CS452/652 Real-Time Programming Course Notes

Daniel M. Berry, Cheriton School of Computer Science
University of Waterloo

What is a RT System?

A RT system (RTS) is a system that must respond to each external event within a finite and *specifiable* amount of time, called the event's *time limit*.

RT System, Cont'd

- hard RTS: a missed deadline ⇒ failure of the system

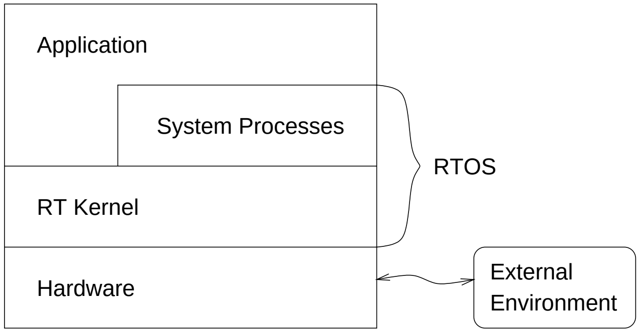
  e.g., air-traffic control: a missed deadline ⇒ people die
- soft RTS: a missed deadline costs in some way, but the system may be able to recover, and costs vary

  e.g., streaming video: a missed deadline ⇒ dropped frames and degredation of video and audio quality

Characteristics of a RT OS

- controls a set of devices,
- event driven,
- runs indefinitely, like `while (1) { … }`.
- bounded time responses
- *embedded* in a dedicated system comprising the computer and the devices

RT Software Structure


Application

System Processes

RTOS

RT Kernel

Hardware

External Environment

Processes

Each activity is abstracted as a *process* or *task*.

These notes and the literature use these two terms interchangeably.

Each process:

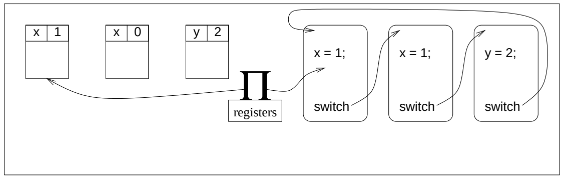
- communicates with other processes to control decision making and
- generates and receives external events.

Virtual CPUs

A CPU uses its registers to execute the instructions in a stored program using data in its environment, so that when the program says `x = 1;`, the cell for `x` in the CPU's environment gets the value 1.

Virtual CPUs, Cont'd

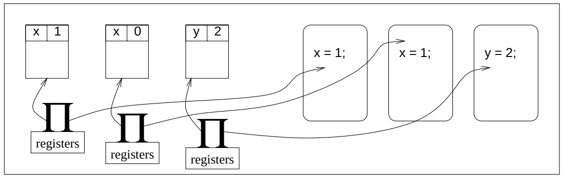
We could have a single CPU run many different activities in cycle:



but then the code for each activity has to contain code to switch between activities.e.g., to go on to next one.

Virtual CPUs, Cont'd

We pretend that we have an unbounded number of CPUs, each doing one activity, i.e., executing the activity's code in its environment.

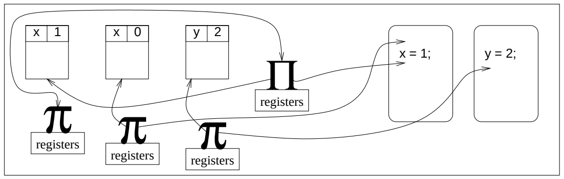


Now no activity has to know about another to switch to others.

Virtual CPUs, Cont'd

But we cannot implement the pretension, so we use processes.

A process is a virtual CPU, a data item with the same registers as a CPU.



Note also that two processes executing the same program can share one copy of the program, so long as each has its own copy of the data environment *and no process modifies any code*.

Virtual CPUs, Cont'd

That each process:

- communicates with other processes to control decision making and
- generates and receives external events.

Is this high-level view.

