Training in Real-Time

Tasks, Threads and Processes, Confused?

Niall Cooling BSc CEng MBCS MIEEE
Feabhas Limited

5 Lowesden Works
Lambourn Woodlands
Hungerford
Berks RG17 7RY
Tel. +44 1488 73050
Fax. +44 1488 73051
mailto:niall@feabhas.com
www.feabhas.com

Introduction

With the growth of the use of commercial off-the-shelf real-time operating systems, the terms task, thread and process are widely used in magazines, conference papers and marketing literature. Everyone using these terms has a very clear idea of their meaning. However, this paper intends to demonstrate that these seemingly innocuous terms are ambiguous and their exact meaning is dependent on the authors programming background.

Drive towards concurrent programming

The programming language "C" has undoubtedly become the most popular language for developing embedded systems over the last decade. It is a sequential language, in that code developed follows the basic structure of most standard programming languages; sequence, selection (if, case) and iteration (while, for). There is no inherent support within the language to build parts of the program that can execute concurrently.

Modern embedded systems have a growing requirement to service and respond to numerous asynchronous and synchronous inputs. Developing a sequential program that can meet real-time requirements is incredibly difficult (and quite an art). A simpler programming model than one large sequential C program, is to separate the code into multiple programs, each of which is written as a block of smaller sequential code. Each "sub-program" has a clearly defined task[1] (i.e. detecting, servicing and reacting to a given input).

Breaking the program up into a set of tasks doesn't address the issue of allowing them to run concurrently. One approach is to place each task on a separate processor referred to as *multi-processing*. This model has many advantages, most of all performance. Nevertheless, for many embedded systems (especially high volume) cost is the overriding factor and this solution isn't practicable[2].

Many small embedded systems employ a foreground/background programming model to achieve some level of responsiveness [LAB98]. This involves writing a background loop in the main program that calls the tasks (written as functions) to perform the appropriate work when required. Interrupt service routines (ISRs) are used to react to the asynchronous requests. When the ISR is invoked, due to a demand for service from an external peripheral, it pre-empts the background loop and executes (this pre-emption and invocation is performed by the hardware). The ISR can be viewed as running in the foreground. It signals the background loop (normally through some global variable) that a particular task should run. Once the ISR completes, the background loop continues from where it was interrupted. By looking at the global variables it detects that a particular task should run, and calls the appropriate "sub-program".

---

[1] Task – a specific piece of work required to be done. The New Collins Concise English Dictionary.

[2] For high-end, high-performance systems the use of Symmetrical Multi-Processing (SMP) and multi-core designs are used. However, these systems are beyond the scope of this paper and we shall limit our discussion to single processor solutions.

This model is widely (and very successfully) used on many smaller (e.g. 4 and 8 bit) microcontroller units (MCU). Nevertheless it has a number of drawbacks:

- Critical code that should ideally be in the "task" part has to be put in the ISRs for performance/safety reasons, which in turn affects the overall responsiveness of the application.
- Organising prioritisation among a number of tasks that are ready to run becomes difficult.
- Most significantly, the tasks in the background loop operate in a "run to completion" mode. Once a function has been called, the background loop will not run again until the function returns. This means that the execution of the tasks is non-preemptive.

There are some alternative models to overcome the issues listed. For many companies, though, the next natural step from a foreground/background model is to use a real-time operating system (RTOS). The majority of RTOSs today employ a very similar model. They support the scheduling of a number of tasks (each written as a sequential background program). Each task is given a priority, and the RTOS uses this to schedule tasks requesting service. Significantly, a lower priority task may be pre-empted by a higher priority task. This means that effectively one program is halted and another started. When that one finishes, the first one continues from where it was stopped (similar to the interrupt model). Switching between tasks is referred to as performing a *context switch*.

Context switching

To understand a context switch we need to establish what is happening in a target when code is executing. First most embedded systems have a similar architecture:

- A microcontroller unit (MCU) or microprocessor unit (MPU) containing a central processing unit (CPU) where the algorithmic and logical operations are performed.
- Read/Write memory – RAM.
- Persistent memory – EPROM or FLASH
- Peripherals – e.g. Serial, Timers, PWM, ADC, DAC, etc.

Consider, for example, we have developed a program in C. This has its starting function (main), and is made up of a number of further functions. Each function may consist of parameters, local (automatic) variables and executable statements (sequence, selection, iteration). In addition, there may be global (external) variables (which may or may not be initialised), constants, and dynamically allocated memory[3] (e.g. from malloc).

[3] Though we shouldn't be doing dynamic memory allocation in a hard real-time system.

