B L A C K W O O D D E S I G N S

Real-Time Kernel

CONCEPTS & TECHNIQUES

AUTHOR RANDALL MAAS

OVERVIEW This is fascicle describes a portable, real-time scheduler suitable for embedded systems.

BENEFITS Consistent software behaviour

HOW IT WORKS Classic algorithms and data structures

USES Embedded systems

STRUCTURES
Deterministic allocators
Queues
Timers
Priority Lists
Schedulers
Flags
Mutexes

SURGEON GENERAL'S WARNING – Prolonged butt-scratching may result in Repetitive Stress Injury. Workers-compensation is not approved for ass failure. Rubber Donut not included.

FILE: G:\projects\RTOS-emu\doc\RTK overview - 2006-9-20.doc

Preface

This fascicle aims to describe relevant techniques for *real-time scheduling in embedded systems.*These are time critical systems, which guarantee that all the critical sections of the program are processed in a timely manner and meet their deadlines.

1 ORGANIZATION OF THIS FASCICLE

CHAPTER 1: PREFACE. This chapter, describing the other chapters and further reading

CHAPTER 2: OVERVIEW. We begin with a descriptive overview of real-time scheduling.

CHAPTER 3: LISTS. This chapter describes lists, which serve as the basis for fast allocation nad queues.

CHAPTER 4: TIMERS. This chapter outlines an efficient mechanism for timers.

CHAPTER 5: SCHEDULING. This chapter outlines priority based scheduling.

CHAPTER 6: THREAD SWITCHING. This chapter describes the mechanisms for switching between eligible threads.

CHAPTER 7: IINTERPROCESS CONTROL MECHANISMS. This chapter describes mechanisms to signal threads including flags, queues and mutexes.

2 GLOSSARY, ACRONYMS, AND ABBREVIATIONS

Table 1: *Common acronyms and abbreviations*

Abbreviation / Acronym	Phrase
Idx	index
Knl	kernel
Flg	flag
Len	length
MCU	microcontroller (unit)
Mem	memory
Sys	system
Thr	thread

3 REFERENCE DOCUMENTATION AND RESOURCES

Note: most references will appear in the margins, significant references will appear at the end of their respective chapter.

001-40921, Cypress, Edward Nova, *"AN2046 Algorithm – Real Time Operation System for PSoC® MCUs"*, Rev. A 1 2010-March 4

A succinct introduction to rate monotonic analysis. See *Real-Time SystemsDesign and Analysis* for a wider and more detailed treatment.

Dunkels, Adam. *Protothreads*. A generalized implementation of Simon Tatham's coroutines. http://dunkels.com/adam/pt/

Labrosse, Jean *MicroC/OS-II: The Real-Time Kernel,* 2nd Ed, CMP Books, 2002

Laplante, Phillip. *Real-Time Systems Design and Analysis,* 3rd edition. IEEE, 2004

Tatham, Simon. *"Coroutines in C"*
http://www.chiark.greenend.org.uk/~sgtatham/coroutines.html

Varghese, George and Tony Lauck, "Hashed and Hierarchical Timing Wheels: Data Structures for Efficient Implementation of a Timer Facility" 1987-Nov, *SOSP 87 Proceedings of the eleventh ACM Symposium on Operating Systems principles,* p25-38

https://en.wikipedia.org/wiki/Setjmp.h

CHAPTER 2

Overview

RANDALL MAAS This chapter is an overview of the real-time kernel.

- Introduction
- Key concepts
- Patterns

1. INTRODUCTION

Real-time software measures, monitors, analyzes and controls real-world events as they occur; often it must respond within in strict time constraints. This includes:

- Monitoring or data capture from the external environment
- Analysis of data in order to transform it into forms required by the application
- Controls to respond to external events
- Coordinating system components.

Other fascicles will examine those features. This one will focus on the time critical aspects. Time critical systems guarantee that all the critical sections of the program are processed in a timely manner and meet their deadlines.

As we are only concerned with deadlines we will look at the scheduler and tasking kernel. A kernel is "a small nucleus of software that provides only the minimal facilities necessary for implementing additional … services"

It is important that the system have very consistent behaviour. Very consistent timing. Fancy, speculative methods have too much variation. Don't volkswagen the test!

All fundamental threads (procedures) return with an error or success within a bounded time. These either

- Are implemented with a constant time (O(1)) algorithm, or
- Take a timeout value, and may pause the thread.

Extra:

- Prioritization to select for eligible work threads. A subsequent chapter will describe prioritization and how it is implemented.
- Division of work
- Watchdog

Fast allocation of internal resources. Fast management of internal resources.

All fundamental threads (procedures) return with an error or success within a bounded time. These either

- Are implemented with a constant time (O(1)) algorithm, or
- Prioritization to select for eligible work tasks; pre-emptable

2. PATTERNS

2.1. FAST ALLOCATION

Deterministic – fast – allocation of resources is a key theme in a real-time system. Allocations are done in O(1) allocation time, which means there is no looping – no iteration over a list, etc. The most common method is to employ a linked list, removing the first item

The linked list is initially constructed at initialization. The memory is divided up and the lists are constructed.

2.2. PATTERN: PEND & POST

The pattern in the API is

- To request a resource, *pend* on it.
- To release a resource (or pass it) *post* it.

If the resource is available, a pend returns immediately with the resource.

2.3. TIME BOUNDS

There are procedures that can't return immediately with a resource. These wait for a resource –the processor, mutex, data in a queue, etc – for bounded amount of time. (An 'infinite' can be used to disable the timeout).

- If the resource is immediately available, the resource is returned
- If the resource is not immediately available, the thread is blocked and given a timer.
- When the resource becomes available the highest effective priority waiting thread is given the resource, its timer is canceled, and the thread is placed onto the processor run queue.
- When the timer it expires, the thread is removed from the waiting list of the resource in question, the thread is given an error, and placed onto the processor run queue.

3. PRIORITIZATION AND EFFECTIVE SEQUENTIAL EXECUTION

A modern microcontroller divides work up among:

- Peripherals
- Interrupt handlers
- Threads of execution

Each of these three may have their own prioritization levels, and are scheduled independently. A later chapter will discuss how to assign these. As the peripherals work independently of the

processor, it is often better to delegate work to them. ISR's are always higher priority than threads.

