7 Concurrent Software

Contents

7.2

Introduction

A thread is an executing program, with a context

A program is a list of instructions for the computer to execute. A *thread* is an executing program, including the current values of the program counter, registers and variables. A thread has an execution state (such as running, ready, waiting) and a saved thread context when not running. Conceptually, each thread has its own CPU. In reality, of course, the real CPU switches back and forth from thread to thread.

A process is a multiprogramming concept

By contrast, a *process* is a conceptual entity used when dealing with multiple programs running on a general purpose computer (such as a PC with Windows®). A process has its own virtual address space and has protected access to the CPU, other processes, files, and I/O resources.

Simple embedded systems only have threads

In an embedded system, there is no need for multiple programs to be executing, since the embedded system is usually designed for a specific application. Thus, in an embedded system there is no need for the concept of a process and we will deal exclusively with threads.

Simple embedded systems are foreground / background systems

The execution of the main program is called the *background* thread. In most embedded applications, the background thread executes a loop that never ends. This thread can be broken (execution suspended, then restarted) by *foreground* threads (interrupt service routines). These threads are run using a simple algorithm. The ISR of an input device is invoked when new input is available. The ISR of an output device is invoked when the output device is idle and needs more data. Last, the ISR of a periodic task is run at a regular rate. The main program runs in the remaining intervals. Many embedded applications are small in size, and static in nature, so this configuration is usually adequate.

The limitation of a single background thread comes as the size and complexity of the system grows. Projects where the software modules are loosely coupled (independent) more naturally fit a multiple background thread configuration.

A *scheduler* is a piece of software that implements multiple background threads, and forms the basis of a program known as an *operating system* (OS). Synchronization tools that allow threads to interact with each other (such as *semaphores*) are also a key feature of operating systems.

A scheduler lets us implement multiple background threads

Systems that implement a thread scheduler still may employ regular I/O driven interrupts. In this way, the system supports multiple foreground threads and multiple background threads.

7.1 Threads

A *thread* is the execution of a software task that has its own stack and registers. Since each thread has a separate stack, its local variables are private, which means it alone has access.

Each thread has its own registers and stack

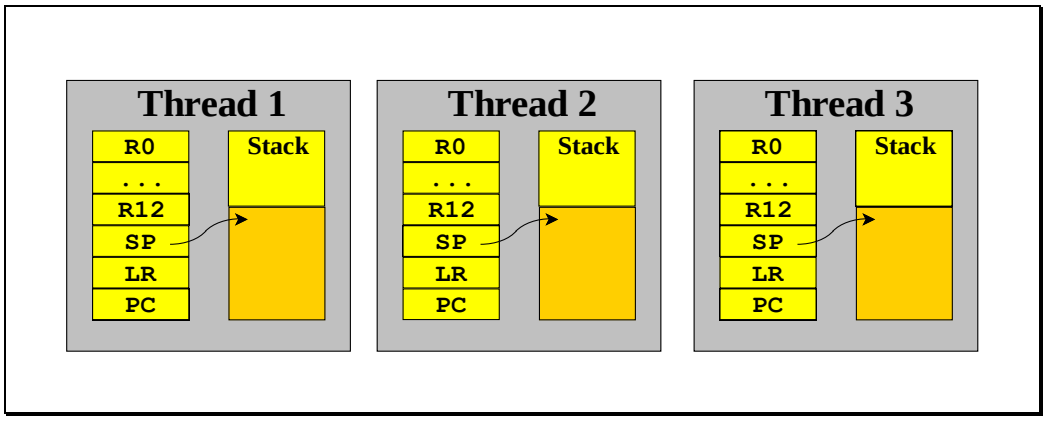

Figure 7.1

Multiple threads cooperate to perform an overall function. Since threads interact for a common goal, they do share resources, such as global memory, and I/O devices.