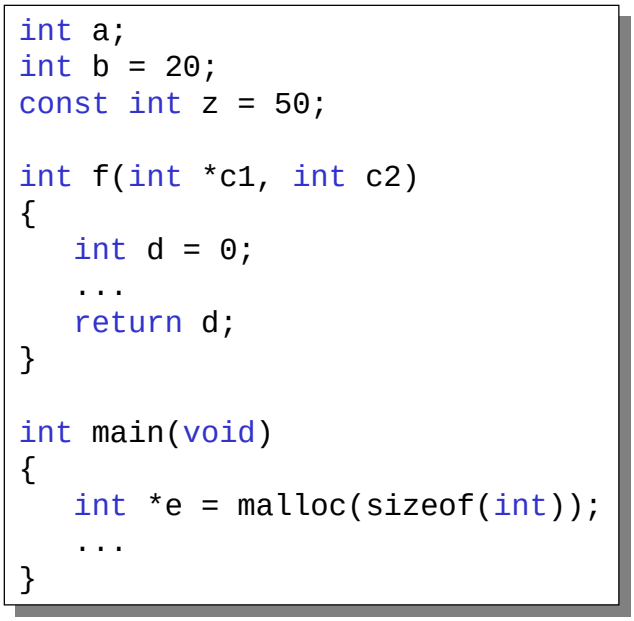


Figure 1 C Program Sections

When the C program is compiled and linked, memory has to be allocated for different parts of the program (code, globals, locals, etc.). It is the linker's responsibility to map these onto physical addresses dictated by the hardware architecture. The sections for a C program will normally consist of the following (linker specific names):

Memory Area	Section Name	Section Type	Write Operation	Initial Value	Contents
Program area	.text	Code	Disabled	Yes	Stores machine codes.
Constant area	.rodata	Data	Disabled	Yes	Constant data. This section may not be produced, especially for host compilers (e.g. UNIX/PC)[4]
Initialized data area	.data	Data	Enabled	Yes	Initialized global and static data.
Non-initialized data area	.bss	Data	Enabled	No	Stores global data whose initial values are not specified (zero initialized). BSS - "Block Started by Symbol"
Stack area	__	__	Enabled	No	Required for program execution. Dynamic Area Allocation.
Heap area	__	__	Enabled	No	Used by a library function (malloc, realloc, calloc, and new). Dynamic Area Allocation.

Table 1 Linker Sections

The areas *.text* and *.rodata* should reside in ROM, all other areas in RAM[5].

[4] This is because constants in C aren't really constants! This is beyond the scope of this paper.

Most modern CPUs have a similar core register set, consisting of the following:

- Program counter (PC) – holds the address of the current program instruction
- Stack pointer (SP) – holds the current address of the top of the stack
- Status register (SR) – indicates processing states (e.g. interrupt information)
- General registers (Rn) – a bank of registers used for data processing and address calculations

Other registers may exist (floating point registers, stack limit register, etc.). The stack is primarily used when one function calls another. When a function is called, the arguments to the parameters must be passed. In addition, any return values must be given a placeholder to write to. Finally, memory must be reserved for local variables. For general-purpose programming, arguments, local variables and the return values are stored on the stack[6] [LIN94]. Modern cross-compilers for MCUs will prefer to pass parameters, return values, and place local variables in registers, where available [FUR96, HIT98, GHS99, GHS00]. Other general registers are then used for calculations, etc.

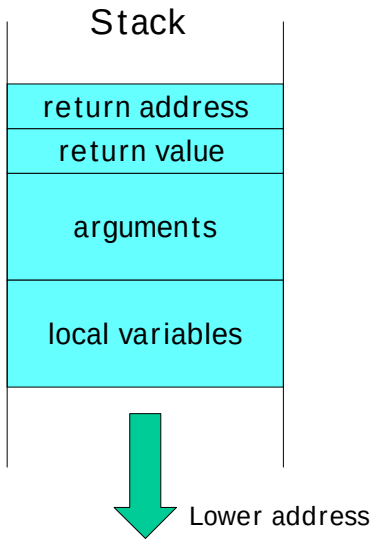


Figure 2 Function Call Stack Frame

For a sequential program (as written in C) we now have what we can refer to as a *context*. This consists of all the memory areas (including heap and stack) and the register set. The values contained are unique to this program. If we want to run two or more programs on a single processor we need a way of saving the context of one program and swapping that for the context of a second program without loosing information. This is referred to as a *context switch*. The new (second) program will make use of all the memory areas and registers. Each program assumes it "owns" the processor, so each program sees the context as a *virtual machine*.