Interrupts are used for:

- Used to service peripherals quickly (but at a conceptual level underneath this)
- Process logic only when needed
- Trigger faster response in logic

On some microcontrollers, the prioritizable interrupt controller can have a great range of prioritization. On many others the controller can have as few as four levels of prioritization. The diagram gives some idea of how higher interrupts & exceptions can interrupt lower ones.

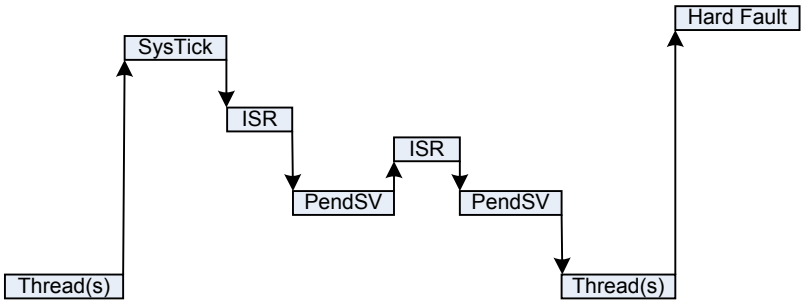

***Figure 1: L**inked list.*

Exceptions are such as systick and pendsv are used for the kernel's purposes. Systick is used to implement the main timers. Pensdsv is used to perform work that is lower priority than interrupts, and to switch between threads.

3.1. SCHEDULER

The kernel does scheduling and context switching. The scheduling is time based, and prioritize from ready set. Threads are switched between in one of three circumstances:

- A thread is waiting for a timer to expire
- A timer has expired and woken a thread
- An IPC object has changed state, waking the highest priority thread waiting on it.

3.2. THE MORE SOPHISTICATED IPC MECHANISMS

IPC mechanism

- Flags. The intent is that an interrupt handler posts to the binary semaphore, waking the thread that is pending on it.
- Mutexes to protect critical sections within a thread. The thread that pends on the mutex posts to it.
- Queues. This allows one thread to send messages to another thread.

The other kinds of IPC can be built on those two templates. E.g. have a flag so blocking on the queue is to block on the flag. The other kinds of IPC can include:

- Channel-based (e.g. queues): using & what goes over it, variable-length data, message-data

- Shared memory
- Locks, Semaphore, Conditions
- Call, or interrupt
- Process control

4. WHAT IS NOT SHOWN

Some other things that won't be covered

- Memory protection
- Access control of resources and peripherals,
- Privileged code execution
- Service calls, which is often a technique to allow access to privileged code within protected areas
- Flags and Queues can be implemented as single-waiter or multiple-waiter. With multiple waiters, a flag is simply a mutex.

This is shown using modification. Most accesses are to *volatile* variables, and interrupts are disabled around the code blocks. This design was chosen for two reasons:

1. Clarity: this style is easier to understand than compare-and-swap (CAS) forms
2. Many (if not most) embedded microcontrollers do not support CAS.

Future variations can extend this and implement procedures using CAS.

SLEEP / LOW POWER SUPPORT. The task switch can be extended to support lower power usage. When there are no ready threads – and the time to next timer expiration is fairly long – the processor can be put into an idle state. Before it would so, it would set the processors timer to wake after that duration (e.g. slow down systick), and set the processor to wake on interrupt/exception. When the processor wakes up, it notes the duration it slept for, so as to be able to adjust clocks.

CHAPTER 3

Lists

RANDALL MAAS This chapter is an overview of lists – fast data structures – as used within the real-time kernel.

- Introduction and uses of lists
- Types and structure of linked lists

5. INTRODUCTION

Linked lists provide fast, constant time (O(1)) algorithms for their access operation. This feature makes them desirable for use within the real-time kernel. Linked lists are used by the kernel to:

- Track timers in the timer module,
- Tracking the ready-to-run tasks and other work units
- Manage ingoing and outgoing communication buffers
- Provide fast, deterministic allocation of internal resources.

5.1. FAST QUEUE MANAGEMENT OPERATIONS

Lists can provide the following, often useful operations in O(1) time:

1. Appending an item on the list
2. Prepending an item on the list
3. Removing an arbitrary item from the list
4. Joining two lists

5.2. FAST ALLOCATION

Deterministic, fast allocation of a memory resource is done in O(1) allocation time. This means that there is no looping or iteration over elements. The allocators are built on linked lists, using them for the following operations:

- The allocation removes the first item from the list,
- Freeing an item, places it back onto the head (or tail) of the list.

The list is constructed at initialization. At that time the memory is divided up and used to construct the initial list.

6. LINKED LISTS

This section describes the linked lists used to manage resource allocation and lists (or queues).

There are two kinds of linked lists, and doubly linked lists. The former just has a link to the next node, while later also offers a link to the previous node.

And there are two kinds of list topologies:

- *Null-terminated* lists. The last node in the list has it's "next" pointer with a null value.
- *Circular* lists. The advantage of this structure is that it allows fast (O(1) time complexity) appending or prepending of nodes to the list.

6.1. SIMPLE LINKED LISTS

A basic linked list looks like:

***Figure 2:** Linked list.*

It can also be useful to put items onto the end of the tail of the list, while taking items from the head of the list:

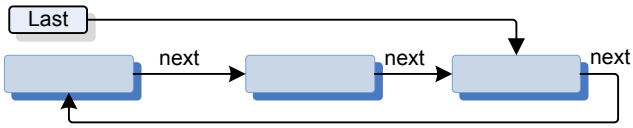

***Figure 3:** Circular linked list.*

6.2. DOUBLY LINKED LISTS: REMOVING AN ARBITRARY NODE

The null-terminated *doubly* linked list is sketched below:

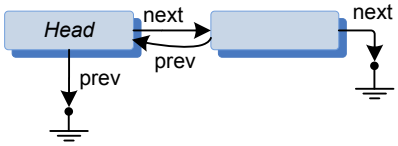

***Figure 4:** Null terminated list*

The corresponding circular *doubly* linked list is sketched below:

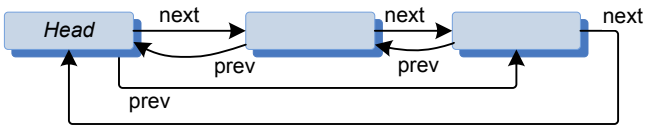

***Figure 5:** Circular list*

The prev field lets each of the following be done in constant time

1. Removing an arbitrary item from the list
2. Prepending an item on the list
3. Appending an item on the list
4. Joining two lists

6.3. RULES FOR A NULL-TERMINATED LIST

The rules of the list management for null-terminated lists:

1. Each node's `next` field points to the next node of the list
2. The `next` field of the last node is *null* in a null-terminated list.
3. The `prev` field of each node – other than the first node – points to the previous node of the list. That is `node->prev->next == node`
4. The first node's `prev` field is null in a null-terminated list.

