RTOS

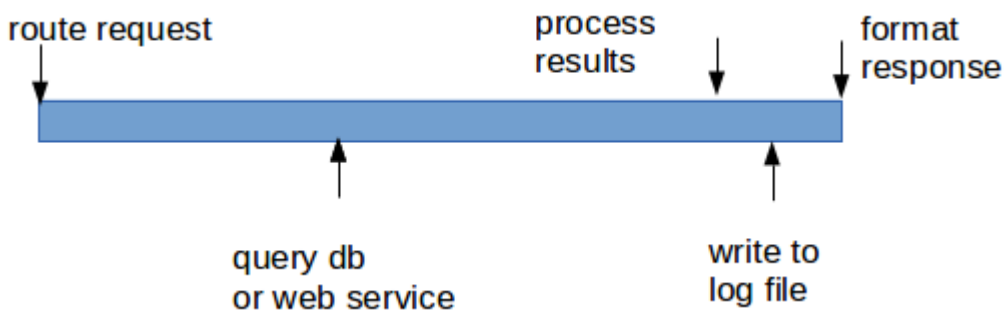
The followings are the characteristics of RTOS:

1. **Context switching latency**:
   Context switch latency is the time from one context switching to another and it should be short. In other words, the time taken while saving the context of current task and then switching over to another task should be short. In general, switching context involved saving the CPU's registers and loading a new state, flushing the caches, and changing the virtual memory mapping. Context switch latency is highly architecture dependent and different hardware may get different results.
2. **Interrupt latency**:
   Interrupt latency is the time from interrupt generation until the interrupt service routine starts executing.
   Factors that affect interrupt latency include the processor architecture, the processor clock speed, the particular OS employed, and the type of interrupt controller used.
   Minimum interrupt latency depends mainly on the configuration of the interrupt controller, which combines interrupts onto processor lines, and assigns priority levels (visit Priority Inversion) to the interrupts.
   Maximum interrupt latency depends mainly on the OS.
   For more on Interrupt and Interrupt Latency, please visit my another page Interrupt & Interrupt Latency
3. **Dispatch latency**:
   The time between when a thread is scheduled and when it begins to execute. Theoretically, in a preemptive OS the dispatch latency for a high-priority thread should be very low. However, in practice preemptive OSs are non-preemptive at times; for example, while running an interrupt handler. The duration of the longest possible non-preemptive interval is said to be the worst-case dispatch latency of an OS.
4. **Reliable and time bound inter process mechanisms** should be in place for processes to communicate with each other in a timely manner.
5. **Multitasking and task preemption**:
   An RTOS should have support for multitasking and task preemption. Preemption means to switch from a currently

executing task to a high priority task ready and waiting to be executed.

6. **Kernel preemption**:
   Most modern systems have preemptive kernels, designed to permit tasks to be preempted even when in kernel mode.
   The bright side of the preemptive kernel is that sys-calls do not block the entire system.
   However, it introduces more complexity to the kernel code, having to handle more end-cases, perform more fine grained locking or use lock-less structures and algorithms.
7. **Note: Preemptive**:
   Preemptive means that the rules governing which processes receive use of the CPU and for how long are determined by the **kernel process scheduler**.