Threads share global memory and I/O ports

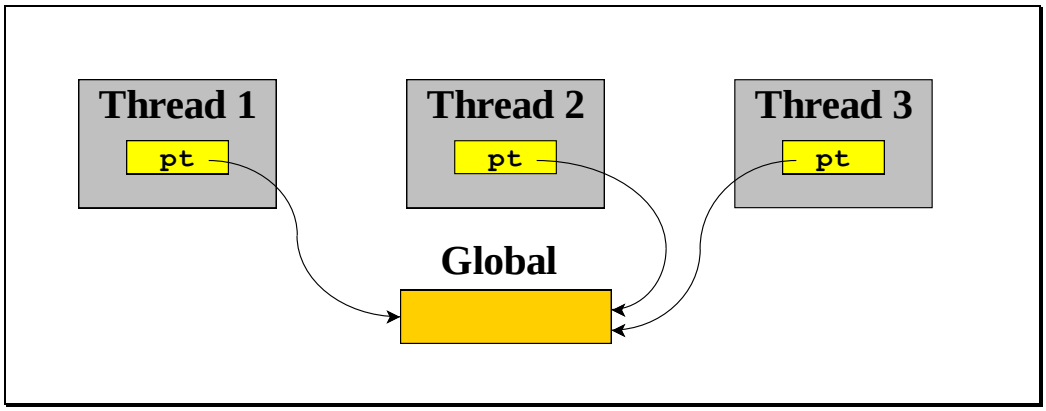

Figure 7.2

In summary, a thread:

- is the execution of a software task
- has its own stack and registers
- has local variables which are private
- cooperates to perform an overall function

A thread can be in one of three states.

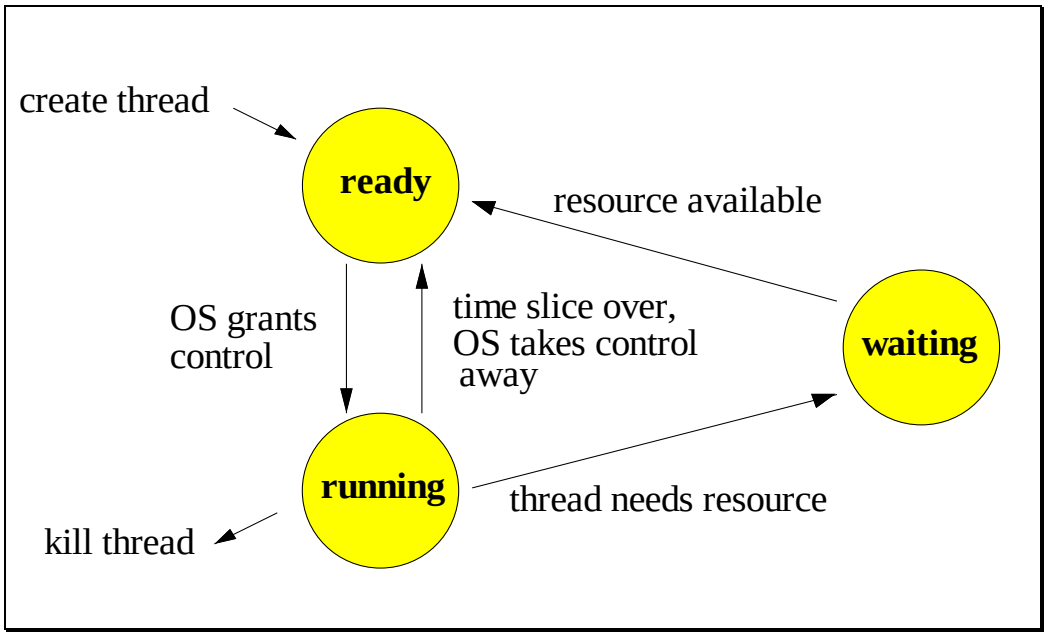

Figure 7.3

A thread is in the *ready* state if it is ready to run but waiting for its turn.

A thread is in the *running* state if it is currently executing. With a single instruction stream (i.e. one core) processor like the K70, at most one thread can be in the run state at a time.

A thread is in the *waiting* state when it is waiting for some external event like I/O (keyboard input available, printer ready, I/O device available). If a thread communicates with other threads, then it can be waiting for an input message or waiting for another thread to be ready to accept its output message. If a thread wishes to output to the serial port, but another thread is currently outputting, it will wait. If a thread needs information from a FIFO (calls `FIFO_Get`), then it will wait if the FIFO is empty (because it cannot retrieve any information). On the other hand, if a thread outputs information to a FIFO (calls `FIFO_Put`), then it will wait if the FIFO is full (because it cannot save its information).

7.6

An OS may use a linked list data structure to hold the ready and waiting threads. It may create a separate waiting linked list for each reason why the thread cannot execute. For example, one waiting list for “full” during a call to **`FIFO_Put`**, and one for “empty” during a call to **`FIFO_Get`**. In general, the OS could have one waiting list associated with each cause of waiting.

In the figure below, thread 5 is running, threads 1 and 2 are ready to run, and threads 3 and 4 are waiting because a FIFO is empty.