6.4. RULES FOR CIRCULAR LIST

The rules of the list management for circular lists:

1. Each node's `next` field points to the next node of the list
2. The `next` field of the last node points to the first node in a circular list.
3. The `prev` field of each node – other than the first node – points to the previous node of the list. That is `node->prev->next == node`
4. The first node's `prev` field points to the last item of the list, in a circular list.

CHAPTER 4

Timers

RANDALL MAAS This chapter is an overview of timers in the real-time kernel.

- Introduction
- Key concepts: timing wheels

7. INTRODUCTION

Timers are provided at the kernel level for three reasons:

1. The kernel bounds the amount of time a thread waits for a resource,
2. To allow a thread to delay for a period of time,
3. Allowing the application to trigger events for purposes of creating a responsive, reliable system. This may include timeouts in communication, debouncing signals, and so forth.

Note: The timer facility is not intended to measure durations with accuracy. The processor high resolution time counter (e.g. cycle counter) is more often employed for measuring duration.

A basic timer has three common elements:

- List management information. This is usually a doubly-linked list.
- The number of ticks left before the timer expires.
- A handler to call when the timer expires.

7.1. SOURCES FOR TIMER EVENTS

The core concept is that a timer expires after a specified number of timer events (called ticks). Sources for timer events can be:

- System tick (SysTick) interrupts on the ARM Cortex processors,
- Timer interrupts
- Event counter interrupts
- Interrupts from external real-time clocks. Setting alarm, selecting next timer.
- Watchdog timers.

8. TIMING WHEEL

Timers are managed using a timing wheel. There is at least one wheel per timer event source. (Using multiple timers is useful for very slow time-bases with very long term timers.) The timing wheel approach is one of the most efficient data structures for managing timers. It

scans few timers scanned at interrupt time, and has a worst case performance of the best naive timer management algorithm.

The timing wheel has slots, which correspond to ticks. The current slot matches the current tick; the next slot matches the next tick, and so on.

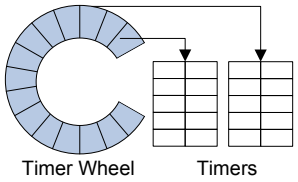

Figure 6: *Timing wheel*

Each spoke of the wheel has two lists of timers

- The list of timers that expire when the tick occurs.
- A list of timers that will expire in a later cycle.

When the timer event occurs:

1. The interrupt handler moves to the next spoke at the end of the wheel.
2. All of the timers in the spoke are append to the expired list. This is a constant time operation. The timers are not called to during the interrupt, reducing the duration spent in SysTick or other interrupt.
3. The second list is used to reload the spoke with timers. The timers on this list have their remaining time decremented by the number of spokes on the while. If they will expire on the next cycle thru the wheel, they are removed from this list and appended to the first list. This operation is not constant time, but is often very short. The worst case time is proportional to the number of active timers in the system.

8.1. CALLING THE TIMER HANDLERS

To allow other interrupts to serviced in a timely fashion – and respect their priority – the handlers for expired timers are not called at the timer interrupt. The expired timers are called in one of four places:

- A custom thread just to run timers.[1]
- The main run loop, if the system has a fixed, predictable cycle time.
- In a lowest priority interrupt or exception handler. On systems like the ARM Cortex, this is the PendSV handler. A service-call is made to trigger the PendSV handler at the end of the timer tick handler.
- In the thread that owns the timer. In this case, the expired timers are queued back with their thread at one of the above techniques. Usually the stack is the kernel stack, not the thread's stack. (This is done as thread switch often cannot accurately preserve the threads stack if the timer is run then as well.)

[1] µC/OS-II has a separate, often high-priority, task for the purpose of performing the callbacks. There is a risk with approach that the timer handlers are pre-empting higher priority work.

As a generalization, the PendSV handler is the best approach. It practice not all timers can be meaningfully assigned an owning thread.

8.2. CANCELLING TIMERS

Timer lists are implemented as doubly linked lists. This allows a timer to be cancelled – and removed from a list – in constant time. Interrupts must be disabled during this operation.

CHAPTER 5

Scheduling

RANDALL MAAS This chapter is an overview of scheduling real-time tasks (and other units of work)

- Introduction
- Priorities & Key concepts for the scheduler
- Prioritized wait queues

9. INTRODUCTION

The scheduled work is held in prioritized work queues. These queues used with such constructs as:

- A CPU's ready-to-run queue
- The waiters on a semaphore, mutex, CPU or other resource
- The IO work for a storage device, or communication bus

The steps in preparing a real-time system for scheduling are:

1. Organize information about the system
2. Check that the system can meet its deadlines
3. Assign priorities
4. Create tasks (or other work unit) structures that can be managed, and assign them priorities.
5. Create priority queues to manage the tasks.

10. PROCESSOR LOAD, AND MEETING DEADLINES

The first design task is to analyze whether the system – or specific work processor – can meet its deadlines. This involves identifying the following:

- The number of tasks the system is to do
- Whether the system is pre-emptive or not,
- How frequently each task will use the processor
- How long each task will use the processor
- How long each task will be blocked, if it is a pre-emptive system

A task is kernel-managed thread, interrupt or other unit of analysis.