For now, we take the low-level view of the first figure in which each activity switches to the next.

Cycle Execution (Polling)

Suppose a system has *N* tasks to perform repeatedly, e.g., fetch input, react, etc.

```
while (1) {
    doTask1();
    doTask2();
    …
    doTaskN();
}
```

Example: Device I/O

```
while (1) {
    if (deviceInputReady()){
        c=readInput();   /* Activity 1
        changeState(c);
    }
    if (outputAvailable() &&
        deviceOutputReady()){
        c=getOutputCharacter();  /* Activity 2
        writeOutput(c);
    }
}
```

Note

No activity busy waits or busy loops.

Why not?

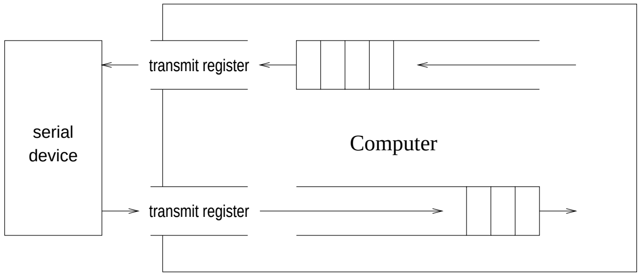
Why No Busy Waits?

- If device is not ready, could read twice before write, and vice versa.

  What would happen if you were busy waiting?

- ∴, need buffered access to devices.

Buffered Access



Polling, Cont'd

Each iteration of the polling loop executes each of the N tasks once.

For each task t_i, let

$f_i \equiv$ the function (procedure, code) that is executed by t_i.
$c_i \equiv$ the maximum time needed to execute f_i.
$T_i \equiv$ the maximum time that may elapse between successive executions of t_i.

Polling Loop Code

```
while (1){
    $f_1$;
    $f_2$;
    ...
    $f_N$;
}
```

Time Analysis

Worst case run time: $\sum_{i=1}^{N} c_i$

Real-time requirement: $\forall j, 1 \le j \le N \ (\sum_{i=1}^{N} c_i \le T_j)$

i.e., $\sum_{i=1}^{N} c_i \le \min_{1 \le i \le N} T_i$ *

What if condition * is not satisfied?

Scenario 1

Some task, say t_1, has a short service period, i.e.,

$$T_1 << \sum_{i=1}^{N} c_i$$

Then perform t_1 more often, say every other time:

Scenario 1, Cont'd

```
while (1){
    $f_1$; $f_2$;
    $f_1$; $f_3$;
    …
    $f_1$; $f_N$;
}
```

Scenario 1, Cont'd

To satisfy t_1's requirements: $c_1 + \underset{2 \le i \le N}{\mathrm{MAX}} c_i \le T_1$

and to satisfy other tasks' requirements:

$$(N-1) \times c_1 + \sum_{i=2}^{N} c_i \le \min_{2 \le i \le N} T_i$$

Scenario 2

Task t_1 has a long execution time c_1 and a long service period T_1

Split t_1 into two parts a and b:

f_a with execution time c_a, and
f_b with execution time c_b,
with $c_a + c_b \ge c_1$

Scenario 2, Cont’d

```
boolean partA = true;
while (1) {
    if (partA){
        $f_a$(); partA = false;
    } else {
        $f_b$(); partA = true;
    }
    $f_2$; …
    $f_N$;
}
```

Scenario 2, Cont’d

Constraints:

$$c_a + c_b + 2 \times \sum_{i=2}^{N} c_i \leq T_1$$

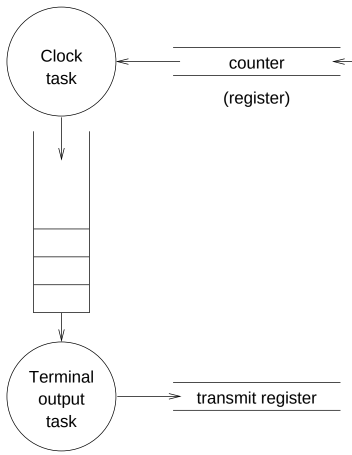
$$MAX(c_a, c_b) + \sum_{i=2}^{N} c_i \leq \min_{2 \leq i \leq N} T_i$$