Threads are placed in linked lists depending on whether they are ready or waiting

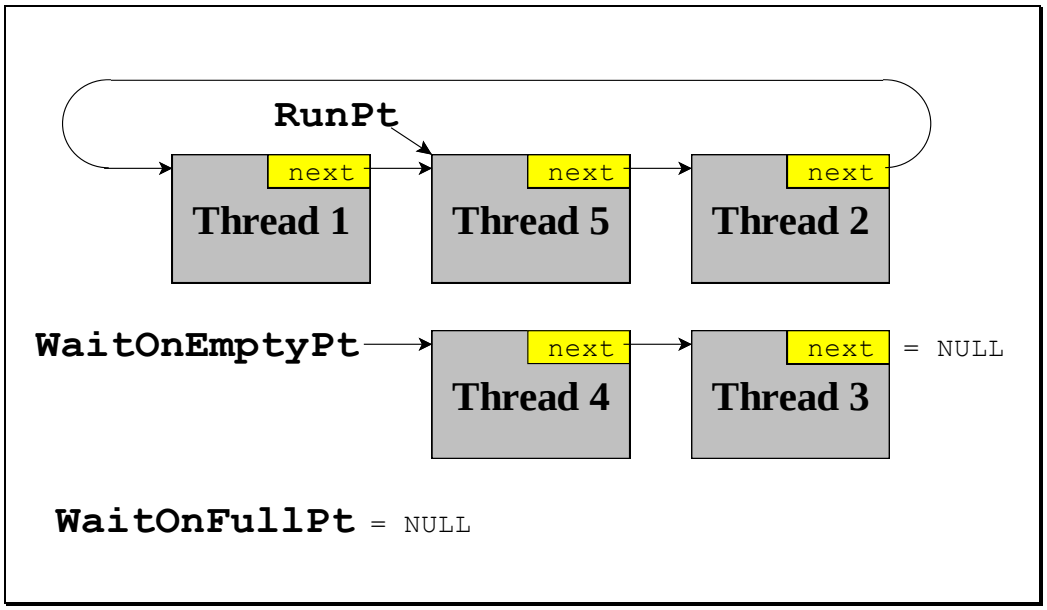

Figure 7.4

7.1.1 Thread Control Blocks (TCBs)

If a thread is ready, it may be granted control of the CPU by the OS at any time. Conversely, while running, the OS may stop the thread executing and make it ready. We therefore need a way for the scheduler to save and restore the state of a thread. A *thread control block* (TCB) is used to store the information about each thread.

The TCB must contain:

1) a pointer to its private stack;
2) a pointer so it can be chained into a linked list;
3) a stack area that includes local variables

Essential thread control block components

While a thread is running, it uses the actual hardware registers, `R0-R15`. In addition to these necessary components, the TCB might also contain:

4) Thread number, type, or name;
5) Age, or how long this thread has been active;
6) Priority;
7) Resources that this thread has been granted.

Optional thread control block components

The structure of a typical TCB is shown below:

