Operating Systems

Xinu – Clocks and Timers

1

Timed Events

- Many systems and applications need to make use of timed events
  - network protocols, user interfaces, interfaces to external devices
- Many systems are structured around an asynchronous event paradigm
  - programmer defines a set of handler functions that are invoked based on a given event
  - arguably the OS functions like this internally
- Other systems provide a synchronous approach
  - The Xinu system provides delay and the programmer creates processes to schedule events

2

Clocks and Timers

- Systems contain various sorts of clocks and timers
  - Processor clock – regular pulses to drive processor operation
  - Real-time clock – independent of processor operation, pulses in some fraction of a second and generates an interrupt for each pulse
  - Time of day clock – once set (or reset), computes elapsed time, does not interrupt
  - Interval timer – real time clock and counter that is modified per pulse, interrupts when a target is reached

3

Timers and Interrupts

- A real time clock interrupts regularly
  - Can have significant overhead but may be required for real time applications
- An interval timer interrupts after a specified delay
  - Can be used to emulate a real time clock
  - Can vary the granularity of clock ticks
  - Set timer and decrement until zero, interrupt
  - Reset the timer when the interrupt fires

4

Hardware

- The BeagleBone has an interval timer
  - The Xinu code configures the interval timer to interrupt after 1 millisecond
  - The interval timer can be configured to reset itself when an interrupt occurs
- The Galileo has a real-time clock
  - Configured at startup to interrupt every millisecond
- This provides consistent behavior on both platforms, with regularly occurring (1ms) timer interrupts

Regular (Real-time) Timer Interrupts

- A real-time clock interrupts regularly without accumulating interrupts
  - When using a timer to emulate a clock, the same is true
- Clock-related interrupts must be serviced quickly or subsequent interrupts can be lost
- The processor must be able to execute many instructions between timer interrupts and thus must operate faster then the clock
  - How many cycles between interrupts at 1 GHz?
- Preference is given to the clock interrupt over I/O device interrupts

Timed Delay

- The OS does two main things with time
  - timed delay
  - preemption
- The timed delay mechanism allows a process to sleep for a specified amount of time
- Sleeping removes a process from the ready queue and restores it to the ready queue after the specified amount of time
  - Once back in the ready queue, it executes according to the scheduling policy

Preemption

- The process manager uses preemption to return control to the operating system
  - Without preemption, an errant process can disrupt the system in an infinite loop
- Time slicing – the OS may switch to another runnable process once the running process has used its share of time
- When there are several processes of the same priority, a call to resched() will place the current process at the end of the set in the ready list, and switches to the first process on the list
- Thus, processes of the same priority get scheduled round-robin

Time Slicing

- Define a maximum time that a process may execute without allowing other processes to execute – a time *quantum*
- Short time slice values enable more sharing, but increase overhead
- All processes could potentially execute for their entire time slice, but generally few processes do, but rather get descheduled due to e.g. I/O

Preemption Implementation

- *QUANTUM* specifies the number of clock ticks in a timeslice
- When a process is scheduled, *preempt* is set to *QUANTUM*
- When the clock tick occurs, *preempt* is decremented
- When it reaches 0, reset *preempt* to *QUANTUM* and call *resched()*
  - Two possibilities for resetting *preempt*

Delay for Processes

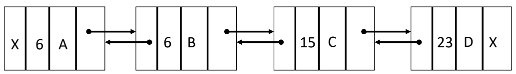
- Maintain a list of processes that have requested delay
- Xinu uses a "delta list" to avoid searching the list for processes to wake
- A list in the queuetab ordered by time at which the process should be made ready to run again
- Rather than keeping absolute times, the list simply stores the difference between process wakeup times
- Only the first item of the list needs to be updated at each clock tick

Delta List

- The key of the first process on a delta list specifies the number of clock ticks a process must delay beyond the current time; the key of each other process on a delta list specifies the number of clock ticks the process must delay beyond the preceding process on the list.