A non-preemptive system may work as:

- A main run-loop that dispatches short fixed-length elements of work (often called a run-to-completion model). There are few interrupts and the interrupts are very short, setting only a flag to the main loop.

Preemptive context switching can add complexity. A pre-emptive system may be:

- A set of independent, prioritizable interrupts, where the interrupts do work in their handlers, or
- A system-call that switches between threads that the kernel manages, blocking threads to wait for resources, or when a regular interrupt occurs to switch threads. This approach is often the most complex, while also being the most flexible.

10.1. RATE MONOTONIC ANALYSIS

The first is: *can* the system meet deadlines? This is true so long as long as the computed utilization is less than the utilization bound. The utilization bound is given by:

$$n\left(2^{\frac{1}{n}}-1\right)$$

Equation 1: *Utilization bound*

Where:

n is the (maximum) number of tasks

NON-PREEMPTIVE COMPUTED UTILIZATION: The computed utilization in a non-preemptive system is given by:

$$\sum_{j=1}^{n}\left(\frac{E_j}{P_j}\right)$$

Equation 2: *Computed utilization without preemption*

Where

E_j – the execution duration of task j

P_j – the period of task j

PREEMPTIVE COMPUTED UTILIZATION: The computed utilization in a preemptive system is given by:

$$\sum_{j=1}^{n}\left(\frac{E_j}{P_j}\right)+\max_j\sum_{j=1}^{n}\left(\frac{B_j}{P_j}\right)$$

Equation 3: *Computed utilization in a preemptive system*

Where

B_j is the blocking time for task j

10.2. PRIORITY ASSIGNMENT: OPTIMAL SCHEDULING

Priorities have a very specific role and meaning. Resources such as the processor, mutexes, flags, and so on can have a set of interrupt handlers and threads waiting to use them. Priorities are used to answer the question: When there is more than one waiting, which should be selected?

Priority is the simple, fast, consistent mechanism to do this. Each interrupt and thread is assigned a number. A run-loop would select the next task with most significant priority from its ready pool.

A pre-emptive system is similar. The interrupt or thread with the lowest number in the waiting pool is selected to run. With interrupts, the current executing interrupt or context can

be blocked if an interrupt is raised with a more significant priority. When an interrupt exits, the next most significant pending interrupt is selected. The process is similar for waiting threads in a kernel scheduler.

There is a systematic, rational method for optimally assigning priorities. That is, one can demonstrate that a set of tasks will always meet their deadlines, even under worst case situation. Priority inversions must be bounded (i.e., unbounded inversions must be prevented). In the analysis tasks are assigned a fixed priority (either absolute # or relative to other task), that is not changeable at runtime.

The priority assigned based on how often the task runs in the worst case.

Tasks that are given the same priority, any of the following can be done:

1. Merge the tasks, and just run Task 1, Task 2, etc.
2. Give them equal priority, with round robin or run-to-completion behavior.
3. Give them adjacent unequal priorities

Thread prioritizations. Thread id ordering is the same as the prioritization ordering. Internally the priority number and thread id are the same.

11. PRIORITIZED WAIT QUEUES

11.1. PRIORITY LIST

Apriority is list need to represent the ready & waiting work for any worker or processor. This set of eligible waiters is represented as a bit list. A single word if the possible max size is small. The following diagram shows how a single-word list contains only thread 1:

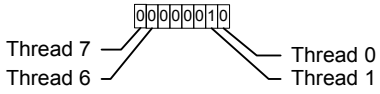

Figure 7: *Word-size list*

To queue the highest priority element is a matter of finding the least-significant bit that is set. On most processors this is a single instruction. Finding the least-significant bit without such an instruction is a small, fixed number of instructions and no iteration.

When the system has more priorities than a single word can hold more levels are added. The second level of this bit-tree is used to tell which of the words has a non-zero bit. (This dequeuing process will remain constant time). A three level tree is rarely needed, but if it were, the (top) third level would indicate which of the two-level trees contains a set bit.

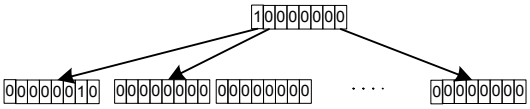

Figure 8: *Two-level tree*

The actual waiting items can be stored in an array that is indexed by the priority. In the case of several items with the same priority – such as IO requests – the array could hold list pointers.

11.2. EXAMPLE STRUCTURE

A prioritize queue might be include the follow fields:

Field	Description
level0	A word that holds the list of items waiting for this resource. In the case of a two-level tree, it is list which items in level1[] have ready items.
level1[]	An array of words that holds the lists of items waiting for this resource. This is optional, used only in a two-level tree.

Table 2: *Prioritized thread list*

Threads can be added and removed from the list using the following operations (macros):

Operation	Description
ffsl()	Return the least-significant set bit in the word. This is often a machine instruction, or implemented with a de Bruijn lookup.
ThrAdd()	Adds the specified thread to the waiting list
ThrNext()	Returns the highest priority thread in the waiting list.
ThrRemove()	Removes the specified thread from the waiting list.

Table 3: *Thread functions*

12. THE PROCESSOR CONTROL

The processor, and IPC mechanisms share the common base structure above.

12.1. CONTROLLERING THE PROCESSOR'S WAITING LIST

The central waiter list in the system is the one waiting for processor use. Manipulating this list is needed to coordinate threads. This is done with a core set of fundamental inter-process communication (IPC). This is a tiny set, and the fancier IPC mechanisms are built on them.

These core mechanisms are mutexes and flags. They share a very similar structure. Each is a simple (abstract) resource that can have only one owner. That is, only thread can own the resource. These resources also have a list of threads waiting to get ownership. (It is recommended that flags be configured so that they allow only a single waiter.)

Raising/setting the flag

1. It unblocks the waiter (or highest priority waiter) when the flag goes from low to high.
2. If no owner, the flag's owner is made to be that of the highest priority waiter
3. Event is queued to service the waiting task.

On wake of the owning tasks, it completes the receive. The flag is set to 0.

12.2. SPLITTING A TASK UP

It is worth discussing splitting a work unit into multiple parts, each with successively lower units of priority.