TCB structure

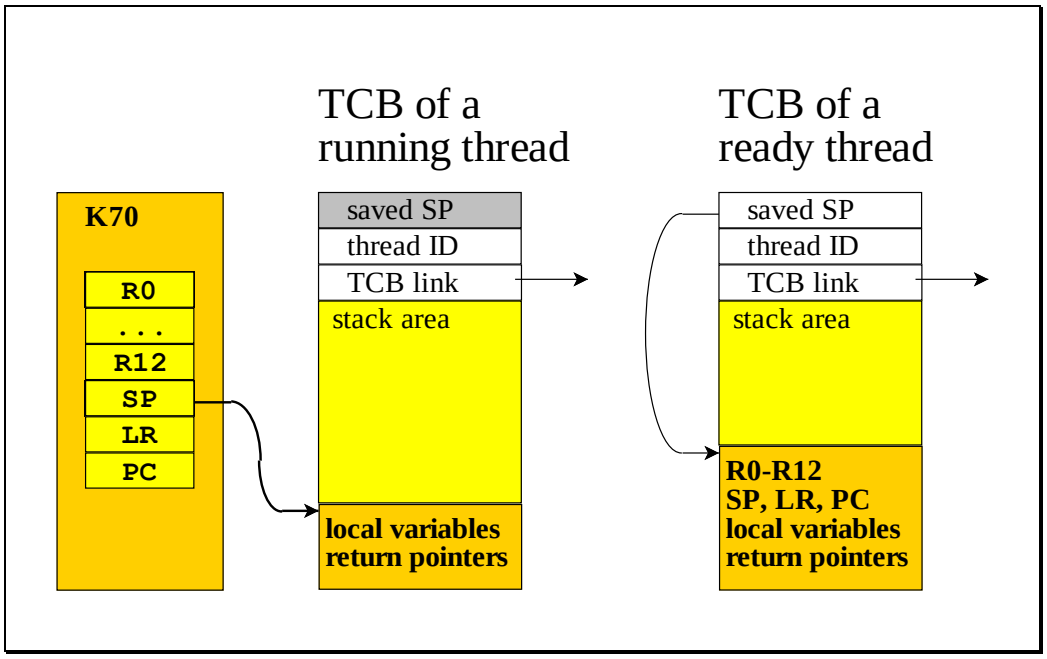

Figure 7.5

The running thread uses the actual registers, while the other threads have their register values saved on the stack.

7.2 Schedulers

A scheduler is responsible for changing the running thread

A *scheduler* is an OS component that has responsibility for switching threads between states. A scheduler has to implement two aspects of this operation. One aspect is to save the currently running thread's state in its TCB and to restore the state of the next thread to run (the process of changing threads, which is also called a context switch). The other aspect is *when* the scheduler actually changes threads, and what it does with waiting threads.

In a preemptive scheduler, the OS interrupts each thread regardless of whether the thread is "in the middle of something important" – the OS is the sole arbiter of when the thread will actually get CPU time.

A round-robin scheduler runs each thread in a fixed order for a certain amount of time

The simplest scheduling system is a round-robin scheduler – a scheduler that runs each "ready" thread for a certain amount of time in a fixed cyclic order. It does this by "hooking" into a periodic timer whose ISR performs the thread changeover function.

EXAMPLE 7.1 Round-Robin Shceduler

Suppose we have three dynamically allocated threads that are executed in a round-robin fashion. Even though there are three threads, there are only two programs to run, **ProgA** and **ProgB**. Recall that a thread is not simply the software but the execution of the software. We will have two threads executing the same program, **ProgA**.

```
void main(void)
{
  OS_AddThread(&ProgA);
  OS_AddThread(&ProgA);
  OS_AddThread(&ProgB);
  OS_Start(TIMESLICE); // doesn't return
}
```

A circular linked list allows the scheduler to run all three threads equally.