To perform a context switch, therefore, we need to:

- Save the values of all registers - these are normally pushed onto the stack

---

[5] Many systems put the stack into on-chip ram if available due to its heavy usage.
[6] This assumes stack-relative addressing, supported by most modern processors.

- Save references to all memory sections (including the heap and stack)
- Set the values of the registers to values for the second program
- Set the references for all memory sections for the second program
- Set the program counter to point at the code for the second program and start executing

On many processors, the simplest way of performing a context switch is by invoking a software interrupt (SWI or Trap). This automatically stores the PC and SR onto the stack. By manipulating the SP and returning from the interrupt, the second program can start executing. In order to return to the original program, the steps must be performed again, which reinstalls that saved register values, etc.

Process & Task

UNIX circa 1980 - Process

During the 1980's UNIX became a very widespread platform for software development. Commercial variants derive from one of two sources; AT&T's SVR4 (System V) or UCB's (University of California, Berkley) 4.4BSD. Both "multi-programming" models were very similar.

SVR4 was design as a multi-user environment. In SVR4, a *program* is an executable file, and a ***process*** is an instance of the program in execution [BAC86]. The context of a process (for SVR4) is its state, as defined by its code, the values in global user variables, the values of processor registers it uses and the contents of the stack[7]. A process would also run from RAM, i.e. all sections would reside in RAM.

SVR4 enabled multiple processes to share a single processor. This feature, at that time, was referred to as *multiprogramming* or *multitasking* [BAC86]. Sitting at the heart of UNIX is the kernel. The kernel schedules processes so that each gets a fair share of the CPU. This is performed by allocation the CPU to a process for a time quantum. If the time quantum is exceeded, the kernel pre-empts the process and performs a context switch to another waiting for the CPU (known *round-robin* scheduling). The priority of a process is a function of recent CPU usage, with processes getting a lower priority if they have recently used the CPU.

Significantly, the majority of processes were independent, and therefore UNIX implements protection to stop one process bringing down another (by inadvertently corrupting the others context). Absolute protection can only be achieved via hardware.

Memory Management Unit (MMU)

To achieve process independence, UNIX architectures require special hardware called a *memory management unit* (MMU). This unit sits between the CPU and the memory. The MMU uses a form of lookup table (*page tables*) to map the CPUs desired address (call a *virtual address*) onto a physical address in memory. Each process sees a linear virtual address memory map based on processor size (e.g. for a 32-bit processor, a

[7] There are some addition items, such as its table slot, u area, and kernel stack, which are very UNIX specific and not necessary for the discussion here.

process can address 2^{32} addresses – 4GB address range). When a context switch takes place between two processes, the new process sees it's own 4GB virtual address space. The MMU now maps the new processes virtual addresses onto physical addresses. Because each process must use physical memory, the overall requirements for running all processes may exceed the actual memory physically available. Usually a disk is used as a secondary backing store where images of physical memory are stored and retrieved based on demands (*demand-paging*).

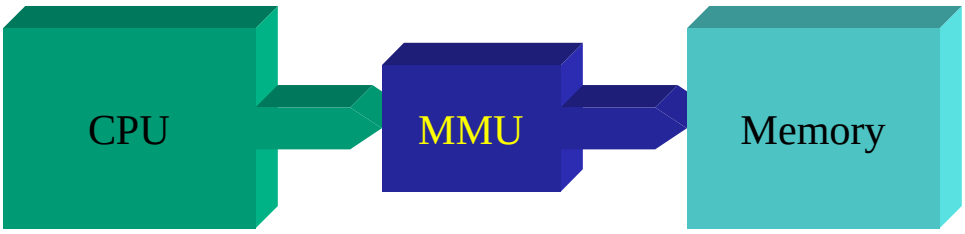


Figure 3 Memory Management Unit

As an additional protection mechanism, because of the virtual-to-physical translation using the page tables, the MMU can also specify blocks of memory as read-only, rather than read-write. Sections such as code and constants placed in RAM can then be protected against accidental updates.

When a context switch takes place, the page tables the MMU is using for the current process need saving and new page tables installing. Finally, if two process need to communicate or share data, then explicit operating system (OS) services must be used (e.g. pipe or shared memory) as each cannot, by default, see the same memory.

Real-Time Operating Systems circa 1980 - Task

In the early 1980's the first of a number of successful real-time operating systems (RTOS) started to appear, e.g. VRTX, pSOS and iRMX. Later to appear was, probably one of the best known today, VxWorks. They all supported a form of multiprogramming (similar to UNIX), but specifically for real-time embedded systems. The scheduling performed by the UNIX kernel (fair-share approach) was not appropriate for systems with fixed deadlines to meet. The RTOS programming model was referred to as *multitasking*.