Assignment 1 Devices

- terminal, input and output
- train, input and output
- clock

Managing State: DFAs, counters, queues

Example, Display Clock on Terminal



Clock task

counter

(register)

Clock

Programmable
Interval Timer (PIT)

Read documentation

Terminal output task

transmit register

Clock Polling Loop

```
while (1) {
    clock();
    terminalOutput();
}
```

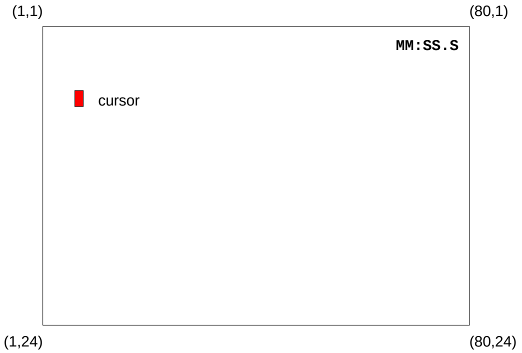
Clock Polling Loop, Cont’d

```
clock(){
    readClockCounter();
    if (counterChanged()){
        formatTime();
        if (bufferSpaceAvailable())
            writeToBuffer();
    }
}
```

Clock Polling Loop, Cont’d

```
terminalOutput(){
    char c;
    if (!bufferEmpty() && transmitReady()){
        c=bufferRemove();
        writeTransmitRegister(c);
    }
}
```

Wyse Terminal



Wyse Terminal, Cont'd

Escape sequences are commands:

For example:

Position cursor is "esc [*row* ; *col* H" with no spaces

e.g., to position at row 12, column 24:

```
\033[12;24H
```

Clear screen is "esc [2 J" with no spaces, i.e.

```
\033[2J
```

Assignment 1 Requirements

Documentation:

- operating instructions, i.e., user's manual
- program description
  - ○ main tasks
  - ○ state management, i.e., queues, DFAs, …
  - ○ diagram
  - ○ tour of source code
  - ○ etc.

Assignment 1, Cont'd

Train commands:

- tr *trainNumber speed*
- rev *trainNumber*
- sw *switchNumber direction*

Assignment 1, Cont'd

Memory management:

- no malloc, new
- Memory is statically allocated, e.g.,

  struct *x y* [*max*]

RT OS

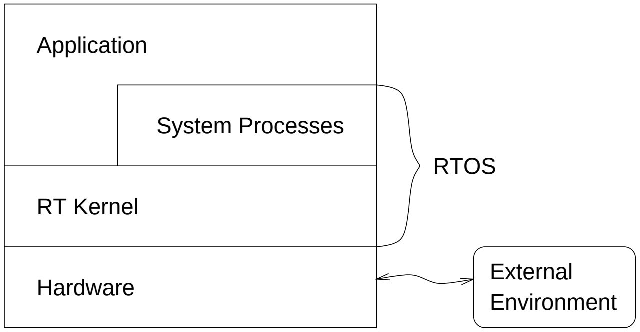
The purpose of a RT OS is to isolate the programmer from:

- explicit context switching
- state management, synchronization, and communication
- low-level device management

Requirements for a RT OS

- Asynchronous handling of events from external devices
- Explicit process abstraction, with each task:
  - having a separate address space
  - scheduled asynchronously according to its priority
  - preemptable
- Task communication and synchronization
- Time

RT Software Structure



Application

System Processes

RT Kernel

Hardware

RTOS

External Environment

RT Kernel Services

- context switching
- scheduling of processes
- communication among processes, with and without synchronization
- interrupt handling

