**Interrupt Handling and Multi-threading**

**What is a difference between interrupt and polling**  
A) The first method is the simple one - **Polling**:   
  
The **main drawback** of this method when writing program is **waste of time of microcontroller, which needs to wait and check whether the new information has arrived.**The microcontroller continuously monitors the status of a given device. When the condition is met, it performs the necessary action for the device.

B) The second method is - Interrupt:

An interrupt is a hardware/ software triggered software action.

The second type of interrupt is **a periodic interrupt, which is triggered periodically by a hardware timer**. The MSP432/ TM4C microcontrollers have **SysTick and Timer interrupts**. The ISR will perform an action we wish to perform on a regular basis. For example, a data acquisition system needs to read the ADC at a regular rate.

The third type of interrupt is triggered by input/ output events. With an input device, the hardware will request an interrupt when input device has new data. The software interrupt service routine (ISR) will read from the input device and save (put) the data into a data structure located in shared memory. When the system wishes to process the data, it will check the status of the data structure, and if there is some data it will get it from the data structure located in shared memory. With an output device, the hardware will request an interrupt when the output device is idle. The ISR will get data from a data structure located in shared memory, and then write to the device. When the system wishes to output data, it will check the status of the data structure, and if there is room in the data structure, software will write (put) its data. Interrupts are an important synchronization mechanism in a real-time operating system because there will be multiple tasks to perform. To achieve real-time response interrupt-based synchronization serves as an important tool.

On the ARM Cortex-M processor, exceptions include resets, software interrupts and hardware interrupts. Each exception has an associated 32-bit vector that points to the memory location where the ISR that handles the exception is located. Vectors are stored in ROM at the beginning of memory. Check startup\_TM4C123.s file.

ROM location 0x0000.0000 has the initial stack pointer, and location 0x0000.0004 contains the initial program counter, which is called the reset vector. It holds the address of a function called the reset handler, which is the first thing executed following reset. There are hundreds of possible interrupt sources and their 32-bit vectors are listed in order starting with location 0x0000.0008. From a programming perspective, we can attach ISRs to interrupts by writing the ISRs as regular assembly subroutines or C functions with no input or output parameters and editing the startup\_TM4C123. s or startup\_msp432. s file to specify those functions for the appropriate interrupt.

In this class, we will write our ISRs using standard function names so that the startup files need not be edited. For example, we will simply name the ISRfor SysTick periodic interrupt asSysTick\_Handler. The ISR for this interrupt is a 32-bit pointer located at ROM address 0x0000.003C. Because the vectors are in ROM, this linkage is defined at compile time and not at run time. After the first 16 vectors, each processor will be different so check the data sheet.

**Interrupt controller and more**

**An *interrupt controller* multiplexes several input interrupts into a single output interrupt.** The controller also allows control over these individual input interrupts for disabling them, prioritizing them, and showing which are active. Because many embedded processors contain peripherals on-chip, the interrupts from these peripherals are also routed to the interrupt controller within the main processor.

Sometimes more interrupts are required in a system than there are interrupt pins in the processor. For these situations, peripherals can share an interrupt. The software must then determine which device caused the interrupt. **Interrupts can be either *maskable* or *nonmaskable*. Maskable interrupts can be disabled and enabled by software. Nonmaskable interrupts (*NMI*) are critical interrupts, such as a power failure or reset, that cannot be disabled by software.**

**Types of interrupts**

1. **Hardware interrupt**   
   A hardware interrupt is a signal which can tell the CPU that something happened in hardware device, and should be immediately responded. Hardware interrupts are triggered by peripheral devices outside the microcontroller. An interrupt causes the processor to save its state of execution and begin execution of an interrupt service routine.  
   Unlike the software interrupts, **hardware interrupts are asynchronous** and can occur in the middle of instruction execution, requiring additional care in programming. The act of initiating a hardware interrupt is referred to as an **interrupt request (IRQ)**.
2. **Software interrupt**   
   Software interrupt is an instruction which cause a context switch to an interrupt handler similar to a hardware interrupt. Usually it is an interrupt generated within a processor by executing a special instruction in the instruction set which causes an interrupt when it is executed.   
   Another type of software interrupt is triggered by an exceptional condition in the processor itself. This type of interrupt is often called a **trap** or **exception**.  
   Unlike the hardware interrupts where the number of interrupts is limited by the number of interrupt request (IRQ) lines to the processor, software interrupt can have hundreds of different interrupts.

**What is Interrupt Latency?**

Interrupt latency refers primarily to the software interrupt handling latencies. In other words**, the amount of time that elapses from the time that an external** **interrupt arrives** at the processor until **the time that** the **interrupt processing begins**. One of the most important aspects of kernel real-time performance is the ability to service an interrupt request (IRQ) within a specified amount of time.

**What all things should be there in standard interrupt service routine**

1. Return type should be void
2. **No function arguments** allowed in ISR
3. ISR should be as small as possible. There should not be any loops. It should be as simple as possible
4. The **ISR should disable the source of interrupt** only to be enabled later after the handler task gets over.

**Level triggered interrupt or any other interrupt. What is the sequence of enabling disabling interrupts**

void io\_init()

{

//UART intialization code

LPC\_UART2->IER = 1; //enable RDA interrupt

NVIC\_EnableIRQ(UART2\_IRQn); // Enable the source of the interrupt

}

**volatile int count=0**; //volatile global declared to be used in ISR

extern "C"

{

void UART2\_IRQHandler()

{

long int handler=0;

//check the status register for the interrupt

if (LPC\_UART2->IIR & 4) //4 = 0100 [010 : recv data available, 0 : atleast 1 interrupt pending]

{

recv\_char[count] = LPC\_UART2->RBR;

count++;

if(count == 15)

NVIC\_DisableIRQ(UART2\_IRQn); **//clearing the source of interrupt**

if(count>=15)

{

count=0;

{

xSemaphoreGiveFromISR(startdisp,&handler); // Jump t the task dependent on the ISR

portYIELD\_FROM\_ISR(handler);

}

}

}

}

}//end of extern “C”

void led\_update(void \*p)

{

xSemaphoreTake(startdisp,portMAX\_DELAY);//Task executed immediately after the ISR

while(1)

{

// some operation

NVIC\_EnableIRQ(UART2\_IRQn); //**Enable the source of interrupt only after the dependent task is done**

}

}

**Reentrant:**

In a multi-tasking system there is a potential for conflict if one task starts to access a resource but does not complete its access before being transitioned out of the running state. **If the task leaves the resource in an inconsistent state, then access to the same resource by any other task or interrupt could result in data corruption or other similar error.**

Following are some examples:

1. Task A writing to LCD and Task B also writing to LCD. Task A is pre-empted by task B just before completing the writing task. Task B writes the message on LCD. Task B enters in blocked state after completion of writing the message. Task A now resumes and finishes of writing the remaining message.   
   The output on the LCD will be corrupted.
2. **Read-Modify-Write Operations**

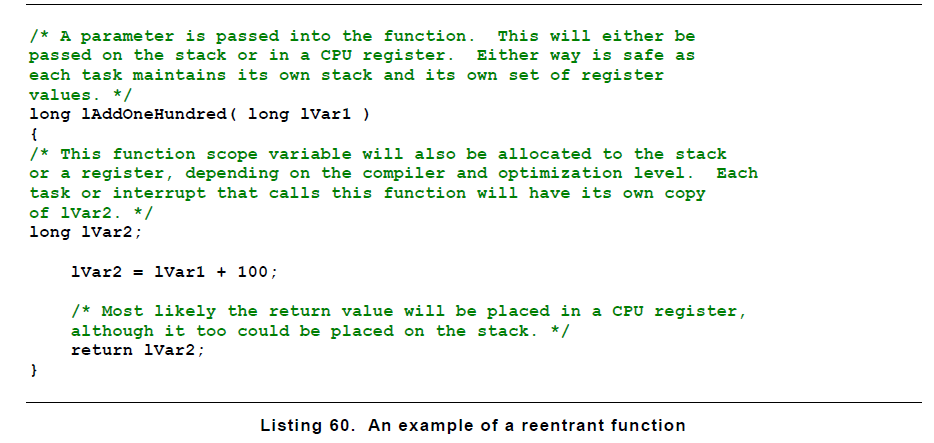
A C code to **read, modify and write is no-atomic operation in assembly**. It is **non-atomic** because it **takes more than one instruction to complete the operation** and hence can be interrupted.   
Task A reads the data from register and is preempted by task B. Task B writes new value to the register and enters Blocked state. Now task A resumes and works on previously held value of the register and updates the register. Thus, the value of the register gets corrupted as task A has modified the most recent value provided by task B.