Delta List

- Processes A, B, C and D requesting delays of 6, 12, 27 and 50 ticks (respectively) at roughly the same time (within one clock tick)
- These result in a delta list like this:



13

Delta List Implementation

- The global variable *sleepq* points to the queue ID of the delta list
- When the clock ticks, the clock interrupt handler decrements the key of the first item in the list (if non-empty)
- When the key == 0, it is time for the process to be awakened
- The hander calls function wakeup()

14

Delta List Implementation

- To maintain the list, insertd() takes a PID, Queue ID, and a delay in the argument key
- The key can be directly compared to the first key in the list
- But not to successive nodes as they specify delays relative to their predecessor
- insertd subtracts the relative delays from key at each step so that it is comparable to queuetab[next].qkey
- Must also decrement the next key in the list by the key value being inserted

15

insertd.c

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

#include <xinu.h>

/*------------------------------------------------------------------------
 *  insertd  -  Insert a process in delta list using delay as the key
 *------------------------------------------------------------------------ */

status  insertd(                        /* Assumes interrupts disabled  */
    pid32         pid,             /* ID of process to insert      */                 qid16
q,               /* ID of queue to use           */                int32         key
/* Delay from "now" (in ms.)    */
        ){

    int32   next;                   /* Runs through the delta list  */                int32
prev;                     /* Follows next through the list*/

    if (isbadqid(q) || isbadpid(pid)) {
        return SYSERR;
    }

    prev = queuehead(q);
    next = queuetab[queuehead(q)].qnext;
```

16

insertd.c

```
        while ((next != queuetail(q)) && (queuetab[next].qkey <= key)) {
                key -= queuetab[next].qkey;
                prev = next;
                next = queuetab[next].qnext;
        }

        /* Insert new node between prev and next nodes */

        queuetab[pid].qnext = next;
        queuetab[pid].qprev = prev;
        queuetab[pid].qkey = key;
        queuetab[prev].qnext = pid;
        queuetab[next].qprev = pid;

        if (next != queuetail(q)) {
            queuetab[next].qkey -= key;
        }


        return OK;
}
```

17

Putting a Process to Sleep

- Processes use the sleep() or sleepms() calls
  - sleep() is in seconds
  - sleepms() is in milliseconds
- Single implementation in which sleep() multiplies argument and calls sleepms()
  - Matches the clock interrupt
- Sleeping is a distinct state, PR_SLEEP

18

sleep.c

```
/* sleep.c - sleep sleepms */

#include <xinu.h>

#define MAXSECONDS  4294967   /* Max seconds per 32-bit msec  */

/*------------------------------------------------------------------------
 *  sleep  -  Delay the calling process n seconds
 *------------------------------------------------------------------------
 */
syscall sleep(
    uint32  delay   /* Time to delay in seconds */
  )
{
  if (delay > MAXSECONDS) {
    return SYSERR;
  }
  sleepms(1000*delay);
  return OK;
}
```

19

sleep.c

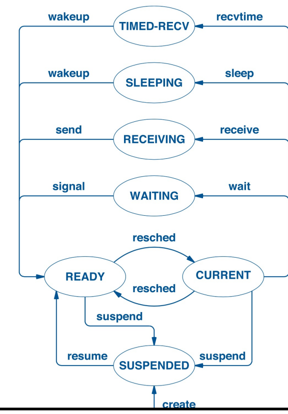
```
/*------------------------------------------------------------------------
 *  sleepms  -  Delay the calling process n milliseconds
 *------------------------------------------------------------------------
 */
syscall sleepms(
    uint32  delay   /* Time to delay in msec. */
  )
{
  intmask mask;     /* Saved interrupt mask   */

  mask = disable();
  if (delay == 0) {
    yield();
    restore(mask);
    return OK;
  }

  /* Delay calling process */

  if (insertd(currpid, sleepq, delay) == SYSERR) {
    restore(mask);
    return SYSERR;
  }

  proctab[currpid].prstate = PR_SLEEP;
  resched();
  restore(mask);
  return OK;
}
```

20

Timed Message Receive

- Useful in networked applications and network protocols
  – Wake for lost message recovery
- Block for a message for some time and then unblock
  – Receive() OR Sleep(x)
- Place process in the sleep queue but in state TIMED-RECV
- If a message arrives, remove the process from the sleep queue
  – Detected by the send() call, which calls unsleep()

Complete State Diagram



unsleep.c

```
/* unsleep.c - unsleep */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  unsleep  -  Internal function to remove a process from the sleep
 *        queue prematurely.  The caller must adjust the delay
 *        of successive processes.
 *------------------------------------------------------------------------
 */
status  unsleep(
    pid32   pid   /* ID of process to remove  */
        )
{
  intmask mask;     /* Saved interrupt mask    */
        struct  procent *prptr;   /* Ptr to process' table entry  */

        pid32 pidnext;    /* ID of process on sleep queue */
          /*   that follows the process */
          /*   which is being removed */

  mask = disable();

  if (isbadpid(pid)) {
    restore(mask);
    return SYSERR;
  }

  /* Verify that candidate process is on the sleep queue */
```


unsleep.c

```
  /* Verify that candidate process is on the sleep queue */

  prptr = &proctab[pid];
  if ((prptr->prstate!=PR_SLEEP) && (prptr->prstate!=PR_RECTIM)) {
    restore(mask);
    return SYSERR;
  }