Task Abstraction

Each task or process has a unique identifier (TID or PID)

Each task, *t*, requires:

- kernel state, i.e., a task descriptor, containing essential state information about *t*
- *t*'s code, which may be shared with other tasks
- *t*'s data space, including a stack for variables and temporaries
- *t*'s parent task, the task which created *t*; not defined for initial task

Task Abstraction, Cont'd

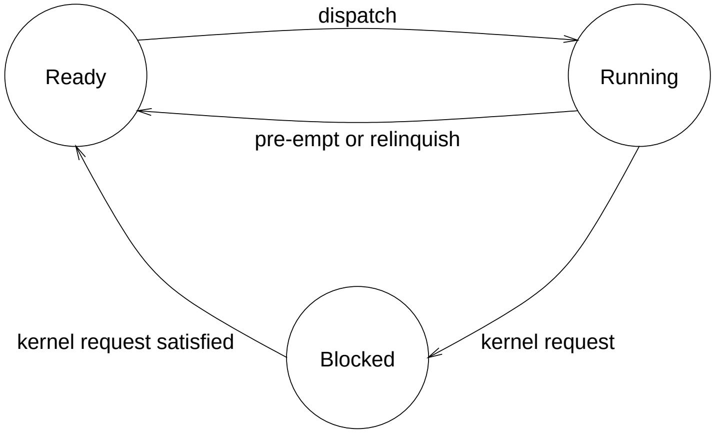
Note that the kernel state for *t*, addressed by *t*'s PID, will probably contain, a copy of the PID, pointers pointing to the other three bullet items, and other data

Task State

At any time, a task is in exactly one of three states:

1. running (or active), i.e., assigned to a real CPU which is running on its behalf
2. ready, i.e., not running, but ready to run
3. blocked, i.e., not running, but not ready to run, because the task is waiting for data which are not ready.

Task State, Cont'd



Task State, Cont'd

Note that both ready and blocked are not running, but the reasons for not running are different. In blocked, the task itself has decided that it is not ready. In ready, the external system has decided that some other task should be running.

Kernel Code

The kernel

- is itself a program and maintains its own data space and
- implements the task abstraction using *software interrupts*.

To do kernel entry, a task will do:
int n (assembly, not C)

To do kernel exit, the kernel will do;
iretl

Kernel Code, Cont'd

```
kernel(){
    initialize();
    createFirstProcess();
    while(1){Request request;
        request = getNextRequest();
        switch(request){
            case req0: …
            case req1: …
            …
        }
    }
}
```

Kernel Code, Cont'd

getNextRequest():

- determines the next active process
- dispatches the next process, with a context switch

```
getNextRequest(){Task active;
    active = getActiveProcess(); /*scheduler*/
    return(exitKernel(active)); /*context switch*/
}
```

Kernel Project

The Kernel Project has 3 parts:

1. Process Creation and Context Switching
2. Communication
3. Devices, Events, Time

Part 1

Part 1 is about:

- process creation
- context switching
- memory management
- scheduling

Some Primitives

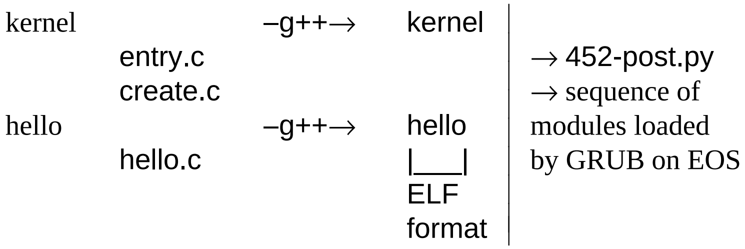
```
Pid Create(char* progName, int priority)
        /* create a named process at a priority */

void Pass()
        /* make another process running,
          while leaving this one ready */

void Exit()
        /* kill the executing process */

Pid MyPid()
        /* get executing process’s PID */
```

