Real-Time Systems Programming

Summer-Semester 2002
Lecture 22
28 June 2002

Scheduling

Overview

- ***Goal***
  - To understand the role that scheduling and schedulability analysis plays in verifying that real-time applications meet their deadlines
- ***Topics***
  - Scheduling in the context of real-time systems
  - Rate and response requirements
  - Scheduling facilities under ***UNIX***
  - Simple process model
  - The cyclic executive approach
  - Process-based scheduling

Scheduling of Real-Time Systems

- Typical RT system: Concurrent tasks (processes, threads, fibres)
- ***What*** results are produced should not depend on ***schedule***
- However, a RT-system has to meet certain ***deadlines*** – and whether these deadlines are met or not
- However, ***when*** the results are produced (and whether deadlines are met) may depend on schedule
- Furthermore, the functionality (what is produced) may also depend on when tasks execute
  - ***Example:*** throughput measurement

Scheduling Concerns

Generally, we want to do one or more of the following:

- Make sure A happens at or before time *t*
- Make sure A happens before B
- Make sure important job A is not delayed if that delay is not part of A
  - ***Example:*** Job B has caused A to be swapped out to disk
- Analyse the schedulability of a given task set

Rate and Response Requirements

- Performance requirements fall into two broad categories
  - ***Rate requirements:*** job A must run X times a second
  - ***Response requirements:*** if event B occurs, job C must complete within Y msecs
- Hardware I/O
  - Getting sensor inputs, controlling actuators
  - ***Example:*** on the Space Shuttle, the ***rate requirement*** on the engine control during ascent is 100 Hz
  - ***Example:*** a phase change interrupt may have a certain ***response requirement***

Rate and Response Requirements

- Data logging
  - Typically have a ***rate requirement*** on the data I/O
- User I/O
  - Humans are fairly slow and non-deterministic I/O devices
  - Typically, they can wait (while the other tasks keep the engines from blowing up)
  - However, faster is better
- Background computation tasks
  - Have to run at some time
  - However, no stringent deadlines
  - Partial solutions may be acceptable

Standard Scheduling under UNIX

- Standard UNIX runs a ***time-sharing*** scheduler whose behavior is undefined by any standard
- Crucial item for any time-sharing scheduler is to balance the needs of
  - interactive or I/O bound tasks and
  - compute-bound tasks
- Each process's scheduling priority is continuously adjusted depending on what the process is doing
  - The more a process waits, the higher gets its priority

Being `nice` *under UNIX*

- One additional parameter for assessing the current priority of a process is the ***nice*** value of each process
  - Default value is 0
  - A positive value reduces the priority
  - A negative value increases the priority
- *Nice* value is set with **`nice`** system call
- ***Example:*** **`% nice -20 make huge_job`**
- Can influence the ***average-case*** behavior of processes this way

UNIX `priocntl`

- UNIX System V introduced **`priocntl`**, which is better suited for RT-scheduling than nice
  - Also supported under ***Solaris***
  - ***Linux*** does not have it – yet (2.4.0 kernel)
- Can specify
  - Scheduling class of a process – "RT" or "TS"
  - Scheduling priority
  - Scheduling quantum (seconds and nanoseconds)
- If scheduling quantum is set to "infinity", the process will run in true FIFO-mode – that is, until it gets blocked or voluntarily releases the CPU

UNIX `priocntl` Example – Part I

```
#include <sys/types.h>
#include <sys/procset.h>
#include <sys/priocntl.h>
#include <sys/rtpriocntl.h>
#include <sys/tspriocntl.h>

#define VIRTUAL_PRIORITY 10

main()
{
  pcinfo_t   rt_class_info;
  rtinfo_t   *rtinfo;
  pcparms_t  my_rt_params;
  rtparms_t  *rtparms;
```

UNIX `priocntl` Example – Part II

```
  // Check whether priority level is valid
  strcpy(rt_class_info.pc_clname, "RT");
  priocntl(0, 0, PC_GETCID ; &rt_class_info);
  rtinfo = (rtinfo_t *) rt_class_info.pc_clinfo;
  if (rtinfo->rt_maxpri < VIRTUAL_PRIORITY) {
    fprintf(stderr, "Cannot run at RT prio %d: max is %d\n",
      VIRTUAL_PRIORITY, rtinfo->rt_maxpri);
    exit(1);
  }

  // Now set the process class and priority
  my_rt_params.pc_cid = rt_class_info.pc_cid;
  rt_parms = (rtparms_t *) my_rt_parms.pc_clparms;
  rtparms->rt_pri = VIRTUAL_PRIORITY;
  rtparms->rt_tqnsecs = RT_TQINF;  // Infinity – run FIFO
  rtparms->rt_tqsecs = 0;          // This is now ignored

  priocntl(P_PID, getpid(), PC_SETPARMS, &my_rt_parms);
}
```