On some processors it is possibly to assign priorities to interrupts. The question is: how much work should be done in the high priority interrupt, how much should be done at a lower (possibly intermediate) priority, and how is the separation to be determined and performed?

For example, the ARM Cortex processors have prioritized interrupts. The handlers can be implemented so that the time critical works is done in the high priority interrupt handler, then

queue a service call and have the remaining work handled in PendSV. The PendSV is the lowest priority exception / interrupt. This allows further interrupts to occur.

CHAPTER 6

Thread Switching

RANDALL MAAS This chapter discusses switching between threads.

- Introduction
- The thread switching
- Cooperative switchers
- Pre-emptive switchers

13. INTRODUCTION

This chapter discusses the switching between threads. Modern, small real-time systems will have few threads. Most of their work will be done by peripherals (and their associated interrupts), and a core thread.

Threads are switched between in one of three circumstances:

- A thread is waiting for a timer to expire
- A timer has expired and woken a thread
- An IPC object has changed state, waking the highest priority thread waiting on it. The elements of an IPC resource will be examined in more detail in the next chapter.

13.1. KINDS OF SWITCHER

There are two broad categories of context switching can be cooperative or pre-emptive. The cooperative method can switch contexts with:

- setjmp/longjmp
- Green threads (aka proto-threads) that employ a callback procedure
- State machine

The pre-emptive methods include:

- CPU interrupt pre-emption and changing return context
- Emulation

13.2. PRIORITY INHERITANCE

A thread is waiting for a resource held by a lower priority thread. The fix is to boost the priority of the thread that holds the resource. To keep it constant time, mutexes are assigned a priority. The mutex must have a higher priority than any of the threads that may pend on it. No thread can have the same priority as a mutex. When a thread successfully pends on the mutex, its effective priority is raised to that of the mutex. When it releases the mutex, the

threads priority returns to the highest priority of the mutex's it holds, or (if no mutexes are held) it's own.

This implementation has the advantage of being constant time. If a mutex did not have a priority assigned to it, it would be a complex task to manage the priority of multiple resources, and the chains of escalated privilege.

14. THREAD SWITCHING

The thread switcher

1. If there are no further threads ready, go to the idle thread. Otherwise
2. Get highest priority thread id
3. If this id is for a mutex, get the mutex
4. Get the thread control pointer for this thread
5. Resume the thread

Threads can, within the code, use the following operations (macros). Many of these are defined by the particular thread-switching technique:

***Table 4:** Thread management functions*

Operation	Description
ThrDelay()	Delays the thread's execution for the specified duration.
ThrExit()	The thread should no longer be called.
ThrYield()	Let another thread run, then resume running this thread.

The thread switcher has the following operations (macros) that it employs:

***Table 5:** Thread switching functions*

Operation	Description
ThrResume()	Called to pass control to the given thread

Timers and IPC mechanisms have the following operations (macros) that they may employ:

***Table 6:** IPC thread management functions*

Operation	Description
KnlReschedule()	Triggers the kernel to switch threads. This is similar to ThrYield(), but does so in a manner that is safe for use in timers, interrupts and other exception handlers.
ThrEffectivePriority()	Typically the priority the thread is scheduled at. A thread may be scheduled at a higher priority if an IPC object it holds is being pended upon by a higher priority task.
ThrTimeout()	A timeout handler, for IPC objects. When it expires, it sets the thread error to a "timeout error" and wakes the thread to process the error, using ThrWake().
ThrWaitFor()	Wait for the IPC mechanism, such as a mutex, semaphore, queue, or mailbox, to be ready.
ThrWake()	This is used to move the thread to the waiting-for-the-processor list. It cancels any timer, and removes the item from the IPC waiting list. It sets the threads waitingOn field to null.
ThrWakeup()	A timeout handler, for pausing a thread. When it expires, it wakes the thread, using ThrWake().

The thread control (TCB) structure for threads in this model includes:

Field	Description
effectivePriorities	The priorities of the thread and the mutexes it holds.
error	Any error code that is to be delivered to the thread. This primarily is a timeout while waiting for a resource.
owner	The thread id. This is used for compatability with the mutex structure, which uses this to map back to the owning thread when said threads effective priority has been escalated to that of the mutex.
timer	A timer used to bound the period we wait on IPC objects, or delays the thread's execution for a period. When this timer expires, it gives up pending on the waitingOn object (if any), sets a reason for waking the thread, and
waitingOnCancel	The handler to call when the thread is woken, so that the IPC resource can be "cleaned up" or unlinked from the current waiting thread.
waitingOn	If set, the IPC resource (e.g. mutex, semaphore, queue, mailbox, etc) that the thread is waiting on.
extra	Threading-mechanism specific data structures.

Table 7: *Thread control block*

The implementations that are specific to the threading technique will be covered in their respective sections. The core operations above have the following code implementation:[2]

Operation	Implementation
ThrDelay(duration)	`TmrDelayFunc(TmrWakeup, duration);` `ThrYield();`

ThrDelayFunc(hdlr,duration)	`// Remove from the CPU's ready list`
	`ThrRemove(&PCB, effectivePriority);`
	
	`// Attach a timer`
	`if (~0uL != duration)`
	`{`
	`  // Initialize the timer`
	`  TmrStart(&TCBPtr->timer, duration, func, hintV(currentThreadId));  }`

ThrEffectivePriority()	`ThrNext(TCBPtr->effectivePriorities)`

ThrTimeout(timer, hint)	`// Get the thread id for the timer`
	`TCB_t* TCBPtr = &TCB[hint.v];`
	
	`// Mark the thread as having an error`
	`TCBPtr->error = ErrTimeout;`
	
	`// Wake the thread for further execution (e.g. to receive the error)`
	`ThrWake(hint.v);`

ThrWaitFor(effectivePriority, waitCancel, waitOn, duration)	`TCB_t* TCBPtr = TCB + currentThreadId;`
	
	`// Set how we can cancel a wait, as the thread is woken`
	`TCBPtr->waitingOnCancel = waitCancel;`
	`TCBPtr->waitingOn = waitOn;`
	