Round-robin scheduling

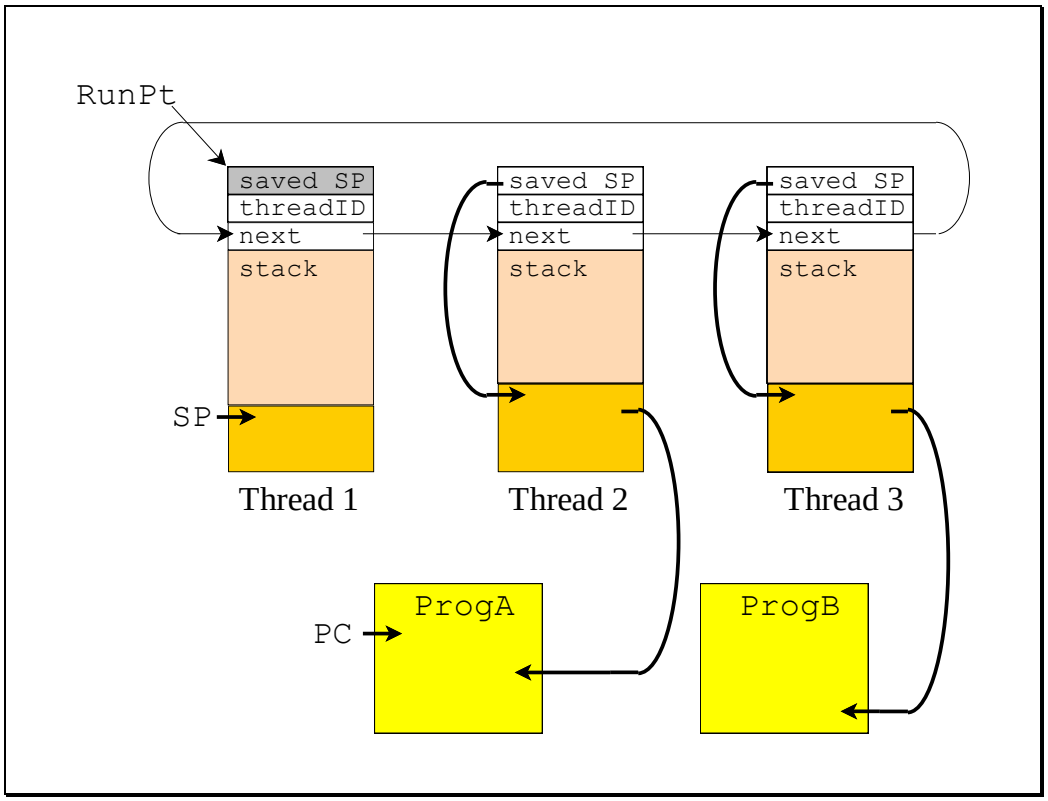

Figure 8.1

This example illustrates the difference between a program (e.g. **ProgA** and **ProgB**) and a thread (e.g. Thread 1, Thread 2 and Thread 3). Notice that Threads 1 and 2 both execute **ProgA**. There are many applications where the same program is being executed multiple times.

7.2.1 Other Scheduling Algorithms

Non-preemptive scheduling

A non-preemptive (cooperative) scheduler trusts each thread to voluntarily release control on a periodic basis. Although easy to implement, because it doesn't require interrupts, it is not appropriate for real-time systems.

Priority scheduling

A priority scheduler assigns each thread a priority number (e.g. 0 is the highest, 15 is the lowest). Two or more threads can have the same priority. A priority 1 thread is run only if no priority 0 threads are ready to run. Similarly, we run a priority 2 thread only if no priority 0 or priority 1 threads are ready. If all threads have the same priority, then the scheduler reverts to a round-robin system. The advantage of priority is that we can reduce the latency (response time) for important tasks by giving those tasks a high priority. The disadvantage is that on a busy system, low-priority threads may never be run. This situation is called *starvation*.

7.3 The SysTick Timer

An overview of the SysTick timer

The Cortex®-M processors have a small integrated timer called the SysTick (System Tick) timer. It is integrated as a part of the NVIC and can generate the SysTick exception (exception type #15). The SysTick timer is a simple decrement 24-bit timer, and runs off the processor's core clock.

In modern operating systems, a periodic interrupt is needed to ensure that the OS *kernel* can be invoked regularly for task management and context switching. This enables a processor to handle different tasks in different time slots. The SysTick timer is similar to the PIT module:

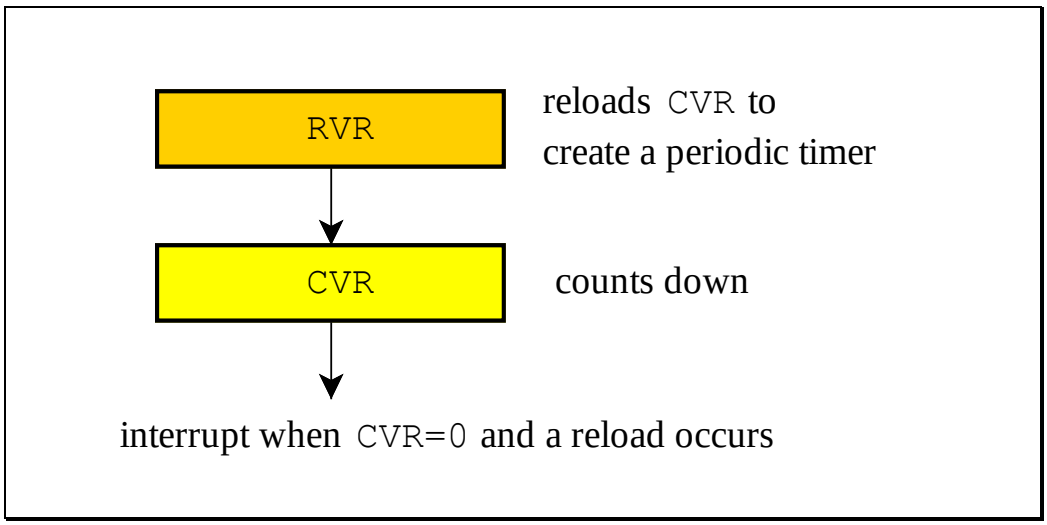

Figure 7.6

There is a current value CVR and a reload value RVR. When the SysTick counter is enabled, the CVR decrements every clock cycle. If it reaches zero, it will then load the value from RVR and continue. If the SysTick interrupt is enabled, it will generate an interrupt when it reloads the value.