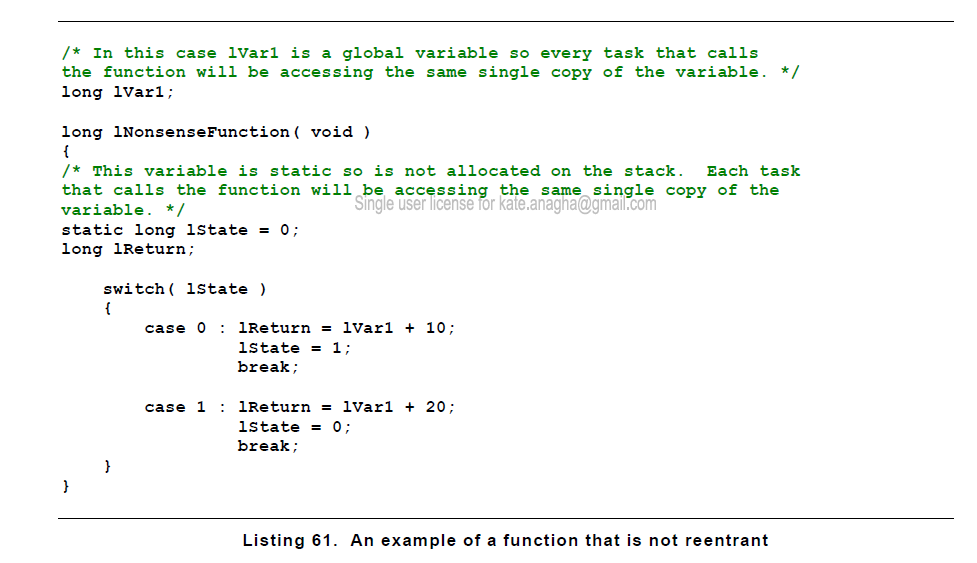
1. Non-atomic access to variables

**Updating multiple members of a structure are non-atomic operations**.

1. Function reentrancy

A **function is re-entrant** if it is safe to call the function from more than one task or from both, tasks and interrupts.   
**Each task maintains its own stack and its own set of core register values. If a function does not access any data other than data stored on the stack or held in a register, then the function is reentrant**.





**Basic critical sections are regions of the code that are surrounded by calls to the macros taskENTER\_CRITICAL() and taskEXIT\_CRITICAL() respectively**. -> interrupts disabled

Another option is to suspend the scheduler.

**Critical sections can also be controlled by suspending the scheduler.** Suspending the scheduler is sometimes also known as locking the scheduler.

Basic critical sections protect a region of code from access by other tasks and by interrupts. **A critical section implemented by suspending the scheduler protects a region of code only from access by other tasks because interrupts remain enabled.**

**A critical section that is too long to be implemented by simply disabling interrupts can instead be implemented by suspending the scheduler. However, resuming the scheduler can be relatively long operation, so consideration must be given to which is the best method to use in each case**.

**vTaskSuspendAll() -> prevents the context switching**. however, interrupts remain enabled. **If an interrupt requests context switch while the scheduler is suspended, then the request is held pending and is performed only when the scheduler is resumed.**

**xTaskResumeAll() -> scheduler is resumed**

When multiple threads are active, it is possible for two threads to be executing the same program. For example, the system may be running in the foreground and calls a function. Part way through execution of the function, an interrupt occurs. If the ISR also calls the same function, two threads are simultaneously executing the function. **If critical sections do exist, we can either eliminate them by removing the access to the global variable or implement mutual exclusion, which simply means only one thread at a time is allowed to execute in the critical section.** In general, if we can eliminate the global variables, then the subroutine becomes reentrant. Without global variables there are no “vulnerable” windows because each thread has its own registers and stack. Sometimes one must access global memory to implement the desired function. Remember that all I/ O ports are considered global. Furthermore, global variables are necessary to pass data between threads.

Program below shows two functions that can be used to implement mutual exclusion.

;\*\*\*\*\*\*\*\*\*\*\* StartCritical \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; make a copy of previous I bit, disable interrupts

; inputs: nonevoutputs: previous I bit

StartCritical

MRS R0, PRIMASK ; save old status

CPSID I ; mask all (except faults)

BX LR ;

\*\*\*\*\*\*\*\*\*\*\* EndCritical \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; using the copy of previous I bit, restore I bit to previous value

; inputs: previous I bit outputs: none

EndCritical

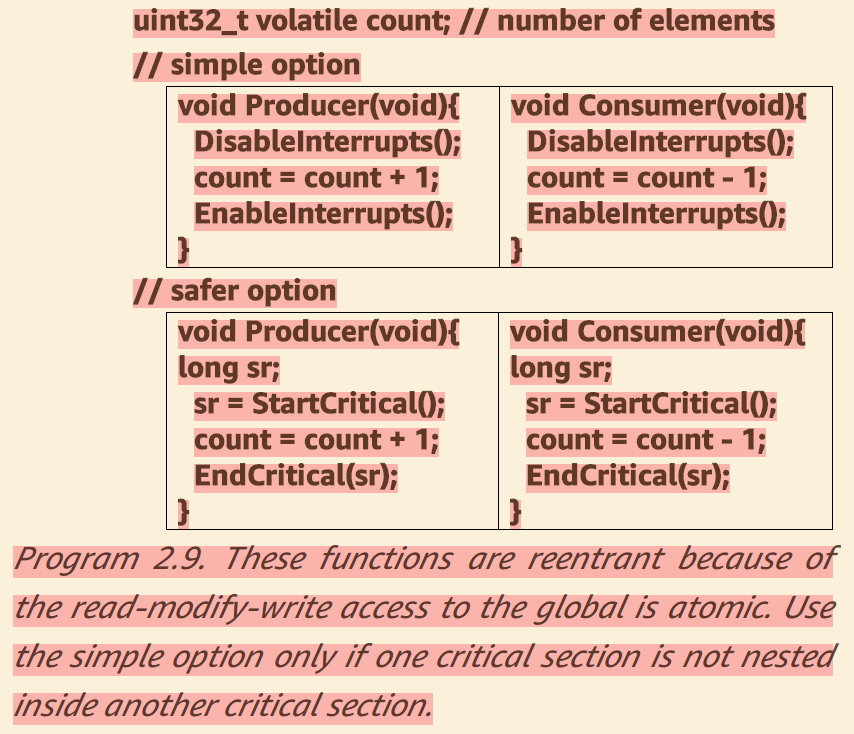
MSR PRIMASK, R0

BX LR

Program above Assembly functions needed to implement mutual exclusion.

A simple way to implement mutual exclusion is to disable interrupts while executing the critical section. It is important to disable interrupts for as short a time as possible, so as to minimize the effect on the dynamic performance of the other threads. While we are running with interrupts disabled, time-critical events like power failure and danger warnings cannot be processed. The assembly code of the above program is in the startup file in our projects that use interrupts.

Next program illustrates how to implement mutual exclusion and eliminate the critical section. When making code atomic with this simple method, make sure one critical section is not nested inside another critical section.



**Question:**

Although disabling interrupts does remove critical sections, it will add latency and jitter to real-time systems. Explain how latency and jitter are affected by the DisableInterrupts() and EnableInterrupts() functions.

**Answer:**

Since real-time events trigger interrupts, and the ISR software services the requests, **disabling interrupts will postpone the response causing latency or jitter.** **The maximum jitter will be the maximum time running with interrupts disabled.**

**Question**

Consider the situation of nested critical sections. For example, a function with a critical section calls another function that also has a critical section. What would happen if you simply added disable interrupts at the beginning and a re-enable interrupts at the end of each critical section?

**Answer**

Notice there are two disable interrupt and two enable interrupt functions, occurring in this order: 1) disable, 2) disable, 3) enable, 4) enable. Interrupts will be incorrectly enabled after step 3). Since the 1-4 represents a critical section and 2-3 is inside this section, a bug will probably be introduced. In this exampleStuff1B runs with interrupts enabled.

Critical1 Critical2

Disable // 1 Disable // 2

Stuff1A Stuff2A

Call Critical2 Enable // 3

Stuff1B return

Enable // 4

return

# [**What is the Re-entrant lock and concept in general?**](https://stackoverflow.com/questions/1312259/what-is-the-re-entrant-lock-and-concept-in-general)

**Re-entrant locking**

A reentrant lock is one where a process can claim the lock multiple times without blocking on itself. It's useful in situations where it's not easy to keep track of whether you've already grabbed a lock. If a lock is non re-entrant you could grab the lock, then block when you go to grab it again, effectively deadlocking your own process.

Reentrancy in general is a property of code where it has no central mutable state that could be corrupted if the code was called while it is executing. Such a call could be made by another thread, or it could be made recursively by an execution path originating from within the the code itself.

If the code relies on shared state that could be updated in the middle of its execution it is not re-entrant, at least not if that update could break it.

**A use case for re-entrant locking**

A (somewhat generic and contrived) example of an application for a re-entrant lock might be:

* You have some computation involving an algorithm that traverses a graph (perhaps with cycles in it). A traversal may visit the same node more than once due to the cycles or due to multiple paths to the same node.
* The data structure is subject to concurrent access and could be updated for some reason, perhaps by another thread. You need to be able to lock individual nodes to deal with potential data corruption due to race conditions. For some reason (perhaps performance) you don't want to globally lock the whole data structure.
* You computation can't retain complete information on what nodes you've visited, or you're using a data structure that doesn't allow 'have I been here before' questions to be answered quickly.  
    