	`// Mark the thread as pending`
	`TCBPtr->error = ErrPending;`

Table 8: *Thread functions implementation*

[2] Note: as these are implemented as macros, they are wrapped in a do{…}while(0) to protect them from interacting unexpectedly with the control flow. This also assumes that there are a small number of priorities in the system (less than the word size)

```
                        // Add the time out, and remove from the run ready list
                        ThrDelayFunc(TCBPtr, effectivePriority, ThrTimeout, duration);

ThrWake(threadId)       // Look up the thread
                        TCB_t* TCBPtr = TCB + threadId;

                        // Cancel the timer and clear out the linkage that this is the threads
                        // special timer
                        TmrCancel(TCBPtr->timer);

                        // Look up its effective priority
                        uint32_t effectivePriority = ThrEffectivePriority(TCBPtr);

                        // Remove the thread from the IPC's waiting list
                        TCBPtr->waitingOnCancel(TCBPtr->waitingOn, threadId, effectivePriority);
                        TCBPtr->waitingOnCancel = doNothing;

                        // Mark the thread as ready to run
                        ThrAdd(&PCB, effectivePriority);

                        // Trigger a rescheduling of tasks
                        KnlReschedule();

ThrWakeup(timer, hint)  // Wake the thread for further execution (e.g. to receive the error)
                        ThrWake(hint.v);
```

15. COOPERATIVE SCHEDULERS

15.1. SETJMP/LONGJMP

The first cooperative threading technique we'll examine uses `setjmp()/longjmp()`. This technique is not recommended for use in practice – I'll explain why soon – but it is helpful to understand the basics of threads and thread switching.

Advantages	Compact memory usage. Looks natural.
Disadvantages	This technique is unsafe.
Other	

Table 9: *setjmp/longjmp cooperative threading strengths and weaknesses*

The thread control (TCB) structure is extended, for threads in this model, include to include the following field:

Field	Description
jmpbuf	A structure holding key register values and program counter to resume.

Table 10: *Thread control block for setjmp/longjmp*

The operations above have the following code implementation:[3]

[3] Note: as these are implemented as macros, they are wrapped in a do{…}while(0) to protect them from interacting unexpectedly with the control flow.

Operation	Implementation

ThrResume(TCBPtr)	if (!setjmp (KnlSwtchBuf))
	{
	  longjmp(TCBPtr->jmpbuf, 1);
	}

ThrYield(TCBPtr)	if (!setjmp(TCBPtr->jmpbuf))
	{
	  longjmp(KnlSwtchBuf,1);
	}

***Table 11:** Thread functions implementation*

When this type of threading is used, the threads are prepared in a initialization step. This step prepares the KnlSwtchBuf, allocates stack space for the kernel thread, prepares the first threads jmpbuf, allocates space on the stack, and repeats these last two steps for all of the threads.

15.1.1 Why to not use setjmp/longjmp

I recommend not using this approach to implement threads. Why? Setjmp/longjmp are dangerous. ISO C standard (7.13.2.1, "The longjmp function") says:

> All accessible objects have values as of the time longjmp was called, except that the values of objects of automatic storage duration that are local to the function containing the invocation of the corresponding setjmp macro that do not have volatile-qualified type and have been changed between the setjmp invocation and longjmp call are indeterminate.

15.2. CALLBACK BASED THREADS (AKA GREEN THREADS, PROTOTHREADS, ETC.)[4]

The next cooperative threading technique is to employ call backs to a procedure (or nested set of procedures) that track their state in the work flow.

Advantages	Compact memory usage. It is the most portable approach, and often the fastest context switch.
Disadvantages	There are many restrictions, and some ceremony required. Local variables and calling parameters are not allowed. Debugging is trickier due to these.
Other	It is possible to create tools convert a restricted C code to the form used here.

***Table 12:** Callback cooperative threading strengths and weaknesses*

A handler is called every time the procedure is given a time slice. The handling procedure looks like:

```
void handler(TCB_t* TCBPtr)
{
  ThrBegin
        user code
  ThrEnd
}
```

***Figure 9:** Basic outline of a green thread procedure*

Within the code, thread can use the operations mentioned earlier (reference) and the following operations (macros):

[4] The technique shown here does not work with Microsoft Visual C. Microsoft's "edit and continue" feature (an otherwise impressive and useful tool) interacts with the __LINE__ "macro", meaning that it could change at run time while edited. This makes it ineligible for use with case: statements and use a state.

Operation	Description
ThrCall()	Calls a next level handler in the thread. This is necessary if the procedure being called will use any of these thread functions.
ThrReturn()	Returns from this thread handler to the previous calling handler.

Table 13: *Thread functions*

The thread control (TCB) structure is extended, for threads in this model, to include the following fields:

Field	Description
stateBuf	A buffer (in stack discipline) to hold the local state.
statePtr	A pointer to the state to return control too.
stateHandlerBuf	A buffer (in stack discipline) to hold the handler to call.
stateHandlerPtr	A pointer to the handler to return control too.

Table 14: *Thread control block for callback*

The local state is replacement for the conventional C-stack. It is used to hold the local variables and where to resume execution (i.e., it holds the activation records):

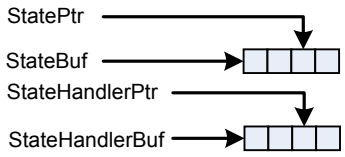

Figure 10: *Thread control block*

The individual state structure holds the following information:

Field	Description
	Pointer to the current handling procedure.
	Some sort of line number or other hint to continue execution in the current state.
	Locals – the local variable values.

Table 15: *Thread local state*

The operations above have the following code implementation:[5]

Operation	Implementation

ThrBegin	`{`
	`  uint16_t* statePtr=&TCBPtr->state;`
	`  switch(statePtr[0])`
	`  {`
	`  default:;`

ThrCall(x)	`*++(TCBPtr->statePtr)  = 0;`
	`*++(TCBPtr->stateHandlerPtr) = x;`
	`ThrWake(currentThreadId);`
	`statePtr[0] = (uint16_t) __LINE__; (x)(TCBPtr);return;} case __LINE__:;`

ThrEnd	`  }` `  statePtr[0] = 0;`

Table 16: *Thread functions implementation*

[5] Note: as these are implemented as macros, they are wrapped in a do{…}while(0) to protect them from interacting unexpectedly with the control flow.