There is a Control and Status Register (CSR) that allows us to control and check the status of the SysTick timer. To generate a periodic interrupt using SysTick, we need to:

- set the RVR reload value for the desired period
- set CSR bit 0 to a 1 to enable the SysTick timer
- set CSR bit 1 to a 1 to enable interrupt generation

7.4 A Simple Operating System

The following example shows how a simple operating system can be created using the SysTick timer, a simple TCB structure, and some low-level assembly language context switching.

EXAMPLE 7.2 Simple Operating System

Suppose we have three statically allocated threads that are each allowed to execute for 250 ms in a round-robin fashion. Even though there are three threads, there are only two programs to run, **ProgA** and **ProgB**. We will have two threads executing the same program, **ProgA**, and one thread executing **ProgB**. The code for these programs is shown below.

```
int Sub(int j)
{
  int i;
  i = j + 1;
  return (i);
}

void ProgA(void)
{
  int i;
  i = 5;
  while (1)
  {
    i = Sub(i);
  }
}

void ProgB(void)
{
  int i;
  i = 6;
  while (1)
  {
    i = Sub(i);
  }
}
```

Notice that both threads call the same subroutine **Sub()**.

The thread control block is defined in `OS.h` as follows:

```
// Thread control block structure
struct TCB
{
  unsigned long *savedSP;         // saved stack pointer
  unsigned long threadId;         // 1, 2, 3, ...
  struct TCB *nextTCB;            // link to next TCB
  unsigned long moreStack[99];    // the free stack
  unsigned long stackedR4;        // the ARM "stack frame" when
  unsigned long stackedR5;        // an exception occurs ...
  unsigned long stackedR6;
  unsigned long stackedR7;
  unsigned long stackedR8;
  unsigned long stackedR9;
  unsigned long stackedR10;
  unsigned long stackedR11;
  unsigned long stackedR0;
  unsigned long stackedR1;
  unsigned long stackedR2;
  unsigned long stackedR3;
  unsigned long stackedR12;
  unsigned long stackedLR;
  void (*stackedPC)(void);
  unsigned long stackedxPSR;
};

typedef struct TCB TCBType;
typedef TCBType* TCBPtr;
```

The **main**() module statically declares the threads as global variables:

```
TCBType Threads[3] __attribute__ ((aligned(0x08))) =
{
  {
    &Threads[0].stackedR4,  // initial SP
    1,                      // thread ID
    &Threads[1],            // pointer to next TCB
    { 0 },                  // empty stack
    0, 0 ,0 ,0, 0, 0, 0, 0, // R4-R11
    0, 1, 2, 3, 12, 0,      // R0-R3, R12, LR
    ProgA,                  // PC
    0x01000000              // xPSR
  },
  {
    &Threads[1].stackedR4,  // initial SP
    2,                      // thread ID
    &Threads[2],            // pointer to next TCB
    { 0 },                  // empty stack
    0, 0 ,0 ,0, 0, 0, 0, 0, // R4-R11
    0, 1, 2, 3, 12, 0,      // R0-R3, R12, LR
    ProgA,                  // PC
    0x01000000              // xPSR
  },
  {
    &Threads[2].stackedR4,  // initial SP
    3,                      // thread ID
    &Threads[0],            // pointer to next TCB
    { 0 },                  // empty stack
    0, 0 ,0 ,0, 0, 0, 0, 0, // R4-R11
    0, 1, 2, 3, 12, 0,      // R0-R3, R12, LR
    ProgB,                  // PC
    0x01000000              // xPSR
  }
};
```

Even though the threads have not yet been allowed to run, they are created with an initial stack area that "looks like" the thread has been suspended by the K70 exception mechanism (i.e. it looks as though an interrupt has occurred). When a thread is launched for the first time, it will execute the program specified by the value in the `.stackedPC` location.

The **main**() function initialises the low-level hardware and calls **OS_Init**() and **OS_Start**() to start multithreading.

```
int main(void)
{
  PE_low_level_init();
  // Initialise OS - sets up SysTick
  OS_Init();
  // Call OS to start multitasking - never returns
  OS_Start(Threads);
}
```

The OS module provides three functions. **ThreadSwitchISR()** handles the context switching for the threads.

```
// Pointer to current thread
static TCBPtr RunPtr;

static uint32_t Count;

void __attribute__ ((interrupt)) ThreadSwitchISR(void)
{
  // Save current context
  __asm (\
    "pop {r0, r7}\n\t"\
    "stmdb r0!,{r4-r11}\n\t"\
    "str r0, %[input]\n\t"\
    ".align 4\n\t"\
    ::[input] "m" (*RunPtr)  \
    : "r0");

  // Increment the number of elapsed ticks
  Count++;

  // Get the next thread pointer - round robin scheduler
  RunPtr = RunPtr->nextTCB;

  // Load next context
  __asm (\
    "ldr r0, %[input]\n\t"\
    "ldmia r0!,{r4-r11}\n\t"\
    "msr msp, r0\n\t"\
    "bx lr\n\t"\
    ".align 4\n\t"\
    ::[input] "m" (*RunPtr)  \
    : "r0");
}
```

The assembly language with the comment “Save current context” does exactly that – it pushes the contents of the Coretex-M4 registers onto the stack to preserve them for later, and saves the current value of the stack pointer into the thread control block structure.