  An example of this situation would be a simple implementation of Dijkstra's algorithm with a priority queue implemented as a binary heap or a breadth-first search using a simple linked list as a queue. In these cases, scanning the queue for existing insertions is O(N) and you may not want to do it on every iteration.

In this situation, keeping track of what locks you've already acquired is expensive. Assuming you want do the locking at the node level a re-entrant locking mechanism alleviates the need to tell whether you've visited a node before. You can just blindly lock the node, perhaps unlocking it after you pop it off the queue.

**Re-entrant mutexes**

A simple mutex is not re-entrant as only one thread can be in the critical section at a given time. If you grab the mutex and then try to grab it again a simple mutex doesn't have enough information to tell who was holding it previously. To do this recursively you need a mechanism where each thread had a token so you could tell who had grabbed the mutex. This makes the mutex mechanism somewhat more expensive so you may not want to do it in all situations.

IIRC the POSIX threads API does offer the option of re-entrant and non re-entrant mutexes.

# [**Does an interrupt handler have to be reentrant?**](https://stackoverflow.com/questions/18132580/does-an-interrupt-handler-have-to-be-reentrant)

The short answer is that Interrupt Service Routines are not inherently required to be reentrant. Reentrancy is only required in the case of [nested interrupts](http://www.electronics.dit.ie/staff/tscarff/6800/Interrupts/interrupts.htm). If the Operating System you use does not support [nested interrupts](http://www.electronics.dit.ie/staff/tscarff/6800/Interrupts/interrupts.htm), then you do not need to worry about reentrancy at all. If it does, you may have control over resetting the interrupt you are servicing so that you should never get a nested interrupt.

Essentially the answer to your question is that Linux masks an interrupt when it is asserted so that it won't preempt itself unless a specific flag is passed when registering the ISR.

Here's a relevant quote:

Interrupt handlers in Linux need not be reentrant. When a given interrupt handler is executing, the corresponding interrupt line is masked out on all processors, preventing another interrupt on the same line from being received. Normally all other interrupts are enabled, so other interrupts are serviced, but the current line is always disabled. Consequently, the same interrupt handler is never invoked concurrently to service a nested interrupt. This greatly simplifies writing your interrupt handler.

<http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0471g/Bgbeacfi.html>

**Nested Interrupts?**

<http://www.electronics.dit.ie/staff/tscarff/6800/Interrupts/interrupts.htm>

# [**Can an interrupt handler be preempted by the same interrupt handler?**](https://stackoverflow.com/questions/11403915/can-an-interrupt-handler-be-preempted-by-the-same-interrupt-handler)

# [**Why kernel code/thread executing in interrupt context cannot sleep?**](https://stackoverflow.com/questions/1053572/why-kernel-code-thread-executing-in-interrupt-context-cannot-sleep)

So what stops the scheduler from putting interrupt context to sleep and taking next schedulable process and passing it the control?

The problem is that the interrupt context is not a process, and therefore cannot be put to sleep.

When an interrupt occurs, the processor saves the registers onto the stack and jumps to the start of the interrupt service routine. This means that when the interrupt handler is running, it is running in the context of the process that was executing when the interrupt occurred. The interrupt is executing on that process's stack, and when the interrupt handler completes, that process will resume executing.

If you tried to sleep or block inside an interrupt handler, you would wind up not only stopping the interrupt handler, but also the process it interrupted. This could be dangerous, as the interrupt handler has no way of knowing what the interrupted process was doing, or even if it is safe for that process to be suspended.

A simple scenario where things could go wrong would be a deadlock between the interrupt handler and the process it interrupts.

1. *Process1* enters kernel mode.
2. *Process1* acquires *LockA*.
3. Interrupt occurs.
4. ISR starts executing using *Process1*'s stack.
5. ISR tries to acquire *LockA*.
6. ISR calls sleep to wait for *LockA* to be released.

At this point, you have a deadlock. *Process1* can't resume execution until the ISR is done with its stack. But the ISR is blocked waiting for *Process1* to release *LockA*.

**Shared Data and Race Conditions associated with ISR**

A common issue when designing software that uses interrupts is how to share data between the ISR and the main program.

Example

The following code example will give you a better understanding of race conditions. Imagine that the serial port ISR serialReceiveIsr is invoked when an incoming character arrives. As characters are received, gIndex is incremented to keep track of the number of characters stored in the memory buffer. The main function also uses gIndex, by decrementing it when it processes the received characters in the memory buffer.

Because an interrupt can occur at any time, the only way to make such a guarantee is to disable interrupts during the critical section. In order to protect the critical section in the previous example code, interrupts are disabled before the critical section executes and then enabled after, as shown here:

int gIndex = 0;

interrupt void serialReceiveIsr(void)

{

/\* Store receive character in memory buffer. \*/

gIndex++;

}

int main(void)

{

while (1)

{

interruptDisable( );

if (gIndex)

{

/\* Process receive character in memory buffer. \*/

gIndex--;

}

interruptEnable( );

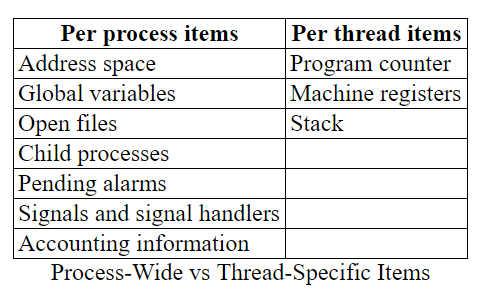
}

}

**Process and Threads**

A process contains a number of resources such as address space, open files, accounting information, etc. In addition to these resources, a process has a thread of control, e.g., program counter, register contents, stack.

The idea of threads is to permit multiple threads of control to execute within one process. This is often called **multithreading** and threads are sometimes called **lightweight processes**.



**User threads and Kernel threads**

### Implementing Threads in User Space

Write a (threads) library that acts as a mini-scheduler and implements thread\_create, thread\_exit, thread\_wait, thread\_yield, etc. This library acts as a run-time system for the threads in this process. The central data structure maintained and used by this library is a thread table, the analogue of the process table in the operating system itself.

There is a thread table and an instance of the threads library in each multithreaded process.

**Advantages of User-Mode Threads:**

* Requires **no** OS modification. Previously, this was the primary advantage since most operating systems did not support threads. Now, the major systems do support threads so this advantage is less significant.
* Very fast since no context switching.
* Can customize the scheduler for each application.

Disadvantages

* Blocking system calls cannot be executed directly since that would block the entire process. For example, consider the producer consumer example above implemented in the natural manner with user-mode threads. This implementation would not work well since, whenever an I/O was issued that caused the process to block, all the threads would be unable to run (but see just below).
* Similarly a page fault would block the entire process (i.e., all the threads).
* In addition, a thread with an infinite loop prevents all other threads in this process from ever running.

#### Possible Methods of Dealing With Blocking System Calls

* Perhaps the OS supplies a non-blocking version of the system call, e.g. a non-blocking read.
* Perhaps the OS supplies another system call that tells if the blocking system call will in fact block. For example, a unix select() can be used to tell if a read would block. It might not block if, for example,
  + The requested disk block is in the buffer cache (see the I/O chapter).
  + The request was for a keyboard or mouse or network event that has already happened.

**Multiprogramming**  
In a multiprogramming system **there are one or more programs loaded in main memory which are ready to execute. Only one program at a time can get the CPU for executing its instructions (i.e., there is at most one process running on the system) while all the others are waiting their turn.**

**The main idea of multiprogramming is to maximize the use of CPU time**. Indeed, suppose the currently running process is performing an I/O task (which, by definition, does not need the CPU to be accomplished). Then, the OS may interrupt that process and give the control to one of the other in-main-memory programs that are ready to execute (i.e. *process context switching*). In this way, **no CPU time is wasted by the system waiting for the I/O task to be completed, and a running process keeps executing until either it voluntarily releases the CPU or when it blocks for an I/O operation.**

Therefore, **the ultimate goal of multiprogramming is to keep the CPU busy** as long as there are processes ready to execute.  
  
**Note that in order for such a system to function properly, the OS must be able to load multiple programs into separate areas of the main memory and provide the required protection to avoid the chance of one process being modified by another one.**

**Multiprocessing**  
**Multiprocessing sometimes refers to executing multiple processes (programs) at the same time**.

In fact, **multiprocessing refers to the *hardware* (i.e., the CPU units) rather than the *software*** (i.e., running processes). If the underlying hardware provides more than one processor, then that is multiprocessing.

Anyway, a system can be both multiprogrammed by having multiple programs running at the same time and multiprocessing by having more than one physical processor.

The most common multiprocessor design is symmetric multiprocessing (or SMP), where all processors are considered peers and run independently of one another

Some systems use **asymmetric multiprocessing**, in which each processor is assigned a specific task. A master processor controls the system; the other processors

either look to the master for instruction or have predefined tasks. This scheme defines a master–slave relationship. The master processor schedules and allocates work to the slave processors. The most common systems use **symmetric multiprocessing (SMP)**, in which each processor performs all tasks within the operating system. SMP means that all processors are peers; no master–slave relationship exists between processors. Figure 1.6 illustrates a typical SMP architecture. Notice that each processor has its own set of registers, as well as a private—or local cache; however, all processors share physical memory.