*Most of the requests are spending time just waiting due to I/Os

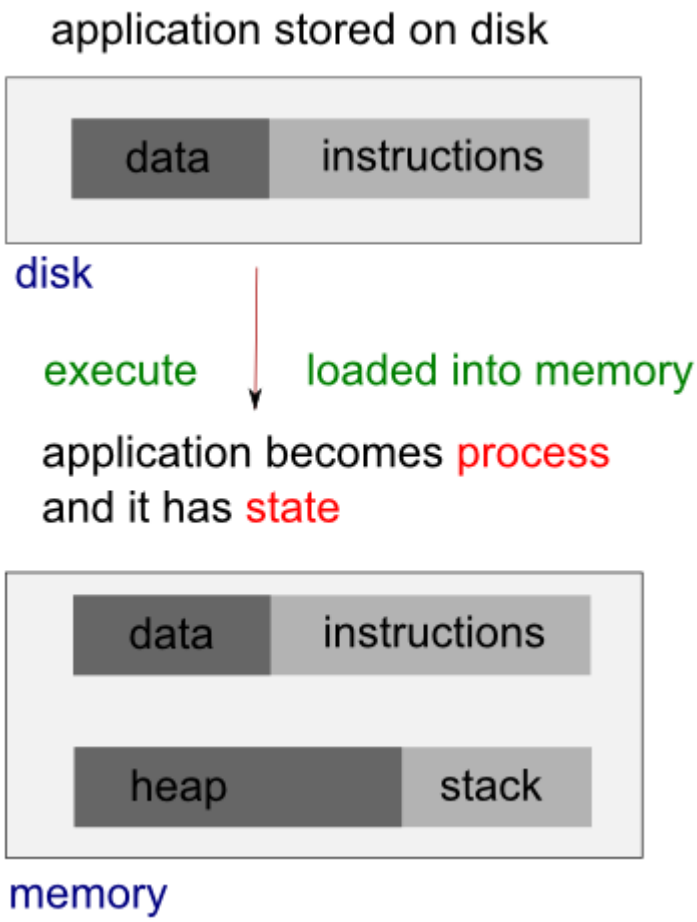


Process

A **process** can be thought of as an instance of a running program. Each process is an independent entity to which system resources such as CPU time, memory, etc. are allocated and each process is executed in a separate address space. If we want to access another process' resources, **inter-process communications** have to be used such as **pipes**, **files**, **sockets** etc.
Process provides each program with **two key abstractions**:

1. Logical control flow
   Each process seems to have exclusive use of the CPU

2. Private virtual address space
   Each process seems to have exclusive use of main memory



Thread

A key difference between processes and threads is that multiple threads **share** parts of their state. Typically, multiple threads can read from and write to the same memory (no process can directly access the memory of another process). However, each thread still has its own stack of activation records and its own copy of CPU registers, including the stack pointer and the program counter, which together describe the **state** of the thread's execution.

Process Vs Thread

1. Processes are independent while thread is within a process.
2. Processes have separate address spaces while threads share their address spaces.
3. Processes communicate each other through inter-process communication.

4. Processes carry considerable state (e.g., ready, running, waiting, or stopped) information, whereas multiple threads within a process share state as well as memory and other resources.
5. Context switching between threads in the same process is typically faster than context switching between processes.
6. **Multithreading** has some advantages over **multiple processes**. Threads require less overhead to manage than processes, and intraprocess thread communication is less expensive than interprocess communication.
7. **Multiple process** concurrent programs do have one advantage: Each process can execute on a different machine (**distribute program**). Examples of distributed programs are file servers (NFS), file transfer clients and servers (FTP), remote log-in clients and servers (Telnet), groupware programs, and Web browsers and servers.

Multiprocessing vs. Multithreading

1. **Multiprocessing**
    1. A process is the unit of resource allocation & protection.
    2. A process manages certain resources, e.g., virtual memory, I/O handlers, and signal handlers.
    3. **Pros**: Process is protected from other processes via an MMU.
    4. **Cons**: IPC between processes can be complicated and inefficient.
2. **Multithreading**
    1. A thread is the unit of computation that runs in the context of a process.
    2. A thread manages certain resources, e.g., stack, registers, signal masks, priorities, and thread-specific data
    3. **Pros**: IPC between threads is more efficient that IPC between processes.
    4. **Cons**: Threads can interfere with each other.

Scheduling Methods in RTOS

Preemptive

1. Each Task has a priority relative to all other tasks
2. The most critical Task is assigned the highest priority
3. The highest priority Task that is ready to run gets control of the processor
4. A Task runs until it yields, terminates, or blocks

5. Each Task has its own memory stack
6. Before a Task can run it must load its context from its memory stack .(this can take many cycles)
7. If a Task is preempted it must save its current state/context; this context is restored when the Task is given control of the processor

In this preemptive scheduling a thread (or a process) must be in one of the following four states:

1. Running - the task is in control of the CPU
2. Ready - the task is not blocked and is ready to receive control of the CPU when the scheduling policy indicates it is the highest priority task in the system that is not blocked.
3. Inactive - the task is blocked and requires initialization in order to become ready.
4. Blocked - the task is waiting for something to happen or for a resource to become available.