	ThrReturn(); }
ThrResume()	TCBPtr->stateHandler(TCBPtr);
ThrReturn()	TCBPtr->statePtr--; TCBPtr->stateHandlerPtr--; return;
ThrYield()	{statePtr[0] = (uint16_t) __LINE__; ThrWake(currentThreadId); return;} case __LINE__:;

16. PRE-EMPTIVE SCHEDULERS

The pre-emptive schedulers have a unique implementation for each processor family. This section does not cover them all. Emulating the features of a microprocessor on Windows is covered in another fascicle.

16.1. ARM CORTEX-M SERIES

The table below gives a sketch of the implementation for the ARM Cortex-M family.

Table 17: *ARM Cortex functions implementation*

Operation	Implementation
IntDisable()	Based on *__disable_irq()*
IntEnable()	Based on *__enable_irq()*
ThrResume(TCBPtr)	*PendSV swizzles the registers so that it's return to main execution is to the newly scheduled thread context*
ThrYield()	*// Queue PendSV exception* SCB->ICSR |= SCB_ICSR_PENDSVSET_Msk;

17. OTHER TASK SWITCHERS TO STUDY

The "XV6" project from MIT are useful for examining other basic operating system kernels. It seeks to replicate the Unix System v6, but supported on x86 platforms. (It is intended to run in a virtual machine in most cases).

http://pdos.csail.mit.edu/6.828/2016/xv6.html

CHAPTER 7

IPC Mechanisms

RANDALL MAAS This chapter is describes the inter-process communication (IPC) mechanisms:

- Introduction
- Flags, mutex and queue implementation

18. INTRODUCTION

The previous chapter provided the foundational details for switching between threads. This chapter is about how a thread can be prevented from running, and how a blocked thread can be woken.

The IPC mechanisms described in this chapter are:

- Flags, which allow an interrupt handler to wake the thread that is pending on it.
- Queues. This allows one thread to send messages – data – to another thread.
- Mutexes which are used to protect critical sections or resources within a thread. These are preferred when the thread would hold the resource for a time longer than would acceptable to disable interrupts. This allows only one thread to modify data structures or use a peripheral at a time.

The pattern in the API for these mechanisms is:

- To request a resource, *pend* on it. If the resource is available, a pend returns immediately. If not, the thread is block until a time out or the resource is acquired.
- To release a resource (or pass it) *post* to it.

18.1. A NOTE ON OTHER "SOPHISTICATED" IPC MECHANISMS

In most discussions of inter-process communication, a mechanism called a *semaphore* is mentioned. The only type of semaphore here is the "flag" mechanism, which may be called a *binary semaphore.* Why was a flag chosen rather than a more general semaphore mechanism?

- There rarely is a need for semaphore features beyond those that a flag provides.
- Semaphores can overflow
- Semaphores have wake issues

19. FLAGS

Flags will be examined first, as they are the simplest and the most useful. Flags are used to wake a thread. That is

- An interrupt handler can wake a thread, letting it know that an event occurred
- A thread can wake another thread

Pend and Post operations are implemented in two parts: a macro and a procedure. The top level is a macro that calls the internal implementation for the pend (or post). Then the macro calls ThrYield(). This is necessary, as some implementations of ThrYield() must be invoked in this way in order to to properly transition to the task switcher.

The threads *error* field is used to signal the result of the work. On success, the error field will be cleared. On error, the timeout handler will set the field to ErrTimeout (and call the cancel handler) when it expires. When the thread resumes, after being woken by either of the above, it will retrieve the error value.

Flags can have only a single waiter. Any thread can post to it.

***Table 18:** Flag management functions*

Operation	Description
FlgCancel()	This clears any references the flag has to waiting threads.
FlgPend()	Wait for a flag to be raised (if it isn't already).
FlgPost()	Raise the flag, and wake the thread that is waiting on it (if any).

The structure for flags in this model is:

***Table 19:** Flag control block*

Field	Description
value	Holds the value; this is only used when the flag is raised without any waiters.
waiter	The thread that is waiting for the flag to be raised.

The operations have the following code implementation:

***Table 20:** Flag functions implementation*

Operation	Implementation
FlgCancel()	((Flag_t*) hint.p) -> waiter = 0;
FlgPend(flag,timeout,err)	if (!_FlagPend(flag, timeout)) ThrYield(); if (err) (err)[0] = TCBPtr->error;

_FlgPend(flag,timeout)	IntDisable();
	
	// *If the flag is set*
	if (flag->value)
	{
	  // *Clear the flag*
	  flag->value = 0;
	  TCB[currentThreadId].error = ErrNone;
	  IntEnable();
	  return 1;
	}
	// *Add self to the waiter list*

```
                    flag->waiter = currentThreadId;

                    // And set how we can cancel a wait, as the thread is woken
                    // Get the threads effective priority
                    uint32_t effectivePriority = ThrEffectivePriority(TCBPtr);

                    // Set it to wait for the time period
                    ThrWaitFor(effectivePriority, FlgCancel, hintP(flag), timeout);
                    IntEnable();
                    return 0;
FlgPost(flag)       _FlgPost(flag);
                    ThrYield();

_FlgPost(flag)      IntDisable()

                    // Cancel timer & wake the thread waiting on it
                    if (flag->waiter)
                    {
                      // Indicate the thread got the resource
                      TCB[flag->waiter] . error = ErrNone;

                      // Wake the thing we are waiting on
                      ThrWake(flag->waiter);

                      // We clear the value, as the nature of waking up consumes it
                      flag->value = 0;
                    }
                    else
                    {
                      // There is no waiter, so set the value
                      flag->value = 1;
                    }
                    IntEnable();
```

20. QUEUES

Queues are very similar to flags. Like flags, any thread can post to it, but only one can be pending on it.

Operation	Description
QCancel()	This clears any references the queue has to waiting threads.
QPend()	Wait for a queue to have a message (if it doesn't already).
QPost()	Put a message in the queue, and wake the thread that is waiting on it (if any).