**Summary**

This **asymmetric multiprocessing** is simple because only one processor accesses the system data structures, reducing the need for data sharing. A second approach uses **symmetric multiprocessing (SMP)**, where each

processor is self-scheduling. All processes may be in a common ready queue, or each processor may have its own private queue of ready processes. Regardless, scheduling proceeds by having the scheduler for each processor examine the ready queue and select a process to execute.

**Multitasking**

**Time sharing (or multitasking) is a logical extension of multiprogramming. In time-sharing systems, the CPU executes multiple jobs by switching among them, but the switches occur so frequently that the users can interact with each program while it is running**

**For Example, FreeRTOS is a real time kernel on top of which LPC17xx applications can be built to meet their hard real time requirements. It allows LPC17xx applications to be organized as a collection of independent threads of execution. As the LPC17xx has only one core, in reality only a single thread can be executing at one time. The kernel decides which thread should be executing by examining the priority assigned to each thread by the application designer.**

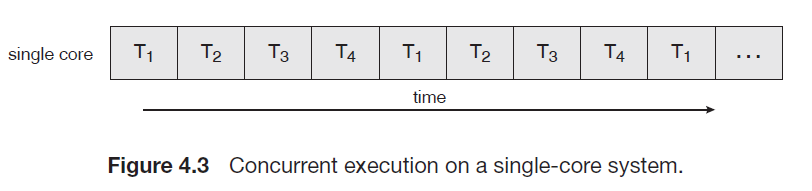
**At any time, the CPU is executing one task only while other tasks waiting their turn. Each task does not hijack the CPU until it finishes like in the multiprogramming but rather runs for a fair share amount of the CPU time called quantum**  
**In conclusion, both multiprogramming and multitasking operating systems are time sharing systems. However, while in multiprogramming one program as a whole keeps running until it blocks, in multitasking (modern OSs) time sharing system, each running process takes only a fair quantum of the CPU time.**

**Multithreading**

**Multithreading is an execution model that allows a single process to have multiple code segments (i.e., *threads*) run concurrently within the “context” of that process**.

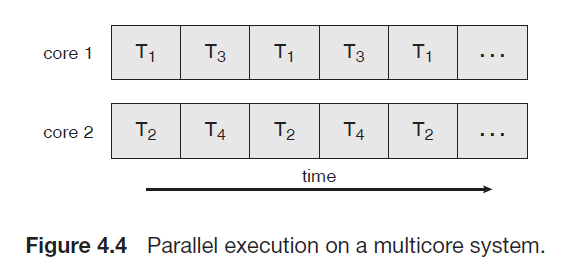
**Multiple threads of a single process can share the CPU in a single CPU system or (purely) run in parallel in a multiprocessing system.**

Consider an application with four threads. On a system with **a single computing core**, concurrency merely means that **the execution of the threads will be interleaved over time**. as the processing core is capable of executing only one thread at a time.



**Multicore Programming**

A recent trend in system design has been to place multiple computing cores on a single chip, where each core appears as a separate processor to the operating system (Section 1.3.2). Multithreaded programming provides a mechanism for more efficient use of multiple cores and improved concurrency. On a system with multiple cores, concurrency means that **the threads can run in parallel, as the system can assign a separate thread to each core**.



*A Side Note on Context Switching*

**Process context switching involves switching the virtual memory address space: this includes memory addresses, mappings, page tables, and kernel resources**.

On the other hand, **thread context switching is context switching from one thread to another in the same process** .

**Of course, switching from thread to thread across different processes is just like process context switching**.

**Real time OS**

A **real-time operating system** (**RTOS**) is an [operating system](https://en.wikipedia.org/wiki/Operating_system) (OS**) intended to serve**[**real-time**](https://en.wikipedia.org/wiki/Real-time_computing)**applications that process data as it comes in, typically without buffer delays.** Processing time requirements (including any OS delay) are measured in tenths of seconds or shorter increments of time. **They either are event driven or time sharing.**

**A key characteristic of an RTOS is the level of its consistency concerning the amount of time it takes to accept and complete an application's**[**task**](https://en.wikipedia.org/wiki/Task_(computing))**.**

The **chief design goal** of RTOS is **not high**[**throughput**](https://en.wikipedia.org/wiki/Throughput)**, but rather a guarantee of a**[**soft or hard**](https://en.wikipedia.org/wiki/Real-time_computing#Criteria_for_real-time_computing)**performance** category.

An RTOS has an advanced algorithm for [scheduling](https://en.wikipedia.org/wiki/Scheduling_(computing)).

Key factors in a real-time OS are **minimal**[**interrupt latency**](https://en.wikipedia.org/wiki/Interrupt_latency)**and minimal**[**thread switching latency**](https://en.wikipedia.org/wiki/Thread_switching_latency); a real-time OS is valued more for how quickly or how predictably it can respond than for the amount of work it can perform in a given period of time.

RTOS provides modularized approach to application development. As in the tasks in RTOS can be written as different and independent modules. This allows flexibility in development and testing. Modularity and lesser interdependence provide great code reusability.

The most common designs for RTOS are:

* **Event-driven –**[**switches tasks**](https://en.wikipedia.org/wiki/Context_switch)**only when an event of higher priority needs servicing**; called [**preemptive priority**](https://en.wikipedia.org/wiki/Preemption_(computing))**,** or priority scheduling.
* Time-sharing – **switches tasks on a regular clocked interrupt**, and on events; called [**round robin**](https://en.wikipedia.org/wiki/Round-robin_scheduling)**.**

In typical designs, a task has three states:

1. Running (executing on the CPU);
2. Ready/active (ready to be executed);
3. Blocked (waiting for an event, I/O for example).

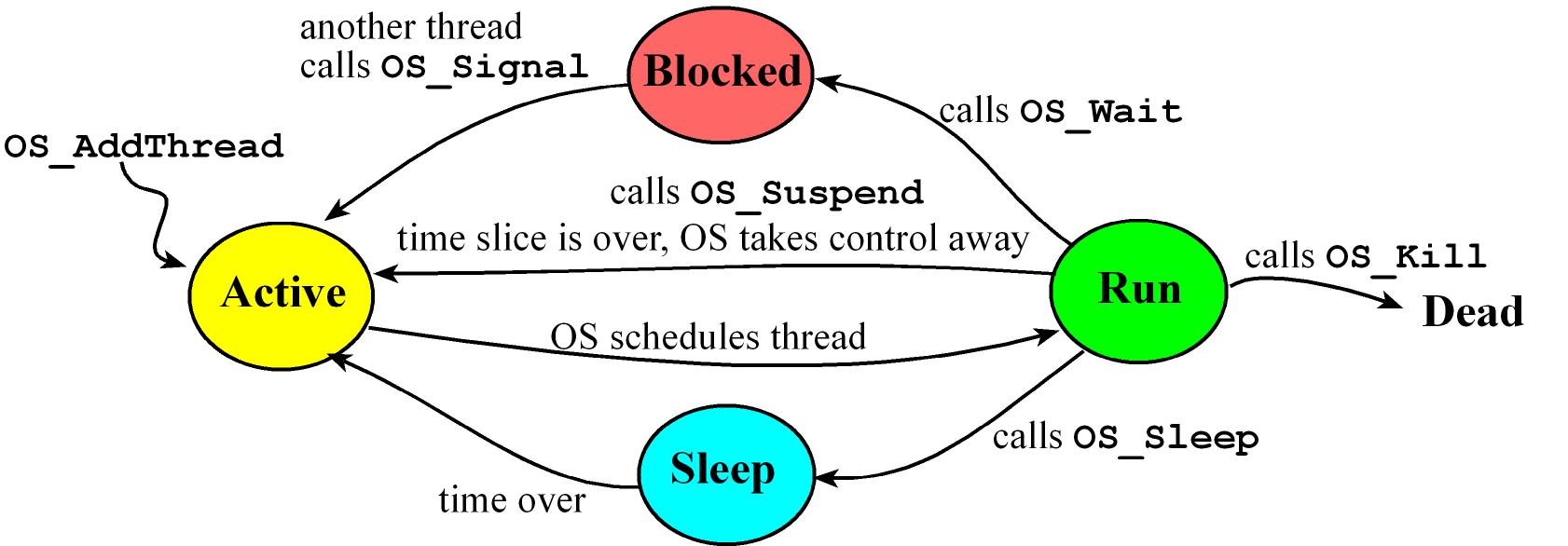
A main thread is in the **run state** if it currently executing. On a microcontroller with a single processor like the Cortex M, there can be at most one thread running at a time. For a multicore processor, there can be multiple threads in the run state.

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

A simple OS does not have sleeping or blocking; there will be one running thread and the other threads are active.

Sometimes a main thread needs to wait for a fixed amount of time. The OS will not run a main thread if it is in the sleep state.

A main thread is in the **blocked state** when it is waiting for some external event like input/ output (keyboard input available, printer ready, I/ O device available.) We will implement blocking in the next chapter.