Assessment of UNIX Scheduling

- **`nice`** is easy to use, ***but*** ineffective for real-time programming, as it only influences average case behavior
- **`priocntl`** is more powerful – ***but***:
  - It is rather complicated in its use – better suited for system administrators than for RT application designers
  - Portability is limited to SVR4 systems
- The ***POSIX*** scheduling interfaces, discussed later, give
  - a simple interface
  - good portability

Scheduling

- In general, a scheduling scheme consists of:
  - An algorithm for ***ordering*** the use of system resources (in particular the CPUs)
  - A means of ***predicting*** the worst-case behaviour of the system when the scheduling algorithm is applied

- The prediction can then be used to confirm the temporal requirements of the application

Simple Process Model

- The application is assumed to consist of a fixed set of processes
- All processes are ***periodic***, with known periods
- The processes are completely ***independent*** of each other
- All system's overheads, context-switching times and so on are ***ignored*** (i.e, assumed to have zero cost)
- All processes have a ***deadline equal to their period*** (that is, each process must complete before it is next released)
- All processes have a ***known worst-case execution time***

Cyclic Executives

- One common way of implementing hard real-time systems is to use a ***cyclic executive***
- Here the design is concurrent but the code is produced as a collection of procedures
- Procedures are mapped onto a set of ***minor*** cycles that together constitute the complete schedule (or ***major*** cycle)
- Minor cycle dictates the minimum cycle time
- Major cycle dictates the maximum cycle time
- ***Main advantage:*** system is ***fully deterministic***

Example of a Cyclic Executive

- A process set:

Process	Period, T	Computation Time, C
a	25	10
b	25	8
c	50	5
d	50	4
e	100	2

- A cyclic executive for this process set:

```
loop
  wait_for_interrupt;
  proc_a; proc_b; proc_c;
  wait_for_interrupt;
  proc_a; proc_b; proc_d; proc_e;
  wait_for_interrupt;
  proc_a; proc_b; proc_c;
  wait_for_interrupt;
  proc_a; proc_b; proc_d;
end loop;
```


- No actual processes exist at run-time; each minor cycle is just a sequence of procedure calls
- The procedures share a common address space and can thus pass data between themselves
  - This data does not need to be protected (via a semaphore, for example) because concurrent access is not possible
- All "*process*" periods must be a multiple of the minor cycle time

Properties of Cyclic Executive

- No actual processes exist at run-time; each minor cycle is just a sequence of procedure calls
- The procedures share a common address space and can thus pass data between themselves
  - This data does not need to be protected (via a semaphore, for example) because concurrent access is not possible
- All "*process*" periods must be a multiple of the minor cycle time

Problems with Cycle Executives

- It is difficult to incorporate processes with ***long periods***
  - The major cycle time is the maximum period that can be accommodated without secondary schedules
- ***Sporadic*** activities cannot be incorporated
- The cyclic executive is difficult to construct and difficult to maintain — ***it is an NP-hard problem***
- Any "process" with a sizable computation time will need to be split into a fixed number of fixed sized procedures
  - This may cut across the structure of the code from a software engineering perspective and may be error-prone
- More flexible scheduling methods difficult to support

Process-Based Scheduling

- *An alternative to the cyclic executive:*
  - Support processes directly
  - Determine which process should execute at any given time by using scheduling attributes
- Ignoring interprocess communication, a process is then in either one of the following states:
  - ***Runnable***
  - ***Suspended waiting for a timing event*** (periodic processes)
  - ***Suspended waiting for a non-timing event*** (sporadic processes)

Scheduling Approaches

- There are numerous scheduling approaches
  - Design and analysis an active research area
- However, the number of scheduling schemes found in current systems is still limited
- Here, we will consider
  - Fixed-Priority Scheduling (FPS)
  - Earliest Deadline First (EDF)
  - Value-Based Scheduling (VBS)

Fixed-Priority Scheduling (FPS)