It then increments its own internal counter `Count`, and advances the `RunPtr` through the linked list to get the next thread control block – this is round-robin scheduling.

The assembly language with the comment “Load next context” does exactly that – it loads the stack pointer with the value that was previously saved in the new thread’s control block, and pops the contents of the Coretex-M4 registers back again.

7.16

OS_Init() initialises the SysTick timer for a 250 ms interval.

```
void OS_Init(void)
{
  // Disable interrupts
  __DI();

  // Disable SysTick
  SYST_CSR = 0;

  // Set reload value for a 250 ms period
  SYST_RVR = CPU_CORE_CLK_HZ / 4 - 1;

  // Clear current value as well as count flag
  SYST_CVR = 0;

  // Enable SysTick, SysTick exception and use core clock
  SYST_CSR = 7;
}
```

OS_Start() initialises the `RunPtr` and launches the first thread.

```
void OS_Start(TCBType threads[])
{
  uint32_t initialStackPtr;

  // Specify Thread 1 as the first thread to launch
  RunPtr = &threads[0];

  // Get savedSP address of Thread 1
  initialStackPtr = (uint32_t)RunPtr + sizeof(TCBType);

  // Set MSP to top of Thread 1 stack
  __asm (                  \
    "LDR R0, %[input]\n\t"\
    ::[input] "m" (initialStackPtr)  \
    : "r0");
  __asm ("msr msp, r0\n\t");
  __asm (".align 4\n\t");

  // Enable interrupts
  __EI();

  // Launch Thread 1
  (*RunPtr->stackedPC)();
}
```

Lastly, **ThreadSwitchISR** is placed into the correct location in the vector table in `vectors.c`:

```
(tIsrFunc)&Cpu_Interrupt,          /* 0x0E  PendableSrvReq */
(tIsrFunc)&ThreadSwitchISR,        /* 0x0F  SysTick        */
(tIsrFunc)&Cpu_Interrupt,          /* 0x10  DMA0_DMA16     */
```

7.5 The Semaphore

An operating system, at the very least, provides scheduling and synchronization tools for threads. One of the most important synchronization tools provided by an OS is the semaphore.

Semaphores defined

A *semaphore* is a non-negative integer, which may *only* be operated on by the primitive operations **`wait`** and **`signal`**. These **`wait`** and **`signal`** operations are *indivisible*. Indivisibility implies that only one thread can access each of these primitives at any one time. That is, only one primitive can modify the value of some semaphore, say `s`. The primitives can't be interrupted or logically cut into any smaller pieces.

The primitive operations are defined as follows:

`wait``(s)`	Decrease (indivisibly) the value of `s` by `1`
`signal``(s)`	Increase (indivisibly) the value of `s` by `1`

Note: A semaphore `s` may only be a non-negative integer, so that if the **`wait`**`(s)` *cannot be completed*, then a thread will be put into the waiting state. Conversely a **`signal`**`(s)` may cause a waiting thread to be made active.

Every **`signal`**`(s)` increments `s` but a **`wait`**`(s)` is only completed if `s > 0`.

7.5.1 Mutual Exclusion with Semaphores

Listing 7.1 shows how semaphores can be used to guarantee that a thread will have *uninterrupted* access to its critical code section. That is, there is mutual exclusion of other threads.

Mutual exclusion between threads

```
void p1(void)
{
  while (1)
  {
    ...
    OS_Wait(&Mutex);
    // critical code of p1
    ...
    OS_Signal(&Mutex);
    // remainder of p1
    ...
  }
}

void p2(void)
{
  while (1)
  {
    ...
    OS_Wait(&Mutex);
    // critical code of p2
    ...
    OS_Signal(&Mutex);
    // remainder of p2
    ...
  }
}

int Mutex; // a binary semaphore

void main(void)
{
  // Mutually excluded threads
  Mutex  = 1;
  OS_AddThread(&p1);
  OS_AddThread(&p2);

  OS_Launch(TIMESLICE); // doesn't return
}
```

Listing 7.1 – Mutual exclusion using Semaphores

There are two separate functions **`p1`** and **`p2`**, each with a critical section of code. Each of these critical sections are protected between **`OS_Wait`** and **`OS_Signal`** operations. A semaphore `Mutex` is initialised to `1` at the beginning of the main

program. This value guarantees only one thread can be in its critical section at one time.