There are a number of significant differences from the UNIX process model. First performance is paramount (i.e. real-time). The architectures of real-time embedded systems of that period did not support the use of an MMU for a number of reasons:

- Speed – context switch with an MMU is much slower than without one.
- Cost – many embedded systems are high volume and cost sensitive.
- Power – portable devices are very sensitive to power consumption.

Rather than developing separate C programs, a program is broken into a number of functions, each function written as if it was a main program (i.e. sequential code), very typically made up of an infinite loop (in most cases you do not want tasks to finish). This group of "main" functions existed at a peer level. The unit of concurrency for all of these RTOSs (VRTX, pSOS, VxWorks, iRMX) is the ***task***, not the process. Each RTOS offers a set of APIs (application programming interfaces) to create tasks, e.g.[LAB98]:

```
OSTaskCreate(task, ptos, prio)
```

task – address of task's code, i.e. function name
ptos – address of the top of task's stack
prio – task's priority

As these tasks are developed as part of a single program, the context for a task differs from the context of a UNIX process as follows:

- They all share the same linear physical address space
- They all share the sections .text, .data, .bss, .rodata and the heap

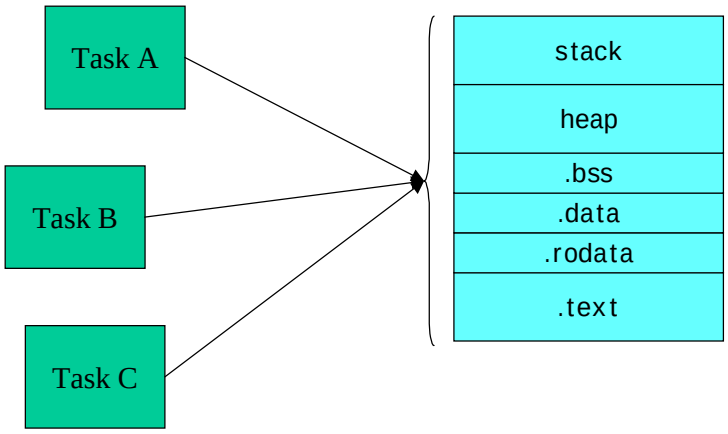
However, as each task is a sequential program (functions calling functions), each requires its own stack. In addition, the register set is part of the task's context. Performing a context switch between two tasks is very similar to switching between two processes, but without the MMU overhead. This, therefore, is much faster than performing a process context switch.

Another area is that the overall multiprogramming development becomes simpler (compared to processes). Inter-task communication is performed using common memory (i.e. no address translations to and from virtual addresses). Especially important for small real-time systems, is easy integration with ISRs. Getting ISRs to communicate with a UNIX process is not a straightforward task. Here, ISRs can communicate with tasks via common memory. Note, appropriate mutual exclusion must be in place for either form of communication (task2task, isr2task).

What also makes the RTOS so different from UNIX is the scheduling policy employed. The majority of RTOS today still employ the same basic model; know as *priority pre-emptive* scheduling. Each task is given a priority at creation, and this priority is used to determine which task runs. A higher priority task that becomes ready to run will pre-empt a lower priority one (forcing a context-switch). Therefore the order of scheduling, and thus the responsiveness of the system are under the control of the programmer, not the kernel.

Nevertheless, there is one significant issue with this model; no memory protection. This means that any task has the potential of corrupting the memory of other tasks.

Figure 4 Non memory protection bewteen tasks



This can happen quite inadvertently with problems such as a stack overrun. A stack overrun occurs when the memory for each task's stack is reserved as a contiguous array of blocks (e.g. n blocks of 1Kb for n tasks). The memory for a stack from one task naturally follows on in memory from another. If a task uses more stack than allocated for it, it will overwrite the top of the next task's stack. Unfortunately, this is only be discovered when the task with the overwritten stack runs. The context switch restores the SP, and then the corrupted values on the stack are used (e.g. a return address). The result can be quite interesting, but not what you want!

UNIX circa 1990 - Thread

At the end of the 1980's a number of experimental operating systems, and some commercial ones included support for concurrent programming. The UNIX community realised that the process structure was very "heavyweight", in that running multiple processes to solve certain natural concurrency challenges (e.g. interacting with slow devices, supporting multiple windows, networking, etc.) was very expensive in terms of computing resources (CPU and memory). The most popular mechanism was to support multiple lightweight ***threads*** within a single address space, where a thread represented a single sequential flow of control [BIR89].