- The most widely used approach and our main focus
- Each process has a fixed, ***static***, priority
  - Is computed pre-run-time
- Execution order of runnable processes determined by their priority
- ***Note:*** In real-time systems, the "priority" of a process is derived from its ***temporal requirements***, not its importance to the correct functioning of the system or its integrity
  - The more critical processes often have the more stringent timing requirements – but this is not necessarily the case

Earliest Deadline First (EDF) Scheduling

- The runnable processes are executed in the order determined by the absolute deadlines of the processes
- The next process to run being the one with the shortest (nearest) deadline
- Although it is usual to know the relative deadlines of each process (e.g. 25ms after release), the absolute deadlines are computed at run time and hence the scheme is classified as ***dynamic***

Value-Based Scheduling (VBS)

- If a system can become overloaded:
  - Use of simple static priorities or deadlines not sufficient
  - Need a more ***adaptive*** scheme
- One approach:
  - Assign a ***value*** to each process
  - On-line ***value-based scheduling algorithm*** decides which process to run next

Preemption and Non-Preemption

- With priority-based scheduling, a high-priority process may be released during the execution of a lower priority one
- In a ***preemptive*** scheme:
  - Immediate switch to the higher-priority process
- ***Non-preemption***:
  - Lower-priority process can complete before the other executes
- ***Deferred preemption (cooperative dispatching):***
  - Lower-priority process can at least execute for some time before being preempted
- EDF and VBS can take on a preemptive or non-preemptive form

Summary I

- Deterministic temporal behavior is critical for real-time systems; one important component to ensure this is ***scheduling***
- The timing requirements can be divided into ***rate*** and ***response*** requirements
- Standard UNIX provides
  - **`nice`**: easy to use, ill-suited for RT requirements
  - **`priocntl`**: more powerful, complicated usage

Summary II

- A scheduling scheme defines an algorithm for resource sharing and a means of predicting the worst-case behaviour of an application when that form of resource sharing is used.
- With a ***cyclic executive***, the application code must be packed into a fixed number of minor cycles such that the cyclic execution of the sequence of minor cycles (the major cycle) will enable all system deadlines to be met
- The cyclic executive approach has major drawbacks, many of which are solved by priority-based systems

To Go Further

- Chapter 13 of [Burns and Wellings 2001]
- Chapter 5 of Gallmeister, POSIX.4: Programming for the Real World, O'Really, 1995

Problem Set 11 – Due Mon, 8 July 2002

1) Create two programs that demonstrate the persistence of shared memory objects (SMOs) – prog1 creates an SMO and writes to it, prog2 then accesses the written data. What happens if you run prog1 multiple times? Modify prog1 such that it first unlinks the SMO before re-creating it **(3 pts)**
2) Pipes and FIFOs are means to pass data from one process to another. Write a program to characterize the bandwidth of pipes on your machine. What results do you get? **(3 pts)**
3) Modify the robot you built last week such that:

   The robot reads in an integer *x*, given as *bar code*; see also next page. **(4 pts)**

   Note: A quantitative success criterion is the max speed at which the given bar codes will be correctly read; bonus points will be assigned (in the discussion class) to the robot that meets this criterion best.

Have fun!

Problem Set 11 contd.

On the code to be used for the bar code reader:

- It is a simplified version of the EAN (European Article Number), on which further information can be found at http://www.tinohempel.de/info/mathe/ean/ean.htm.
- The ***resolution*** of the codes is ***R* = 5 mm**.
- We use a code that consists of one Start Delimiter (width 3*R*, see below for the encoding), followed by 4 decimal digits ***d1...d4***, concluded by an End Delimiter (width 4*R*).
- Each decimal digit is encoded as a sequence of 4 light/dark lines, according to EAN Code A (see below), with a total width of 7*R*.
- The total length of the bar code is thus (3 + 7 + 7 + 7 + 7 + 4)*R* = 35*R* = 17.5cm.
- The digits are interpreted as follows:
  - d1...d3 form an unsigned integer ***y*** ($0 \le y \le 999$).
  - *y* is mapped to signed integer ***x*** as $x = (y < 500)\ ?\ y : (y - 1000)$; thus, $-500 \le x \le 499$.
  - d4 is a ***parity digit***, computed as 9 – ((d1 + d2 + d3) mod 10). For example, 1233 would be a *valid* code word, whereas 0000 and 1234 would be invalid.
- Example bar codes are at http://www.informatik.uni-kiel.de/inf/von-Hanxleden/teaching/ss02/synch/homeworks/strichcodes.pdf