  /* Increment delay of next process if such a process exists */

  pidnext = queuetab[pid].qnext;
  if (pidnext < NPROC) {
    queuetab[pidnext].qkey += queuetab[pid].qkey;
  }

  getitem(pid);     /* Unlink process from queue */
  restore(mask);
  return OK;
}
```

wakeup.c

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

#include <xinu.h>

/*------------------------------------------------------------------------
 *  wakeup  -  Called by clock interrupt handler to awaken processes
 *------------------------------------------------------------------------
 */
void  wakeup(void)
{
  /* Awaken all processes that have no more time to sleep */

  resched_cntl(DEFER_START);
  while (nonempty(sleepq) && (firstkey(sleepq) <= 0)) {
    ready(dequeue(sleepq));
  }

  resched_cntl(DEFER_STOP);
  return;
}
```

25

clkhandler.c

```c
/* clkhandler.c - clkhandler */

#include <xinu.h>

/*-----------------------------------------------------------------------
 * clkhandler - high level clock interrupt handler
 *-----------------------------------------------------------------------
 */
void  clkhandler()
{

  static uint32 count1000 = 1000; /* variable to count 1000ms */
  volatile struct am335x_timer1ms *csrptr = 0x44E31000;
          /* Pointer to timer CSR     */

  /* If there is no interrupt, return */

  if((csrptr->tisr & AM335X_TIMER1MS_TISR_OVF_IT_FLAG) == 0) {
    return;
  }

  /* Acknowledge the interrupt */

  csrptr->tisr = AM335X_TIMER1MS_TISR_OVF_IT_FLAG;

  /* Decrement 1000ms counter */

  count1000--;
```

26

clkhandler.c

```c
  /* After 1 sec, increment clktime */

  if(count1000 == 0) {
    clktime++;
    count1000 = 1000;
  }

  /* check if sleep queue is empty */

  if(!isempty(sleepq)) {

    /* sleepq nonempty, decrement the key of */
    /* topmost process on sleepq     */

    if((--queuetab[firstid(sleepq)].qkey) == 0) {

      wakeup();
    }
  }

  /* Decrement the preemption counter */
  /* Reschedule if necessary      */

  if((--preempt) == 0) {
    preempt = QUANTUM;
    resched();
  }
}
```

27

resched.c

```c
/* resched.c - resched, resched_cntl */

#include <xinu.h>

struct  defer Defer;

/*------------------------------------------------------------------------
 *  resched  -  Reschedule processor to highest priority eligible process
 *------------------------------------------------------------------------
 */
void  resched(void)   /* Assumes interrupts are disabled  */
{
  struct procent *ptold;  /* Ptr to table entry for old process */
  struct procent *ptnew;  /* Ptr to table entry for new process */

  /* If rescheduling is deferred, record attempt and return */

  if (Defer.ndefers > 0) {
    Defer.attempt = TRUE;
    return;
  }

  /* Point to process table entry for the current (old) process */

  ptold = &proctab[currpid];
```

28

resched.c

```
    if (ptold->prstate == PR_CURR) {  /* Process remains eligible */
      if (ptold->prprio > firstkey(readylist)) {
        return;
      }

      /* Old process will no longer remain current */

      ptold->prstate = PR_READY;
      insert(currpid, readylist, ptold->prprio);
    }

    /* Force context switch to highest priority ready process */

    currpid = dequeue(readylist);
    ptnew = &proctab[currpid];
    ptnew->prstate = PR_CURR;
    preempt = QUANTUM;     /* Reset time slice for process */
    ctxsw(&ptold->prstkptr, &ptnew->prstkptr);

    /* Old process returns here when resumed */

    return;
  }
```


clkinit.c

```
/* Set interrupt vector for clock to invoke clkhandler */

set_evec(AM335X_TIMER1MS_IRQ, (uint32)clkhandler);

sleepq = newqueue();  /* Allocate a queue to hold the delta */
      /*   list of sleeping processes   */

preempt = QUANTUM;  /* Set the preemption time    */

clktime = 0;    /* Start counting seconds   */
```


Summary

- Clock interrupts are critical to preemptive operating systems
- Timers exist in various forms but generally OSes use regular interrupts and construct timed events with software