*Figure 2.4. A main thread can be in one of four states.*

**OS Sleep**

**Sometimes a thread needs to wait for a fixed amount of time. We will implement an OS\_Sleep function that will make a thread dormant for a finite time.**

A thread in the sleep state will not be run. After the prescribed amount of time, the OS will make the thread active again. **Sleeping would be used for tasks which are not real-time**. In Program 4.9, the Periodic Stuff is run approximately once a second.

void Task( void)

{

InitializationStuff();

while( 1)

{

PeriodicStuff();

OS\_Sleep( ONE\_SECOND); // go to sleep for 1 second

}

}  
Program 4.9. This thread uses sleep to execute its task approximately once a second

To implement the sleep function, we could add a counter to each TCB and call it Sleep. If Sleep is zero, the thread is not sleeping and can be run, meaning it is either in the run or active state. If Sleep is nonzero, the thread is sleeping.

void Scheduler( void)

{

RunPt = RunPt-> next; // skip at least one

while(( RunPt-> Sleep) | |( RunPt-> blocked))

{

RunPt = RunPt-> next; // find one not sleeping and not blocked

}

}

Program 4.10. Round-robin scheduler that skips threads if they are sleeping or blocked.

Any thread with a nonzero Sleep counter will not be run. The user must be careful not to let all the threads go to sleep, because doing so would crash this implementation.

Next, we need to add a periodic task that decrements the Sleep counter for any nonzero counter. When a thread wishes to sleep, it will set its Sleep counter and invoke the cooperative scheduler. The period of this decrementing task will determine the resolution of the parameter time. Notice that this implementation is not an exact time delay. When the sleep parameter is decremented to 0, the thread is not immediately run. Rather, when the parameter reaches 0, the thread is signified ready to run. If there are n other threads in the TCB list and the thread switch time is Δt, then it may take an additional n\* Δt time for the thread to be launched after it awakens from sleeping.

The most commonly used RTOS scheduling algorithms are:

* [Cooperative scheduling](https://en.wikipedia.org/wiki/Cooperative_Scheduling)
* [Preemptive scheduling](https://en.wikipedia.org/wiki/Preemption_(computing))
  + [Round-robin scheduling](https://en.wikipedia.org/wiki/Round-robin_scheduling)
  + [Fixed priority pre-emptive scheduling](https://en.wikipedia.org/wiki/Fixed_priority_pre-emptive_scheduling)

**Co-operative Scheduling:**

When a pure co-operative scheduler is used, a **context switch will occur only when either the Running task enters the Blocked state or the Running task explicitly calls taskYield().**

**Tasks will never be preempted and tasks of equal priority will not automatically share processing time**. Co-operative scheduling in this manner is simple but can **result in** **less responsive system**.

**Preemptive Scheduling:**

**In a preemptive scheduler, main threads are suspended by a periodic interrupt, the scheduler chooses a new main thread to run, and the return from interrupt will launch this new thread. In this situation, the OS itself decides when a running thread will be suspended, returning it to the active state.**

**Round Robin Scheduling**

**A round robin scheduler simply runs the ready threads in circular fashion, giving each the same amount of time to execute.** A weighted round robin scheduler runs the ready threads in circular fashion, but gives threads unequal weighting. One way to implement weighting is to vary the time each thread is allowed to run according to its importance. Another way to implement weighting is to run important threads more often.

**Preemptive Priority Scheduler**

**A priority scheduler assigns each thread a priority number (e.g., 1 is the highest).** Two or more threads can have the same priority. A priority-2 thread is run only if no priority-1 threads are ready to run. Similarly, we run a priority-3 thread only if no priority-1 or priority-2 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.**

### Time slice

**The period of time for which a process is allowed to run in a preemptive multitasking system is generally called the *time slice* or *quantum***.

**The scheduler is run once every time slice to choose the next process to run. The length of each time slice can be critical to balancing system performance vs process responsivenes**s - if the time slice is too short then the scheduler will consume too much processing time, but if the time slice is too long, processes will take longer to respond to input.

An [interrupt](https://en.wikipedia.org/wiki/Interrupt) is scheduled to allow the [operating system](https://en.wikipedia.org/wiki/Operating_system) [kernel](https://en.wikipedia.org/wiki/Kernel_(computer_science)) to switch between processes when their time slices expire, effectively allowing the processor’s time to be shared between a number of tasks, giving the illusion that it is dealing with these tasks simultaneously, or concurrently. The operating system which controls such a design is called a multi-tasking system.

**Disadvantages of Priority Scheduler**

**One disadvantage of a priority scheduler on a busy system is that low priority threads may never be run. This situation is called starvation**. For example, if a high priority thread never sleeps or blocks, then the lower priority threads will never run. It is the responsibility of the user to assign priorities to tasks.

**One solution to starvation is called aging. Periodically the OS increases the temporary priority of threads that have not been run in a long time.**

For example, the Age field is incremented once every 1ms if the thread is not blocked or not sleeping. For every 10 ms the thread has not been run, its WorkingPriority is reduced. **Once a thread is run, its temporary priority is reset back to its permanent priority. When the thread is run, the Age field is cleared and the FixedPriority is copied into the WorkingPriority.**

struct tcb

{

int32\_t \*sp; // pointer to stack valid for threads not running

struct tcb \*next; // linked-list pointer

int32\_t \*BlockPt; // nonzero if blocked on this semaphore

uint32\_t Sleep; // nonzero if this thread is sleeping

uint8\_t WorkingPriority; // used by the scheduler

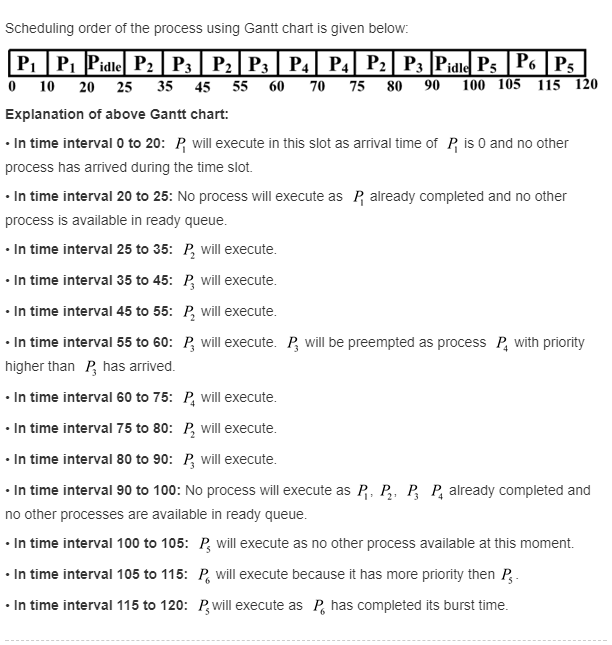
uint8\_t FixedPriority; // permanent priority

uint32\_t Age; // time since last execution

};

Program 5.4. TCB for the priority scheduler

Refer problem 17 in chapter 6 galvin



**Implementation of synchronization:**

Mutex:

**We use the mutex lock to protect critical regions and thus prevent race conditions. That is, a process must acquire the lock before entering a critical section; it releases the lock when it exits the critical section.**

**A mutex lock has a boolean variable** available whose value indicates if the lock is available or not. If the lock is available, a call to acquire() succeeds, and the lock is then considered unavailable. **A process that attempts to acquire an unavailable lock is blocked until the lock is released.**

Calls to either acquire() or release() must be performed atomically.

Semaphore:

Operating systems often distinguish between counting and binary semaphores. **The value of a counting semaphore can range over an unrestricted domain. The value of a binary semaphore can range only between 0 and 1. Thus, binary semaphores behave similarly to mutex locks**. **In fact, on systems that do not provide mutex locks, binary semaphores can be used instead for providing mutual exclusion**.

Counting semaphores can be used to control access to a given resource consisting of a finite number of instances. The semaphore is initialized to the number of resources available. Each process that wishes to use a resource performs a wait() operation on the semaphore (thereby decrementing the count). When a process releases a resource, it performs a signal() operation (incrementing the count). When the count for the semaphore goes to 0, all resources are being used. **After that, processes that wish to use a resource will block until the count becomes greater than 0.**

For example:

In process *P*1, we insert the statements

*S*1;

signal(synch);

In process *P*2, we insert the statements

wait(synch);

*S*2;

**When a process executes the wait() operation and finds that the semaphore value is not positive**, it must wait. However, rather than engaging in busy waiting, the process can block itself. The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state. Then control is transferred to the CPU scheduler, which selects another process to execute.