A process, therefore, becomes an overall context in which the threads run. The process itself isn't executable or schedulable, the threads are[8]. The threads within a process have their own context, exactly the same as task in the RTOS. The difference being that a thread is running in a virtual address space, whereas a task runs in a physical one.

This means that an operating system supporting both the process and the thread has two different context switches:

- A context switch between two threads in the same process
- A context switch between two threads in different processes

Context switching between threads in different processes is a very expensive operation compared to switching between threads within the same process .

The use of these two terms has led to a number of overlapping expressions:

- *Lightweight thread* – read thread (sometimes called a user thread)
- *Heavyweight thread* – read process with only one thread (i.e. the traditional UNIX process)
- *Process* – read traditional UNIX process
- *Lightweight process* – read thread (sometime these only run in kernel mode)

Programming with threads is referred to as *Multithreading (MT).*

[8] Some UNIX systems still schedule based on the process not the thread

Standards (Open & Proprietary)

POSIX

In June 1995 the POSIX threads standard, POSIX.1c[IEE96], was ratified. POSIX (Portable Operating System Interface[9]) is a set of committees in the IEEE that are concerned with creating an API that can be common to all UNIX systems. There is a committee within POSIX concerned with creating a standard for writing multithreaded programs. Threads in POSIX are referred to as ***Pthreads***. However, just to make life interesting, POSIX also defines something calls ***Lightweight Processes*** (LWP). Pthreads are scheduled onto a LWP, and an LWP runs within a ***process***. Multiple Pthreads may run in an LWP, and multiple LWPs may run within a process. It is the LWPs, not the Pthreads, which are scheduled by the POSIX kernel. However, the Pthreads are scheduled within an LWP. LWPs allow a number of scheduling models that suit symmetrical multiprocessor (SMP) systems [LEW98].

Microsoft Win32 API

The Win32 API is the primary programming interface to the Microsoft Windows operating system family, including both NT and CE. The programming model has a very clear distinction between processes and threads [SOL98]:

- A ***process*** is an executable program, which defines initial code and data, a private virtual address space (MMU required), and at least one thread of execution
- A ***thread*** includes the contents of a set of volatile registers, two stacks (user mode and kernel mode[10]), and some private storage area (used by run-time libraries).

In Win32, the thread is the unit of execution and scheduling, not the process. The process is just a collection of resources. On a final note, the windows interface also has Task Manager, which of course allows you to examine processes!

OSEK/VDX

OSEK/VDX is an automotive industry standards effort to produce open systems interfaces for vehicle electronics [COO01]. OSEK defines a number of areas, one of which is OSEK/VDX Operating System [OSE00]. The OSEK OS is designed to require only minimum of hardware resources (i.e. no MMU) and therefore runs even on 8 bit microcontrollers. The OSEK OS specification clearly states that ***tasks*** are the unit of concurrency.

µITRON

The ITRON Project, which grew out of work at the University of Tokyo, creates standards for real-time operating systems used in embedded systems. The µITRON real-time kernel specification [UIT93], which was designed for consumer products and other small-scale embedded systems, has been implemented for numerous 8-bit, 16-bit and 32-bit MCUs. Currently its main support is from Japanese companies (e.g.

---

[9] I have no idea where the X comes from!

[10] UNIX also uses two stacks for user and kernel model. Entering kernel mode involves executing a specific processor instruction. Kernel mode may also be referred to as privileged mode.

Hitachi, Fujitsu, Toshiba, NEC, etc.), but it is aiming to get wider acceptance [TAK97].

The ITRON specification uses the term "***task***" refers to a unit of concurrent processing, although it does allow for both MMU and virtual addressing support. However, it does not distinguish between tasks with and without MMU support.

Linux

Undoubtedly, in the last couple of years there has been a tremendous growth in the popularity of Linux, both as a development platform and a target environment. Developed by Linus Torvalds in 1991, the original kernel was heavily influenced by Maurice Bach's book *"The Design of the UNIX Operating Systems"* [BAC86]. Much of the kernel still has its root in SVR4.

The ***process*** is the unit of execution and scheduling. However, the Linux processes share a large portion of their kernel data structures, and may be referred to as ***lightweight process*** [BOV01]. Different POSIX Pthread implementations have different mappings onto the Linux processes. Finally, processes are often called "***tasks***" in the Linux source code (e.g. a process has a state which is TASK_RUNNING, TASK_STOPPED, etc.).