Each Task is a Program



```
kernel                  –g++→   kernel  |
          entry.c                       |  → 452-post.py
          create.c                      |  → sequence of
hello                   –g++→   hello   |  modules loaded
          hello.c               |___|   |  by GRUB on EOS
                                ELF     |
                                format  |
```

452-post.py is in the course account at public/tools/bin/452-post.py

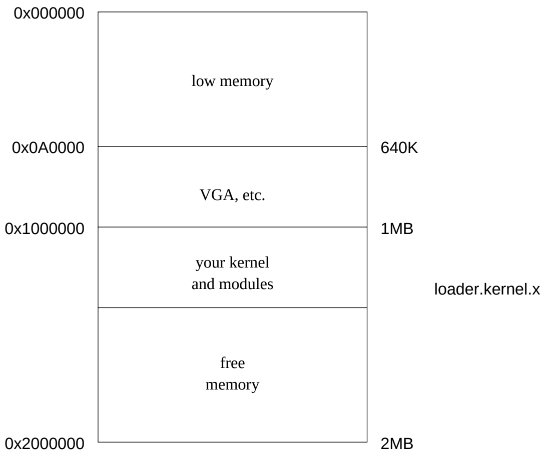
Boot Process

GRUB (GNU’s Boot Loader) implements the *Multiboot* specification, …

a single interface to boot a variety of OSs.

loader.kernel.x mentioned in the next slide is in public/examples/kernel

EOS Memory Map



0x000000

low memory

0x0A0000 640K

VGA, etc.

0x1000000 1MB

your kernel
and modules

loader.kernel.x

free
memory

0x2000000 2MB

Kernel Loading

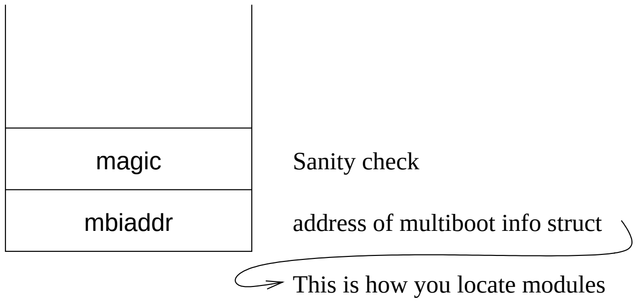
Kernel loading with multiboot.S found in public/examples/kernel

public/examples/kernel/crt0.S:

- executes before main
- linked in by gcc
- sets up task’s data segment

Multiboot Specification

Kernel is located with two values pushed on to its stack:



magic — Sanity check

mbiaddr — address of multiboot info struct

This is how you locate modules

The Way Things Work

A normal program is invoked by a process which calls the program with its paramaters.

A normal main program is invoked by a system process which calls the main program with its parameters.

The Way, Cont’d

A kernel is different.

There is no previously existing process to invoke it.

So the boot software must build, by artificial means, the stack and data that the kernel software would expect to be there if the kernel *were* invoked in the normal manner, with its parameters passed, i.e., the booter must fake it.

The Way, Cont’d

Once the stack and all data are laid out, the booter jumps to the first instruction of the kernel to effectively awaken it, …

and the CPU executes the kernel code with the stack and all other data set right.

Multiboot Specification, Resumed

kernel:

```
int main(unsigned long magic,
        multiboot_info_t * mbiaddr){
    …
}
```

Multiboot Info Struct

```
typedef struct multiboot_info
{
    …
    unsigned long mods_count; /* number of modules */
    unsigned long mods_addr;  /* address of first
                module record (module_t *) */
    …
} multiboot_info_t;
```

Note that we are using unsigned long for all data, even pointers.

Module Records

```
typedef struct module
{
    unsigned long mod_start; /* address of module start */
    unsigned long mod_end;   /* address of module end */
    unsigned long string;    /* name of module */
    …
} module_t;
```

Note that we are using unsigned long for all data, even strings.