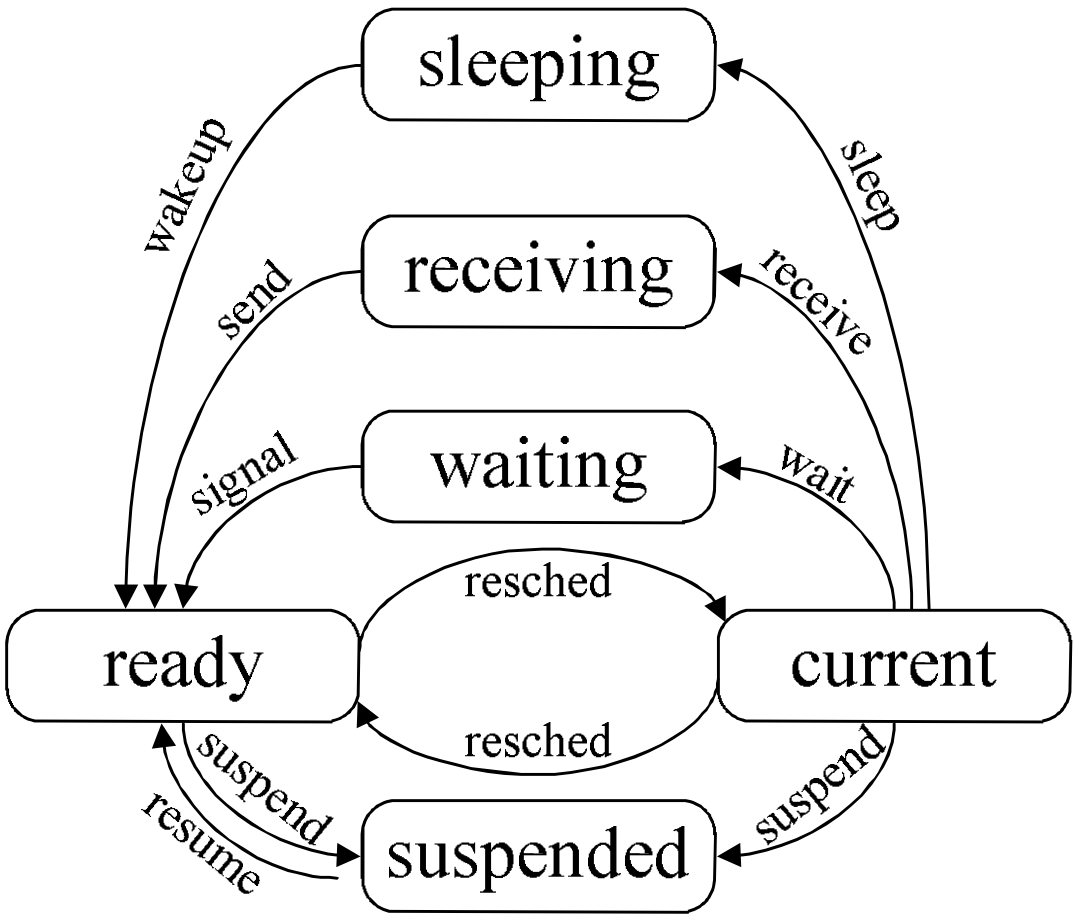
```
/* insertd.c - insertd */

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

/*------------------------------------------------------------------------
 *  insertd  --  insert process pid in delta list "head", given its key
 *------------------------------------------------------------------------
 */
insertd(pid, head, key)
        int     pid;
        int     head;
        int     key;
{
        int     next;                   /* runs through list            */
        int     prev;                   /* follows next through list    */

        for(prev=head,next=q[head].qnext ;
            q[next].qkey < key ; prev=next,next=q[next].qnext)
                key -= q[next].qkey;
        q[pid].qnext = next;
        q[pid].qprev = prev;
        q[pid].qkey  = key;
        q[prev].qnext = pid;
        q[next].qprev = pid;
        if (next < NPROC)
                q[next].qkey -= key;
        return(OK);
}
```

10.5 Putting a process to sleep

- System call `sleep(`n`)` delays calling process (i.e. the current one) for n tenths-of-a-second, by moving the current process to the delta list `clockq`.
- This requires introducing a new process state for that moved process: `SLEEPING`. System variables introduced on page 129:
  - `slnempty`: sleeping list is non-empty
  - `clkruns`: if greater than 0 the hardware includes a real-time clock
  - `sltop`: address of the first key on clock list

  Notice that this **does not concern hard real-time**!
- See page 130: `setclkr` sets `clkruns` to 1 if real-time clock exists, otherwise to 0. It simulates an interrupt mechanism whose only purpose is to see **whether the real-time clock is present**, i.e., runs.