Modern RTOSs

So where does this leave us (exhausted, no doubt!) when discussing operating systems for real-time embedded systems? Implementations can broadly be divided into three groups:

- OSs designed to execute without an MMU
- OSs that require an MMU
- OSs that may work with or without an MMU

MMU-less

As mentioned previously, the majority of traditional RTOSs (e.g. VxWorks, VRTX, pSOS, Nucleus, uC/OS-II) were designed to operate without an MMU. This allows for very small, fast, compact operating systems. The general term used by all of these RTOSs for the unit of concurrency is the ***task***. As expected, there are always exceptions to the norm, for example the RTOS ThreadX referrers to ***threads*** and not tasks.

Many newer RTOSs for specific applications, e.g. automotive (OSEK compliant) and DSP are designed for MMU-less systems.

All RTOSs in this area suffer from the drawback, as mentioned previously, that one task may corrupt any memory area held in RAM (stack, heap or data) of another. A task also has the potential to corrupt operating system structure that have to be held in RAM (e.g. the list of tasks waiting to use the CPU – the ready list).

What has been changing in the last few years is that modern microcontrollers and microprocessors are supporting on-chip MMUs. The majority of real-time applications do not require full MMU support with virtual addresses and a secondary

store. However, not having any memory protection is a concern in any safety related application. This has lead to a number of architectures offering a simpler form of MMU that support a memory protection scheme call segmentation [FUR96]. Segmentation breaks the address map into segments, each having a base address and limit. Access beyond the limit for a given segment causes an access violation, normally resulting in a processor exception. For example, each section from a C program (e.g. .text, .data, .stack) can be allocated their own segment. An MMU only supporting segmentation may be referred to as a Memory Protection Unit (MPU – which can also stand for Microprocessor Unit!). These devices are smaller, cheaper and require less power than conventional MMUs.

MMU required

Certain operating systems require virtual addressing, thus also require an MMU. Most widely recognised of these is Microsoft Windows NT and CE. Both require an MMU, and both have been questioned for use in real-time systems [TIM97, TIM98].

Another widely known OS is EPOC from Symbian, used in the Psion PDAs and mobile phones. This uses similar terminology to NT, in that a *process* is a resource container and a *thread* is a unit of concurrency. Because EPOC has been designed for small mobile devices a significant amount of emphasis is placed on understanding the performance issues when context switching between threads in a process and across process boundaries [TAS00].

Wind River (http://www.windriver.com) have recently announced a sister product to their VxWorks RTOS, called VxWorks AE. VxWorks AE is designed to work with microprocessors that have full MMUs. VxWorks AE, however, is initially only available for certain processor architectures that have on-board memory management units. In spite of this, VxWorks AE still uses the term *task*.

With or without MMU

Probably the commercial RTOS that lead the way with memory protection for real-time embedded system is OSE from Enea OSE Systems (http://www.enea.com). Enea introduced their own memory management system (MMS) to take full advantage of an MMU. The OSE RTOS can also be used without an MMU, but then suffers the same issues as any MMU-less RTOS.

Just to add to the terminology confusion, OSE refers to its unit of concurrency as a ***process***, whether processes are running with or without MMU support!

Languages

So far the discussion has centred on operating system terminology, due to C being used as the example language. C is a sequential language with no inherent support for concurrency. This is also true of C++. Either language requires specific APIs to use any concurrency supported by an underlying RTOS.

Certain languages support the concepts of concurrency within their semantics, most notably, from a commercial perspective, Ada and Java.

Ada was first introduced as a standard in 1983 and then updated in 1995. The execution of an Ada program consists of one or more ***tasks***. Each task represents a separate thread of control that proceeds independently and concurrently between points where it interacts with other tasks [ADA95]. Tasks can communicate using shared variables. This roughly aligns Ada tasks with non-MMU RTOS tasks (from here one, an *RTOS task* shall mean an non-MMU RTOS task) and POSIX threads.

The Ada specification supports a series of specialized-needs annexes that compiler vendors may optionally support. Annex E, defines facilities for supporting distributed systems, based on a concept called *partitions*. The full description of partitions is beyond the scope of this paper, but sufficient to say, an "active" partition can control its own tasks using its own run-time system [COH96]. The concept of a partition maps well onto a POSIX LWPs, with Ada tasks as Pthreads.

Java, the darling of the Internet, is also positioned as a language for real-time embedded systems[11]. The Java language specification directly supports the concepts of ***threads***. Java threads run within a single Java program, and thus map conceptually (as expected) to POSIX threads or RTOS tasks. Threads in separate Java programs have a defined mechanism, call RMI (Remote Method Invocation), with which to communicate.

As stated, Ada Tasks and Java Threads map naturally onto RTOS tasks. However, this doesn't mean they have been mapped onto the underlying OS primitives. Both languages have their own scheduling and dispatching algorithms (e.g. the Java Virtual Machine – JVM and the Ada run-time system), by default these are part of the executable program.