***Table 21:** Message queue management functions*

The structure for queue in this model is:

Field	Description
inQueue	The incoming list of messages.
queue	The list of messages being processed
waiter	The thread that is waiting for the queue to contain messages.

***Table 22:** Message queue control block*

Note that there are two lists in a queue. The incoming list is appended to by the posting thread/interrupt. When the pending thread's second list is empty it moves the first list to the second, and dequeues items from that. This way interrupts are disabled less frequently – only when the second list is empty, rather than during each QPend(). This is especially useful during *bursts* where items may be enqueue rapidly.

The operations have the following code implementation:

***Table 23:** Queue functions implementation*

Operation	Implementation
QCancel()	((Q_t*) hint.p) -> waiter = 0;

QPend(Q,timeout,err)	if (!_QPend (Q,timeout))
	{
	  ThrYield();
	
	  if (err) (err)[0] = TCBPtr->error;
	  if (ErrNone == TCBPtr->error) ret = _QPend(Q,0);
	}

_QPend(Q,timeout)	ret = Q->queue;
	if (!ret)
	{
	  IntDisable();
	  if (Q->inQueue.next != &Q->inQueue)
	  {
	  ret = Q->inQueue.next;
	  Q->inQueue.prev->next = NULL;
	  Q->inQueue.next = &Q.inQueue;
	  Q->inQueue.prev = &Q.inQueue;
	  }
	  IntEnable();
	}
	
	if (ret)
	{
	  Q->queue = ret->next;
	  ret->next = NULL;
	  ret->prev = NULL;
	  TCB[currentThreadId].error = ErrNone;
	  return ret;
	}
	
	if (timeout)
	{
	  Q->waiter = currentThreadId;
	  uint32_t effectivePriority = ThrEffectivePriority(TCBPtr);
	  ThrWaitFor(effectivePriority, QCancel, hintP(Q), timeout);
	}
	return 0;

QPost(Q,msg)	_QPost(Q,msg); ThrYield();

_QPost(Q,msg)	IntDisable();
	LstPrepend(&Q->inQueue,msg);
	if (Q->waiter)
	{
	  TCB [Q->waiter] . error = ErrNone;
	  ThrWake(Q->waiter);
	}

```
IntEnable();
```

21. MUTEXES

Mutexes are the most complex construct here. They are used to ensure that only one thread is modifying a resource, such as a communication channel or memory structure. (This is called serializing access.) Mutexes are used when disabling interrupts is not practical or would be disabled too long; mutex access also disables interrupts for a brief period, hence the gradation

Operation	Description
MtxCancel()	This clears any references the mutex has to waiting threads.
MtxPend()	Wait for a mutex to be available (if it doesn't already).
MtxPost()	Put a message in the queue, and wake the thread that is waiting on it (if any).

***Table 24:** Mutex management functions*

The structure for mutex in this model is:

Field	Description
owner	The current owning thread of the mutex; this is used to map back to the thread when its effective priority has been escalated to that of the mutex.
waitingList	The prioritize set of threads that are waiting for the mutex to be raised. The thread sets its *effective priority* in this

***Table 25:** Mutex control block*

The operations have the following code implementation:

Operation	Implementation

MCancel()	`Mutex_t* waitingOn = hint.p;`
	`ThrRemove(&waitingOn->waitingList, threadId);`
	`ThrRemove(&waitingOn->waitingList, effectivePriority);`

MtxPend(mutex, timeout, err)	`if (!_MtxPend (mutex,timeout))`
	`{`
	`  ThrYield();`
	`  if (err) (err)[0] = TCBPtr->error;`
	`}`

_MtxPend(mutex, timeout)	`Mutex_t* mutex = (Mutex_t*)(TCB + mutexId);`
	`uint8_t ret = 0;`
	`TCB_t* TCBPtr = TCB + currentThreadId;`
	
	`// If there is no owner, or we are the owner`
	`if (!mutex->owner || mutex->owner == currentThreadId)`
	`{`
	`  // Set ourselves as the owner`
	`  mutex->owner = currentThreadId;`
	
	`  // Remove ourselves from the processor's run list at the current`
	`  // priority level`
	`  ThrRemove(&PCB, ThrEffectivePriority(TCBPtr));`
	
	`  // Bump the threads effective priority`
	`  ThrAdd  (&TCBPtr->effectivePriorities, mutexId);`

***Table 26:** Queue functions implementation*

```
    // Add ourselves back the processor's run list at the new effective
    // priority level
    ThrAdd      (&PCB, ThrEffectivePriority(TCBPtr));

    // We were a success, so don't do any errors
    TCBPtr->error = ret = ErrNone;
  }
  else
  {
    // The threads effective priority
    uint32_t effectivePriority = ThrEffectivePriority(TCBPtr);

    // Add self to the waiter list
    ThrAdd(&mutex->waitingList, effectivePriority);

    // Wait, setting a timeout
    ThrWaitFor(effectivePriority, MtxCancel, hintP(mutex), timeout);
    ret = ErrPending;
  }

  // Return the error state
  return ret;
```

MtxPost(mutex)

```
  _MtxPost(mutex);
  ThrYield();
```

_MtxPost(mutex)

```
  Mutex_t* mutex = (Mutex_t*)(TCB + mutexId);

  // Reduce the sending threads effective priority
  ThrRemove(&TCB[currentThreadId].effectivePriorities, mutexId);
  ThrRemove(&PCB, mutexId);

  // Skip if there are no waiters
  if (mutex->waitingList.level0)
  {
    // Get the highest priority thread that is waiting
    // This removes it from the waiting list
    uint32_t threadId = ThrNext(&mutex->waitingList);

    // Look up the thread
    TCB_t* TCBPtr = TCB + threadId;

    // Handle the case the threadId was a mutex
    TCBPtr = TCB + TCBPtr->owner;

    // Add the mutex to the list of resources that the it owns
    ThrAdd(&TCBPtr->effectivePriorities, mutexId);

    // Indicate the thread got the resource
    // This overwrites any error (eg ErrPending or ErrTimeout)
    TCBPtr->error = ErrNone;

    // Mark the thread as ready to run
    ThrWake(TCBPtr->owner);
  }
```