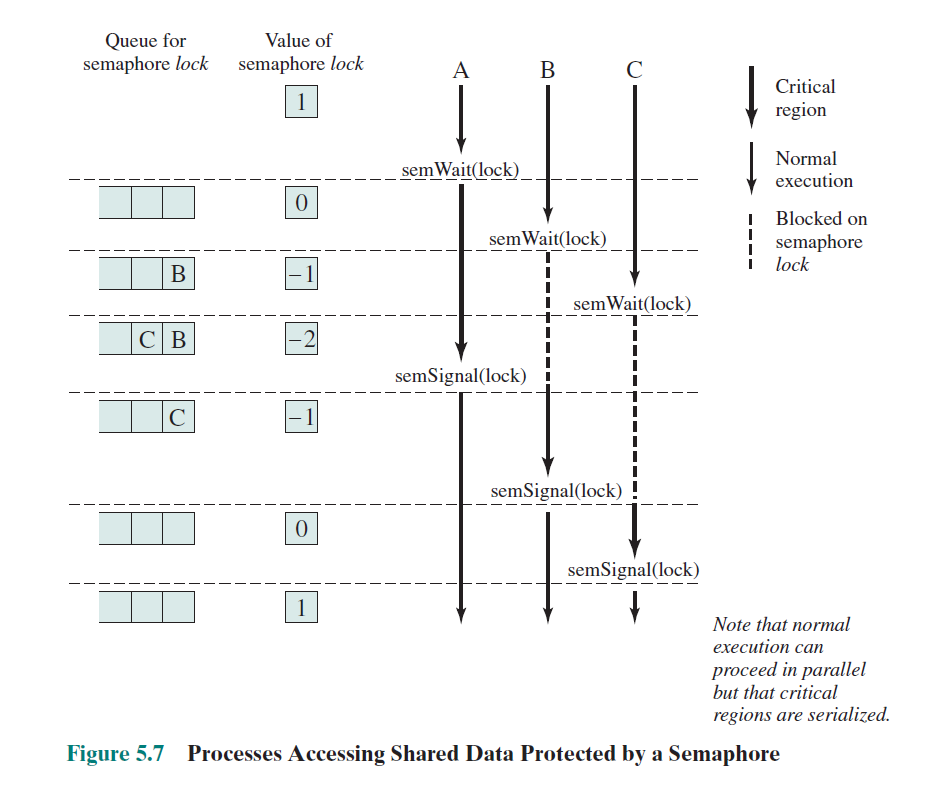
A process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a signal() operation. The process is restarted by a wakeup() operation, which changes the process from the waiting state to the ready state. The process is then placed in the ready queue.

**To achieve the desired effect, we can view the semaphore as a variable that has an integer value upon which only three operations are defined:**

**1. A semaphore may be initialized to a nonnegative integer value.**

**2. The semWait operation decrements the semaphore value. If the value becomes negative, then the process executing the semWait is blocked. Otherwise, the process continues execution.**

**3. The semSignal operation increments the semaphore value. If the resulting value is less than or equal to zero, then a process blocked by a semWait operation, if any, is unblocked.**



**What is a difference between mutex and binary semaphore**

They are **NOT** the same thing. They are used for different purposes!

One significant difference (since I've seen people make this mistake before): **a semaphore may be procured and vacated by any thread in any sequence** (so long as the count is never negative), but **a mutex may only be unlocked by the thread that locked it.** Attempting to unlock a mutex which was locked by another thread is undefined behavior

one VERY crucial distinction between a semaphore and a mutex; that is the usage. **Semaphores are signaling mechanisms at their heart**; the fact that they can be incremented and decremented by any thread is just a result of this. **Semaphores are used to signal to other flows-of-control something related to synchronization (like a fully/empty buffer).** **A mutex on the other hand, is always used to protect multiple access to a shared object.**

**Mutual Exclusion**

Mutual Exclusion semaphores are used to protect shared resources (data structure, file, etc..).

**Binary Semaphore** and **Counting Semaphore**

Binary semaphore being used to synchronize a task with an interrupt. The execution sequence is as follows

* 1. An interrupt occurs
  2. The interrupt service routine executes, ‘giving’ the semaphore to unblock the Handler task
  3. The Handler task executes as soon as the interrupt completes. The first thing the Handler task does is ‘take’ the semaphore.
  4. The Handler task processes the event before attempting to ‘take’ the semaphore again- entering the Blocked state if the semaphore is not immediately available.

This sequence is perfectly adequate if interrupts can occur only at a relatively low frequency. If another interrupt occurs before the Handler task has completed its processing of the first interrupt, then the binary semaphore will effectively latch the event, allowing the Handler task to process the new event immediately after it has completed processing the original event. The handler task will task not enter the Blocked state between processing the two events, as the latched semaphore would be available immediately, when xSemaphoreTake() called.

**Spinlock**

## Sharing spinlocks between interrupt and process-context

It is possible that a critical section needs to be protected by the same lock in both an interrupt and in non-interrupt (process) execution context in the kernel.

In this case spin\_lock\_irqsave and the spin\_unlock\_irqrestore variants have to be used to protect the critical section. This has the effect of disabling interrupts on the executing CPU. Imagine what would happen if you just used spin\_lock in the process context?

Picture the following:

1. Process context kernel code acquires lock A using spin\_lock.
2. While the lock is held, an interrupt comes in on the same CPU and executes.
3. Interrupt Service Routing (ISR) tries to acquire lock A, and spins continuously waiting for it.
4. For the duration of the ISR, the Process context is blocked and never gets a chance to run and free the lock.
5. Hard lock up condition on the CPU!

To prevent this, the process context code needs call spin\_lock\_irqsave which has the effect of disabling interrupts on that particular CPU along with the regular disabling of preemption we saw earlier before trying to grab the lock.

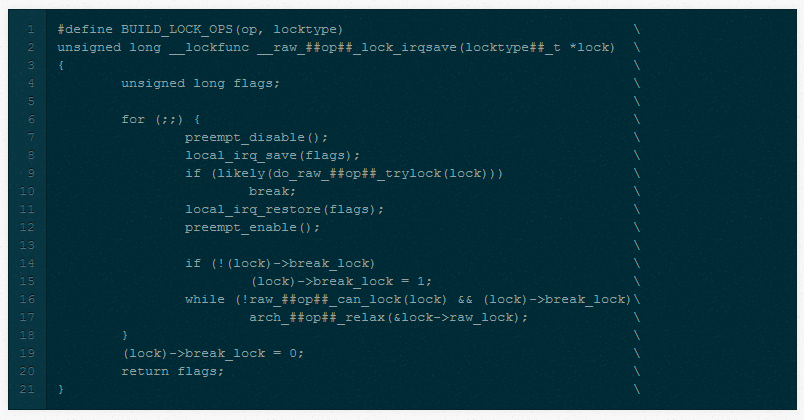
Note that the ISR can still just call spin\_lock instead of spin\_lock\_irqsave because interrupts are disabled anyway during ISR execution.

Also note that during the executing of the critical section protected by spin\_lock\_irqsave, the interrupts are only disabled on the executing CPU.

The same interrupt can come in on a different CPU and the ISR will be executed there, but that will not trigger the hard lock condition I talked about, because the process-context code is not blocked and can finish executing the locked critical section and release the lock while the ISR spins on the lock on a different CPU waiting for it. The process context does get a chance to finish and free the lock causing no hard lock up.

Following is what the spin\_lock\_irqsave code looks like for the SMP case, UP case is similar, look it up. BTW, the only difference here compared to the regular spin\_lock I described in the beginning are the local\_irq\_save and local\_irq\_restore that accompany the preempt\_disable and preempt\_enable in the lock code:

|  |  |
| --- | --- |
|  |  |



Unlike semaphores, spinlocks may be used in code that cannot sleep, such as interrupt handlers. When properly used, spinlocks offer higher performance than semaphores in general. They do, however, bring a different set of constraints on their use

Spinlocks are simple in concept. A spinlock is a mutual exclusion device that can have only two values: "locked" and "unlocked." It is usually implemented as a single bit in an integer value. Code wishing to take out a particular lock tests the relevant bit. If the lock is available, the "locked" bit is set and the code continues into the critical section. If, instead, the lock has been taken by somebody else, the code goes into a tight loop where it repeatedly checks the lock until it becomes available. This loop is the "spin" part of a spinlock.

Spinlocks are, by their nature, intended for use on multiprocessor systems, although a uniprocessor workstation running a preemptive kernel behaves like SMP, as far as concurrency is concerned. If a nonpreemptive uniprocessor system ever went into a spin on a lock, it would spin forever; no other thread would ever be able to obtain the CPU to release the lock. For this reason, spinlock operations on uniprocessor systems without preemption enabled are optimized to do nothing, with the exception of the ones that change the IRQ masking status. Because of preemption, even if you never expect your code to run on an SMP system, you still need to implement proper locking.

#### The Spinlock Functions

There are actually four functions that can lock a spinlock:

void spin\_lock(spinlock\_t \*lock);

void spin\_lock\_irqsave(spinlock\_t \*lock, unsigned long flags);

void spin\_lock\_irq(spinlock\_t \*lock);

void spin\_lock\_bh(spinlock\_t \*lock)

We have already seen how *spin\_lock* works. *spin\_lock\_irqsave* disables interrupts (on the local processor only) before taking the spinlock; the previous interrupt state is stored in flags. If you are absolutely sure nothing else might have already disabled interrupts on your processor (or, in other words, you are sure that you should enable interrupts when you release your spinlock), you can use *spin\_lock\_irq* instead and not have to keep track of the flags. Finally, *spin\_lock\_bh* disables software interrupts before taking the lock, but leaves hardware interrupts enabled.