A quick and "dirty" solution a number of vendors employ to get "real-time Java" solution to market is to run the JVM within one RTOS task [WEI98]. This leads to a two tier scheduling system; the RTOS task scheduling and the Java thread scheduling and neither knows about the other. This has two important consequences:

- The Java threads only run when the containing RTOS task is scheduled
- If a Java thread makes a call to the RTOS library, and that call blocks, all Java threads become blocked

[11] We will not discuss it suitability or not here

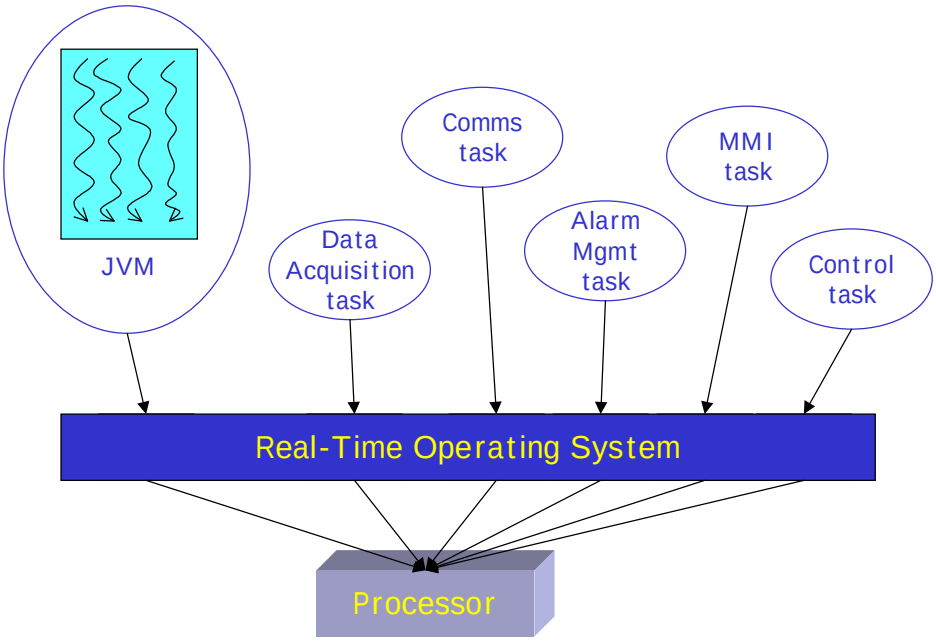


Figure 5 Mixed RTOS and JVM Environment

Real-Time NT, RTLinux and µClinux

For hard or fast real-time systems, both NT and Linux have generally been classed as unsuitable in their "vanilla" form. There are a number of variants for both OSs claiming to add "real-time" capabilities.

Because NT is not open source, only Microsoft can modify the NT kernel. Therefore, for another company to extend NT for real-time, a general approach is to make NT co-exist with an RTOS. This is achieved in one of two ways:

- NT runs as a task on top of the RTOS
- The layer between NT and the hardware (called the Hardware Abstraction Layer – HAL) is modified to intercept interrupts and allow the RTOS scheduler to run.

As Linux is open source, there have been three different approaches to give Linux real-time capabilities:

- Modifications to the Kernel to reduce the non- preemptible parts (this can be achieved various ways)
- Run the Linux kernel as the lowest priority task alongside RTOS tasks.
- Simplify the kernel to remove the need for an MMU

The non-MMU approach, referred to as µCLinux (http://www.uCLinux.org) has been developed by Lineo Inc to target microcontrollers without MMU support.

Unified Modeling Language (UML)

UML is fast becoming an accepted approach for the design of software systems, real-time or not. A design approach should help make the transition from a set of

requirements to a design; that can ultimately be implemented using a programming language and possibly an operating system.

As such, a UML design must be able to represent concurrency. To achieve this UML defines a set of semantics and notation for an item called an *active object*. The definition of an active object is:

> *"An active object is an Object that owns a thread of control. Processes and tasks are traditional kinds of active objects". [UML99 – page 3-122]*

Unfortunately, the UML specification does not define their interpretation of a process or task! This leads to the problem that for a given notation (the active object) is ambiguous. Different designers and vendors are using this notation to represent concurrency without clarifying its actual interpretation. This is important for real-time systems due to the issue of different scheduling policies and context switch times depending on a given mapping.

Summary

So where has this got us? Hopefully to demonstrate that there different concurrency models we need to recognise when discussing task, threads and processes. Each one effects the context switch times and scheduling issues of the system.