New process state `SLEEPING`



Putting a process to sleep

```
/* sleep10.c - sleep10 */

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

/*------------------------------------------------------------------------
 * sleep10  --  delay the caller for a time specified in tenths of seconds
 *------------------------------------------------------------------------
 */
SYSCALL sleep10(n)
        int n;
{
        char ps;

        if (n < 0  || clkruns==0)
                return(SYSERR);
        disable(ps);
        if (n == 0) {                   /* sleep10(0) -> end time slice */
                ;
        } else {
                insertd(currpid,clockq,n);
                slnempty = TRUE;
                sltop = (int *) & q[q[clockq].qnext].qkey;
                proctab[currpid].pstate = PRSLEEP;
        }
        resched();
        restore(ps);
        return(OK);
}
```

Putting a process to sleep

```
/* sleep.h */

#define CVECTOR 0100            /* location of clock interrupt vector   */

extern  int     clkruns;        /* 1 iff clock exists; 0 otherwise      */
                                /* Set at system startup.               */
extern  int     clockq;         /* q index of sleeping process list     */
extern  int     count6;         /* used to ignore 5 of 6 interrupts     */
extern  int     *sltop;         /* address of first key on clockq       */
extern  int     slnempty;       /* 1 iff clockq is nonempty             */

extern  int     defclk;         /* >0 iff clock interrupts are deferred */
extern  int     clkdiff;        /* number of clock ticks deferred       */
extern  int     clkint();       /* clock interrupt handler              */
```

10.6 Delays measured in seconds

- Size of integer n – 16 bits – **limits** the delay allowed by calling `sleep10(`n`)` to $2^{15} - 1$ tenths of a second $= 55$ minutes.
- System call `sleep(`n`)` measures delays in **seconds** and so allows delaying **up to 9 hours** – see the code in `sleep.c`:

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

/*------------------------------------------------------------
 * sleep  --  delay the calling process n seconds
 *------------------------------------------------------------
 */
SYSCALL sleep(n)
        int     n;
{
        char    ps;

        if (n<0 || clkruns==0)
                return(SYSERR);
        if (n == 0) {
                disable(ps);
                resched();
                restore(ps);
                return(OK);
        }
        while (n >= 1000) {
                sleep10(10000);
                n -= 1000;
        }
        if (n > 0)
                sleep10(10*n);
        return(OK);
}
```

10.7 Awaking sleeping processes

- Read the code for `clkint` interrupt dispatcher on page 136 to see:
  - that `clkint` decrements the count of the first key on `clockq` at each tick, until his equals 0, and
  - the high-level interrupt procedure `wakeup()` is called, to put a process with key = 0 on the ready list.
- `wakeup()` assumes that interrupts have been disabled upon entry, as signalled by keyword `INTPROC`:

```
/* wakeup.c - wakeup */

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

/*------------------------------------------------------------------------
 * wakeup  --  called by clock interrupt dispatcher to awaken processes
 *------------------------------------------------------------------------
 */
INTPROC wakeup()
{
        while (nonempty(clockq) && firstkey(clockq) <= 0)
                ready(getfirst(clockq),RESCHNO);
        if ( slnempty = nonempty(clockq) )
                sltop = (int *) & q[q[clockq].qnext].qkey;
        resched();
}
```

10.8 Deferred clock processing

- The **deferred mode** allows the system to accumulate clock ticks in variable `clkdiff` **without initiating events**.
- I.e., it **postpones the reaction** upon real-time interrupts. The clock handler can schedule events that **should have occurred** to the point where it leaves deferred mode and returns to **original mode**.
- A process can place the clock in deferred mode by calling `stopclk`, and return clock to real time mode by calling `strtclk`:
  - `Stopclk` **counts deferral requests** by incrementing `defclk`.
  - `strtclk` counts „restart “ requests by decrementing `defclk`.
  - As long as `defclk` remains **positive** the interrupt handler counts clock times in `clkdiff` **without processing them**.
- `strclk` „makes up for lost time “ when `defclk` equals 0, by catching up on all events that **should have occurred** while the clock remained deferred: `srtclk` **updates the preemption counter** and abstracts the **accumulated ticks** from the delay of sleeping processes.

Deferred clock processing

```
/* ssclock.c - stopclk, strtclk */

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

/*------------------------------------------------------------------------
 *  stopclk  --  put the clock in defer mode
 *------------------------------------------------------------------------
 */
stopclk()
{
        defclk++;
}
```

Deferred clock processing

`ssclock.c` continued:

```
/*------------------------------------------------------------------------
 *  strtclk  --  take the clock out of defer mode
 *------------------------------------------------------------------------
 */
strtclk()
{
        char ps;
        int makeup;
        int next;

        disable(ps);
        if ( defclk<=0 || --defclk>0 ) {
                restore(ps);
                return;
        }
        makeup = clkdiff;
        preempt -= makeup;
        clkdiff = 0;
        if ( slnempty ) {
                for (next=firstid(clockq) ;
                    next < NPROC && q[next].qkey < makeup ;
                    next=q[next].qnext) {
                        makeup -= q[next].qkey;
                        q[next].qkey = 0;
                }
                if (next < NPROC)
                        q[next].qkey -= makeup;
                wakeup();
        }
        if ( preempt <= 0 )
                resched();
        restore(ps);
}
```

10.9 Clock interrupt processing

- The **assembly language routine** dispatching clock interrupts is called `clkint`. Read it **carefully**!