If you have a spinlock that can be taken by code that runs in (hardware or software) interrupt context, you must use one of the forms of *spin\_lock* that disables interrupts. Doing otherwise can deadlock the system, sooner or later. If you do not access your lock in a hardware interrupt handler, but you do via software interrupts you can use *spin\_lock\_bh* to safely avoid deadlocks while still allowing hardware interrupts to be serviced.

There are also four ways to release a spinlock; the one you use must correspond to the function you used to take the lock:

void spin\_unlock(spinlock\_t \*lock);

void spin\_unlock\_irqrestore(spinlock\_t \*lock, unsigned long flags);

void spin\_unlock\_irq(spinlock\_t \*lock);

void spin\_unlock\_bh(spinlock\_t \*lock);

**Spinlock and Mutex? When to use what?**

**The Theory**

In theory, **when a thread tries to lock a mutex and it does not succeed, because the mutex is already locked, it will go to sleep, immediately allowing another thread to run. It will continue to sleep until being woken up, which will be the case once the mutex is being unlocked by whatever thread was holding the lock before**.

**When a thread tries to lock a spinlock and it does not succeed, it will continuously re-try locking it, until it finally succeeds; thus it will not allow another thread to take its place** (however, the operating system will forcefully switch to another thread, once the CPU runtime quantum of the current thread has been exceeded, of course).

**The Problem**

The problem with mutexes is that putting threads to sleep and waking them up again are both rather expensive operations, they'll need quite a lot of CPU instructions and thus also take some time.

**If now the mutex was only locked for a very short amount of time, the time spent in putting a thread to sleep and waking it up again might exceed the time the thread would have wasted by constantly polling on a spinlock.**

**On the other hand, polling on a spinlock will constantly waste CPU time and if the lock is held for a longer amount of time, this will waste a lot more CPU time and it would have been much better if the thread was sleeping instead.**

**In short spinlock is better if the locking is to be done for small amount of time otherwise Mutex**

**The Solution**

**Using spinlocks on a single-core/single-CPU system makes usually no sense, since as long as the spinlock polling is blocking the only available CPU core, no other thread can run and since no other thread can run, the lock won't be unlocked either. If the thread was put to sleep instead, another thread could have ran at once, possibly unlocking the lock and then allowing the first thread to continue processing, once it woke up again.**

**On a multi-core/multi-CPU systems**, with plenty of locks that are held for a very short amount of time only, the time wasted for constantly putting threads to sleep and waking them up again might decrease runtime performance noticeably. **When using spinlocks instead, threads get the chance to take advantage of their full runtime quantum (always only blocking for a very short time period, but then immediately continue their work), leading to much higher processing throughput**.

**Summary**

**If in doubt, use mutexes, they are usually the better choice and most modern systems will allow them to spinlock for a very short amount of time, if this seems beneficial. Using spinlocks can sometimes improve performance, but only under certain conditions** and the fact that you are in doubt rather tells me, that you are not working on any project currently where a spinlock might be beneficial. You might consider using your own "lock object", that can either use a spinlock or a mutex internally (e.g. this behavior could be configurable when creating such an object), initially use mutexes everywhere and if you think that using a spinlock somewhere might really help, give it a try and compare the results (e.g. using a profiler), but be sure to test both cases, a single-core and a multi-core system before you jump to conclusions (and possibly different operating systems, if your code will be cross-platform).

**Inter-process Communication**

Socket Programming

Semaphore

Signal

Message Queue

**Message queues allow one or more processes to write messages, which will be read by one or more reading processes**.

#### Shared-Memory Systems

* In general the memory to be shared in a shared-memory system is initially within the address space of a particular process, which needs to make system calls in order to make that memory publicly available to one or more other processes.
* Other processes which wish to use the shared memory must then make their own system calls to attach the shared memory area onto their address space.
* Generally a few messages must be passed back and forth between the cooperating processes first in order to set up and coordinate the shared memory access.

#### An Example: POSIX Shared Memory ( Eighth Edition Version )

1. The first step in using shared memory is for one of the processes involved to allocate some shared memory, using shmget:

int segment\_id = shmget( IPC\_PRIVATE, size, S\_IRUSR | S\_IWUSR );

* + The first parameter specifies the key ( identifier ) of the segment. IPC\_PRIVATE creates a new shared memory segment.
  + The second parameter indicates how big the shared memory segment is to be, in bytes.
  + The third parameter is a set of bitwise ORed flags. In this case the segment is being created for reading and writing.
  + The return value of shmget is an integer identifier

1. Any process which wishes to use the shared memory must attach the shared memory to their address space, using shmat:

char \* shared\_memory = ( char \* ) shmat( segment\_id, NULL, 0 );

* + The first parameter specifies the key ( identifier ) of the segment that the process wishes to attach to its address space
  + The second parameter indicates where the process wishes to have the segment attached. NULL indicates that the system should decide.
  + The third parameter is a flag for read-only operation. Zero indicates read-write; One indicates readonly.
  + The return value of shmat is a void \*, which the process can use ( type cast ) as appropriate. In this example it is being used as a character pointer.

1. Then processes may access the memory using the pointer returned by shmat, for example using sprintf:

sprintf( shared\_memory, "Writing to shared memory\n" );

1. When a process no longer needs a piece of shared memory, it can be detached using shmdt:

shmdt( shared\_memory );

1. And finally the process that originally allocated the shared memory can remove it from the system suing shmctl.

shmctl( segment\_id, IPC\_RMID );

1. Figure 3.16 from the eighth edition illustrates a complete program implementing shared memory on a POSIX system: Shared Memory Example in Multithreading

#### Pipes

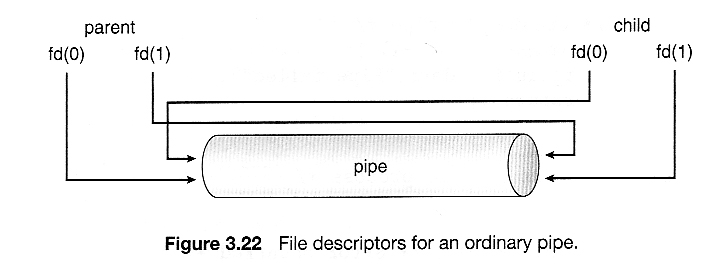
* ***Pipes*** are one of the earliest and simplest channels of communications between ( UNIX ) processes.

##### 3.6.3.1 Ordinary Pipes

* Ordinary pipes are uni-directional, with a reading end and a writing end. ( If bidirectional communications are needed, then a second pipe is required. )
* In UNIX ordinary pipes are created with the system call

"int pipe( int fd [ ] )".

* + The return value is 0 on success, -1 if an error occurs.
  + The int array must be allocated before the call, and the values are filled in by the pipe system call:
    - fd[ 0 ] is filled in with a file descriptor for the reading end of the pipe
    - fd[ 1 ] is filled in with a file descriptor for the writing end of the pipe
  + UNIX pipes are accessible as files, using standard read( ) and write( ) system calls.
  + Ordinary pipes are only accessible within the process that created them.
    - Typically a parent creates the pipe before forking off a child.
    - When the child inherits open files from its parent, including the pipe file(s), a channel of communication is established.
    - Each process ( parent and child ) should first close the ends of the pipe that they are not using. For example, if the parent is writing to the pipe and the child is reading, then the parent should close the reading end of its pipe after the fork and the child should close the writing end.
* Figure 3.22 shows an ordinary pipe in UNIX, and Figure 3.23 shows code in which it is used.

  
**Figure 3.24**

##### Named Pipes

* Named pipes support bidirectional communication, communication between non parent-child related processes, and persistence after the process which created them exits. Multiple processes can also share a named pipe, typically one reader and multiple writers.
* In UNIX, named pipes are termed fifos, and appear as ordinary files in the file system.
  + ( Recognizable by a "p" as the first character of a long listing, e.g. /dev/initctl )
  + Created with mkfifo( ) and manipulated with read( ), write( ), open( ), close( ), etc.
  + UNIX named pipes are bidirectional, but half-duplex, so two pipes are still typically used for bidirectional communications.
  + UNIX named pipes still require that all processes be running on the same machine. Otherwise sockets are used.
* Windows named pipes provide richer communications.
* Full-duplex is supported.
* Processes may reside on the same or different machines
* Created and manipulated using CreateNamedPipe( ), ConnectNamedPipe( ), ReadFile( ),and WriteFile( ).

**Deadlock**

In an [operating system](https://en.wikipedia.org/wiki/Operating_system), a deadlock occurs when a [process](https://en.wikipedia.org/wiki/Process_(computing)) or [thread](https://en.wikipedia.org/wiki/Thread_(computing)) enters a waiting [state](https://en.wikipedia.org/wiki/Process_state) because a requested [system resource](https://en.wikipedia.org/wiki/System_resource) is held by another waiting process, which in turn is waiting for another resource held by another waiting process. If a process is unable to change its state indefinitely because the resources requested by it are being used by another waiting process, then the system is said to be in a deadlock.