The important questions to ask of any implementation are:

- Is an MMU part of the hardware architecture?
- If so, is the MMU being used for just segmentation protection or full virtual addressing?
- Does the implementation language support concurrency?
- If so, how is this mapped onto any underlying OS primitives?
- Do two OSs co-exist, and if so how?

My own general guidance:

- If no MMU is present than I use the term *task*
- If an MMU is present and virtual addressing is being used; *thread* for the schedulable entity, *process* represents the resource container the threads share.

My overriding interest is to understand the impact of communication between to units of concurrency (thus the impact on context switch) and the resource requirements. The missing term is where we have an MMU providing memory protection without virtual addressing. I would, personally, still regard this as a *task* (due to the linear address map).

As with most things, this is purely subjective and I don't expect you to agree with me. What is important, next times you use the phase "*task*", "*thread*" or "*process*" be clear what you mean.

Discussions with my father, Dr. Jim Cooling and Chris Czarnecki motivated me to write this paper. If you find it useful, please thank them!

References:

ADA95	Ada 95 Reference Manual, ANSI/ISO/IEC-8652:1995, January 1995
BAC86	BACH, Maurice J "The Design of the UNIX Operating System", Prentice-Hall, 1986, ISBN: 0-13-201757-1.
BIR89	BIRRELL, Andrew D "An Introduction to Programming with Threads", System Research Center (SRC) Report, Digital Equipment Corporation, 1989.
BOV01	BOVET, Daniel P. and CESATI, Marco "Understanding the Linux Kernel", O'Reilly, 2001, ISBN: 0-596-00002-2
COH96	COHEN, Norman H. "Ada as a second language", McGraw Hill, 1996, ISBN: 0-07-011607-5.
COO01	COOMES, Andrew "Under the hood of an OSEK-compliant RTOS", Embedded Systems, pp44-46, March 2001, Vol. 5 No. 33.
LAB98	LABROSSE, Jean J "MicroC/OS-II", R & D Books; 1998, ISBN: 0-87930-543-6.
LIN94	LINDEN, Peter van der "Expert C Programming", SunSoft Press; 1994, ISBN: 0-13177-429-8.
GHS99	"Embedded ARM/Thumb Development Guide", Green Hills Software Inc. 1999. PubID: D02B-I1199-89NG.
GHS00	"Embedded SH Development Guide", Green Hills Software Inc. 2000. PubID: D05B-I0300-89NG.
FUR96	FURBER, Steve "ARM System Architecture", Addison-Wesley, 1996, ISBN: 0-201-40352-8.
HIT98	"SuperH RISC engine, C/C++ Compiler User's Manual", Hitachi, Ltd., 1998, Ref: ADE-702-179.
IEE96	9945-1 : 1996 (ISO/IEC) [IEEE/ANSI Std 1003.1, 1996 Edition] Information Technology--Portable Operating System Interface (POSIX&)--Part 1: System Application: Program Interface (API) [C Language] *This edition incorporates extensions for realtime applications (1003.1b-1993, 1003.1i-1995) and threads (1003.1c-1995).*
LEW98	LEWIS, Bil and BERG, Daniel J. "Multithreaded Programming with Pthreads", Prentice Hall, 1998, ISBN: 0-13-680729-1.
OSE00	OSEK/VDX Operating System Specification 2.1r1, 13 November 2000. http://www-iiit.etec.uni-karlsruhe.de/~osek/
SOL98	SOLOMON, David A. "Inside Windows NT 2nd Edition", Microsoft Press, 1998, ISBN: 1-57231-677-2.
TAK97	TAKADA, Hiroaki "µITRO: A Standard Real-Time Kernel Specification for Small-Scale Embedded Systems", Real Time Magazine, September 1997, pp57-63.
TAS00	TASKER, Martin et al. "Professional Symbian Programming", Wrox Press, 2000, ISBN: 1-861003-03-X.
TIM97	TIMMERMAN, Martin "Windows NT as Real-Time OS ?", Real Time Magazine, September 1997, pp6-13.
TIM98	TIMMERMAN, Martin "Is Windows CE 2.0a real threat to the RTOS World?", Real Time Magazine, September 1998, pp20-30
UIT93	uITRON 3.0 Specification, General Editor: Ken Sakamura, Version number: Ver 3.02.00, 1993. http://tron.um.u-tokyo.ac.jp/TRON/ITRON/home-e.html
UML99	OMG Unified Modelling Language Specification, Version 1.3, June 1999 http://www.omg.org.
WEI98	WEINBERG, William "Real-Time Java Implementation for Embedded Environments", Real Time Magazine, March 1998, pp43-49.