```
/* clkint.s -  clkint */

/*------------------------------------------------------------------------- /*
clkint  --  real-time clock interrupt service routine
/*-------------------------------------------------------------------------
        .globl  _clkint
_clkint:
        dec     _count6                 / Is this the 6th interrupt?
        bgt     clret                   /  no => return
        mov     $6,_count6              /  yes=> reset counter&continue
        tst     _defclk                 / Are clock ticks deferred?
        beq     notdef                  /  no => go process this tick
        inc     _clkdiff                /  yes=> count in clkdiff and
        rtt                             /       return quickly
notdef:
        tst     _slnempty               / Is sleep queue nonempty?
        beq     clpreem                 /  no => go process preemption
        dec     *_sltop                 /  yes=> decrement delta key
        bgt     clpreem                 /       on first process,
        mov     r0,-(sp)                /       calling wakeup if
        mov     r1,-(sp)                /       it reaches zero
        jsr     pc,_wakeup              /       (interrupt routine
        mov     (sp)+,r1                /        saves & restores r0
        mov     (sp)+,r0                /        and r1; C doesn't)
clpreem:
        dec     _preempt                / Decrement preemption counter
        bgt     clret                   /   and call resched if it
        mov     r0,-(sp)                /   reaches zero
        mov     r1,-(sp)                /      (As before, interrupt
        jsr     pc,_resched             /       routine must save &
        mov     (sp)+,r1                /       restore r0 and r1
        mov     (sp)+,r0                /       because C doesn't)
clret:
        rtt                             / Return from interrupt
```

10.10 Clock initialization

- The clock interrupt vector should be **initialized** at system start-up, **before clock interrupts occur**. This is done by calling `clkinit()`.
- Variable `CVECTOR` ($= 0100_8 = 64_{10}$) is defined in `sleep.h` on an earlier slide. Recall: `setclkr()` initializes `clkruns`. Now read `setclkr()`:
  - Observe in setloop: hardware clock R_1 decreased at tact of hardware clock!
  - If after 30.000 ticks of the hardware clock no real-time interrupt occurred, `clkruns` remains zero. If clock interrupt occurs, `setclkr`'s control jumps to `setint`: `clkruns` > 0.
  - `setclkr` returns, after restoring the original real-time clock vector, to the next instruction pointed at by PC.

Clock initialization

```
/* clkinit.c - clkinit */

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

/*
 *------------------------------------------------------------------------
 * clkinit - initialize the clock and sleep queue (called at startup)
 *------------------------------------------------------------------------
 */
clkinit()
{
        int *vector;

        vector = (int *) CVECTOR;       /* set up interrupt vector      */
        *vector++ = clkint;
        *vector = DISABLE;
        setclkr();
        preempt = QUANTUM;              /* initial time quantum         */
        count6 = 6;                     /* 60ths of a sec. counter      */
        slnempty = FALSE;               /* initially, no process asleep */
        clkdiff = 0;                    /* zero deferred ticks          */
        defclk = 0;                     /* clock is not deferred        */
        clockq = newqueue();            /* allocate clock queue in q    */
}
```

Clock initialization

```
/* setclkr.s - setclkr */

CVECTPC =       100                     / clock interrupt vector address
CVECTPS =       102                     /  "       "         "      "
DISABLE =       340                     / PS that disables interrupts
ENABLE  =       000                     / PS that enables interrupts
COUNT   =       30000.                  / Times to loop (in decimal)

/*------------------------------------------------------------------------
/* setclkr  --  set cklruns to 1 iff real-time clock exists, 0 otherwise
/*------------------------------------------------------------------------
        .globl  _setclkr
_setclkr:
        mov     r1,-(sp)                / save register used
        clr     _clkruns                / initialize for no clock
        mov     *$CVECTPS,-(sp)         / save clock interrupt vector
        mov     *$CVECTPC,-(sp)         /   on caller's stack
        mov     $DISABLE,*$CVECTPS      / set up new interrupt vector
        mov     $setint,*$CVECTPC
        mov     $COUNT,r1               / initialize counter for loop
        reset                           / clear other interrupts, if any
        mtps    $ENABLE                 / allow interrupts
setloop:
        dec     r1                      / loop COUNT times waiting for
        bpl     setloop                 /   a clock interrupt
        mtps    $DISABLE                / no interrupt occurred, so quit
        br      setdone
setint:
        inc     _clkruns                / clock interrupt jumps here
        add     $4,sp                   / pop pc/ps pushed by interrupt
setdone:
        mov     (sp)+,*$CVECTPC         / restore old interrupt vector
        mov     (sp)+,*$CVECTPS
        mov     (sp)+,r1                / restore register
        rts     pc                      / return to caller
```