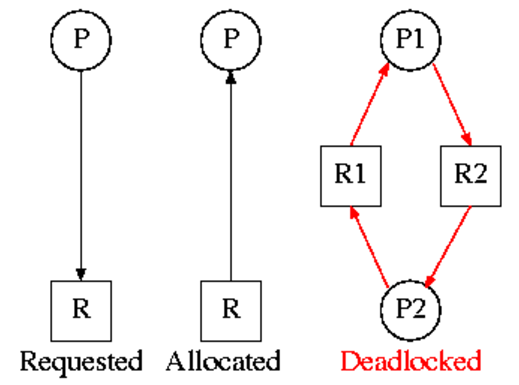
In a [communications system](https://en.wikipedia.org/wiki/Communications_system), deadlocks occur mainly due to lost or corrupt signals rather than resource contention

A deadlock situation can arise **if and only if all of the following conditions hold simultaneously** in a system.

1. [***Mutual exclusion***](https://en.wikipedia.org/wiki/Mutual_exclusion)*:* **The resources involved must be unshareable**; otherwise, the processes would not be prevented from using the resource when necessary. **Only one process can use the resource at any given instant of time.**
2. ***Hold and wait* or *resource holding****:* a process is currently **holding at least one resource and requesting additional resources** which are being held by other processes.
3. ***No***[***preemption***](https://en.wikipedia.org/wiki/Preemption_(computing))*:* a resource can be **released only voluntarily** by the process holding it.
4. ***Circular wait****:* **each process must be waiting for a resource which is being held by another process, which in turn is waiting for the first process to release the resource.** In general, there is a [set](https://en.wikipedia.org/wiki/Set_(mathematics)) of waiting processes, *P* = {*P*1, *P*2, …, *PN*}, such that *P*1 is waiting for a resource held by *P*2, *P*2 is waiting for a resource held by *P*3 and so on until *PN* is waiting for a resource held by *P*1

On the right are several examples of a **Resource Allocation Graph**, also called a **Reusable Resource Graph**.

* The processes are circles.
* The resources are squares.
* An arc (directed line) from a process P to a resource R signifies that process P has requested, but has **not** yet been allocated, resource R.
* An arc from a resource R to a process P indicates that process P has been allocated resource R and has not yet released it. We sometimes say that process P is *holding* resource R.



To illustrate this, consider a system consisting of two processes, *P*0 and *P*1, each accessing two semaphores, S and Q, set to the value 1:

*P*0 *P*1

wait(S); wait(Q);

wait(Q); wait(S);

. .

. .

. .

signal(S); signal(Q);

signal(Q); signal(S);

Suppose that *P*0 executes wait(S) and then *P*1 executes wait(Q).When *P*0 executes wait(Q), it must wait until *P*1 executes signal(Q). Similarly, when *P*1 executes wait(S), it must wait until *P*0 executes signal(S). Since these signal() **operations cannot be executed, *P*0 and *P*1 are deadlocked.**

**Homework:** 9. Are all such reusable resource graphs legal?

Consider two concurrent processes P1 and P2 whose programs are.

P1 P2

request R1 request R2

request R2 request R1

release R2 release R1

release R1 release R2

**Four strategies to deal with deadlocks**

1. Ignore the problem.
2. Detect deadlocks and recover from them.
3. Prevent deadlocks by violating one of the 4 necessary conditions.
4. Avoid deadlocks by carefully deciding when to allocate resources.

**Deadlock Detection and Correction**: Under the deadlock detection, deadlocks are allowed to occur. Then the state of the system is examined to detect that a deadlock has occurred and subsequently it is corrected.

**After a deadlock is detected, it can be corrected** by using one of the following method

1. ***Process termination****:* **one or more processes involved in the deadlock may be aborted**. One could choose to **abort all competing**[**processes**](https://en.wikipedia.org/wiki/Process_(computing)) involved in the deadlock. This ensures that deadlock is resolved with certainty and speed*.* But the expense is high as partial computations will be lost. Or, one could choose to abort one process at a time until the deadlock is resolved. This approach has high overhead because after each abort an algorithm must determine whether the system is still in deadlock.[[*citation needed*](https://en.wikipedia.org/wiki/Wikipedia:Citation_needed)] Several factors must be considered while choosing a candidate for termination, such as priority and age of the process.
2. ***Resource preemption****:* resources allocated to various processes may be successively preempted and allocated to other processes until the deadlock is broken.

**Deadlock Prevention**: Deadlock prevention works by preventing one of the four Coffman conditions from occurring.

1. Removing the *mutual exclusion* condition means that no process will have exclusive access to a resource. This proves impossible for resources that cannot be [spooled](https://en.wikipedia.org/wiki/Spooling). But even with spooled resources, deadlock could still occur. Algorithms that avoid mutual exclusion are called [non-blocking synchronization](https://en.wikipedia.org/wiki/Non-blocking_synchronization) algorithms.
2. The *hold and wait* or *resource holding* conditions may be removed by requiring processes to request all the resources they will need before starting up. This advance knowledge is frequently difficult to satisfy and, in any case, is an inefficient use of resources. Another way is that processes can request resources only when it has none. Thus, first they must release all their currently held resources before requesting all the resources they will need from scratch. This too is often impractical. It is so because resources may be allocated and remain unused for long periods. Also, a process requiring a popular resource may have to wait indefinitely, as such a resource may always be allocated to some process, resulting in [resource starvation](https://en.wikipedia.org/wiki/Resource_starvation).[[12]](https://en.wikipedia.org/wiki/Deadlock#cite_note-12) (These algorithms, such as [serializing tokens](https://en.wikipedia.org/wiki/Serializing_tokens), are known as the *all-or-none algorithms*.)
3. The *no*[*preemption*](https://en.wikipedia.org/wiki/Preemption_(computing)) condition may also be difficult or impossible to avoid as a process has to be able to have a resource for a certain amount of time, or the processing outcome may be inconsistent or [thrashing](https://en.wikipedia.org/wiki/Thrashing_(computer_science)) may occur. However, inability to enforce preemption may interfere with a *priority* algorithm. Preemption of a "locked out" resource generally implies a [rollback](https://en.wikipedia.org/wiki/Rollback_(data_management)), and is to be avoided, since it is very costly in overhead. Algorithms that allow preemption include [lock-free and wait-free algorithms](https://en.wikipedia.org/wiki/Lock-free_and_wait-free_algorithms) and [optimistic concurrency control](https://en.wikipedia.org/wiki/Optimistic_concurrency_control). If a process holding some resources and requests for some another resource(s) that cannot be immediately allocated to it, the condition may be removed by releasing all the currently being held resources of that process.
4. The final condition is the *circular wait* condition. Approaches that avoid circular waits include disabling interrupts during critical sections and using a hierarchy to determine a [partial ordering](https://en.wikipedia.org/wiki/Partial_order) of resources. If no obvious hierarchy exists, even the memory address of resources has been used to determine ordering and resources are requested in the increasing order of the enumeration.[[3]](https://en.wikipedia.org/wiki/Deadlock#cite_note-os_galvin-3)[Dijkstra's solution](https://en.wikipedia.org/wiki/Dining_philosophers_problem#Resource_hierarchy_solution) can also be used.

**Priority Inversion**:

In some cases, priority inversion can occur without causing immediate harm—the delayed execution of the high priority task goes unnoticed, and eventually the low priority task releases the shared resource. However, there are also many situations in which priority inversion can cause serious problems. If the high priority task is left [starved](https://en.wikipedia.org/wiki/Resource_starvation) of the resources, it might lead to a system malfunction or the triggering of pre-defined corrective measures, such as a [watchdog timer](https://en.wikipedia.org/wiki/Watchdog_timer) resetting the entire system.

Consider higher priority task, task 2 having to wait for the lower priority task, task 1 to give up control of the mutex. A high priority task being delayed by low priority task is priority inversion. This undesirable behavior would be exaggerated further I f the medium priority task started to execute while the high priority task was waiting for the mutex- the result would be a high priority task waiting for a low priority task without the low priority task even being able to execute.

**Priority Inheritance**:

In [real-time computing](https://en.wikipedia.org/wiki/Real-time_computing), **priority inheritance** is a method for eliminating unbounded [priority inversion](https://en.wikipedia.org/wiki/Priority_inversion).

FreeRTOS mutexes and binary semaphores are very similar – the difference being that mutexes include priority inheritance mechanism whereas binary semaphores do not.

Priority inheritance is a scheme that minimizes the negative effects of priority inversion. It does not fix priority inversion; it merely lessens its impact by ensuring that the inversion is always time bounded. However, priority inheritance complicates system timing analysis; it is not good practice to rely on it for correct system operation.

Suppose H is blocked by L for some shared resource. The priority inheritance protocol requires that L executes its critical section at H's (high) priority. As a result, M will be unable to [preempt](https://en.wikipedia.org/wiki/Preemption_(computing)) L and will be blocked. That is, the higher-priority job M must wait for the critical section of the lower priority job L to be executed, because L has inherited H's priority. When L exits its critical section, it regains its original (low) priority and awakens H (which was blocked by L). H, having high priority, preempts L and runs to completion. This enables M and L to resume in succession and run to completion.