Since the speed of each thread is indeterminate we have no way of knowing which one will try to execute an **OS_Wait**(&Mutex) first. Let us assume that **p1** tries first. Since the semaphore has a value of 1 at this point, the **OS_Wait** will complete (that is, **OS_Wait** will decrement Mutex to 0) and **p1** will enter it's critical section.

If while **p1** is in the critical section **p2** attempts a **OS_Wait**(&Mutex) then the **OS_Wait** will not complete and **p2** will be waiting (internally to the operating system, **p2** will be added to a queue of threads that are all waiting on the semaphore Mutex).

At some later time **p1** will complete its **OS_Signal**(&Mutex) operation, and **p2** may now complete its **OS_Wait**(&Mutex) and enter it's critical section. After each thread completes its **OS_Signal** operation, the value of the semaphore again becomes 1 allowing *either* thread to again gain mutually exclusive access to its critical section.

7.5.2 Synchronisation using Semaphores

Listing 7.2 shows how two threads **p1** and **p2** can synchronise their operations with each other.

Synchronization between threads

```
void p1(void)
{
  while (1)
  {
    // some amount of code
    ...
    OS_Wait(&Proceed);
    // remainder of p1
    ...
  }
}

void p2(void)
{
  while (1)
  {
    // some other amount of code
    ...
    OS_Signal(&Proceed);
    // remainder of p2
    ...
  }
}

int Proceed; // a binary semaphore

void main(void)
{

  // Synchronized threads
  Proceed = 0;
  OS_AddThread(&p1);
  OS_AddThread(&p2);

  OS_Launch(TIMESLICE); // doesn't return
}
```

Listing 7.2 – Synchronization using Semaphores

In this example **p1** will pause until **p2** executes a **OS_Signal**(&Proceed) before it continues. Of course if **p2** executes the **OS_Signal** before **p1** can execute the **OS_Wait** operation, then **p1** will not be held up.

This is known as *asymmetric* synchronisation – can you re-design this example so that symmetrical synchronisation between the two threads results?

7.5.3 The Producer / Consumer Problem using Semaphores

A classical problem in concurrent programming is the producer / consumer problem. Here two threads communicate through a buffer. The buffer has a finite amount of space and the producer, at its own speed, produces items and deposits them in this buffer of size `SpaceAvailable`.

The consumer, at its own speed, removes items from the buffer. Of course the consumer cannot extract items from an empty buffer nor can the producer deposit items into a full buffer.

There are to be three semaphores:

`SpaceAvailable`	This has an initial value of the size of the empty buffer.
`ItemsAvailable`	This has a value equal to the items in the buffer at initialisation (that is, `0`).
`BufferAccess`	This controls access to the buffer so that only one thread, producer or consumer, can gain access at one time.

Synchronization is achieved through the semaphores `SpaceAvailable` and `ItemsAvailable` whose initial values are the size of the buffer `40` and `0` respectively. Mutual exclusion of threads accessing the buffer simultaneously is effected by the semaphore `BufferAccess` with initial value `1`.

Producer / consumer problem using semaphores

```
void Producer(void)
{
  while (1)
  {
    // produce item
    ...
    OS_Wait(&SpaceAvailable);
    OS_Wait(&BufferAccess);
    // deposit item in buffer
    ...
    OS_Signal(&BufferAccess);
    OS_Signal(&ItemsAvailable);
  }
}

void Consumer(void)
{
  while (1)
  {
    OS_Wait(&ItemsAvailable);
    OS_Wait(&BufferAccess);
    // extract item from buffer
    ...
    OS_Signal(&BufferAccess);
    OS_Signal(&SpaceAvailable);
    // consume item
    ...
  }
}

int SpaceAvailable = 40; // size of the buffer
int ItemsAvailable = 0;
int BufferAccess = 1;

void main(void)
{
  // producer and consumer threads
  OS_AddThread(&Producer);
  OS_AddThread(&Consumer);

  OS_Launch(TIMESLICE); // doesn't return
}
```

Listing 7.3 – Producer / Consumer problem using Semaphores