Round Robin

In this method, time slices are assigned to each process in equal portions and in circular order, handling all processes without priority (also known as cyclic executive). Round-robin scheduling is simple, easy to implement, and starvation-free.

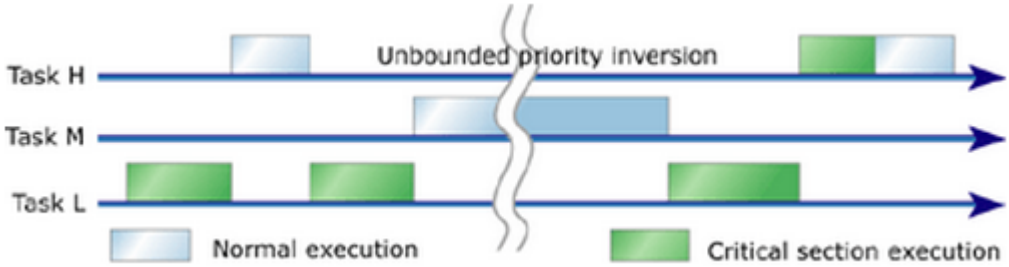
Priority Inversion

The existence of this problem has been known for a while, however, there is no fool-proof method to predict the situation.

To ensure rapid response times, an embedded **RTOS** can use **preemption (higher-priority task can interrupt a low-priority task that's running)**. When the high-priority task finishes running, the low-priority task resumes executing from the point at which it was interrupted.

Unfortunately, the need to share resources between tasks operating in a preemptive multitasking environment can create conflicts. Along with a deadlock, the **priority inversion** is one of the two of the most common

problems which can result in application failure.



To avoid priority inversion, the priority is set (hoisted) to a predefined priority higher than or equal to the highest priority of all the threads that might lock the particular mutex. This is known as **Priority Ceiling**.

Race Condition

This section is mostly from the book "C++ Concurrency in Action Practical Multithreading" by Anthony Williams.

In concurrency, a **race condition** is anything where the outcome depends on the relative ordering of execution of operations on two or more threads where the threads race to perform their respective operations. Most of the time, this is quite benign because all possible outcomes are acceptable, even though they may change with different relative orderings. For example, if two threads are adding items to a queue for processing, it generally doesn't matter which item gets added first, provided that the invariants of the system are maintained.

It's when the race condition leads to broken invariants that there's a problem, such as with the doubly linked list example mentioned in Invariants section of this chapter.

When talking about concurrency, the term race condition is usually used to mean a problematic race condition; benign race condition is not so interesting and should not be a cause of bugs. The C++ Standard also defines the term data race to mean the specific type of race condition that arises because of **concurrent modification to a single object such that data races cause the undefined behavior**.

This happens when a critical section is not executed atomically. An execution of threads depends on shared state. For example, two threads share variable **i**and trying to increment it by 1. It is highly dependent on when they get it and when they save it.

Typically, problematic race conditions occur where completing an operation requires modification of two or more distinct pieces of data, such as the two link pointers in the example of Invariants section. Because the operation must access two separate pieces of data, these must be modified in separate instructions, and another thread could potentially access the data structure when only one of them has been completed.

Race conditions can often be hard to find and hard to duplicate because the window of opportunity is small. If the modifications are done as consecutive CPU instructions, the chance of the problem exhibiting on any one run-through is very small, even if the data structure is being accessed by another thread concurrently. As the load on the system increases, and the number of times the operation is performed increases, the chance of the problematic execution sequence occurring also increases. It's almost inevitable that such problems will show up at the most inconvenient time.

Because race conditions are generally timing sensitive, they can often disappear entirely when the application is run under the debugger, because the debugger affects the timing of the program, even if only slightly.

If we are writing multithreaded programs, race conditions can easily be the bane of our life. A great deal of the complexity in writing software that uses concurrency comes from avoiding problematic race conditions.

There are several ways to deal with problematic race conditions. The simplest option is to wrap your data structure with a **protection mechanism**, to ensure that only the thread actually performing a modification can see the intermediate states where the invariants are broken. From the point of view of other threads accessing that data structure, such modifications either haven't started or have completed.

Another option is to modify the design of your data structure and its invariants so that modifications are done as a series of indivisible changes, each of which preserves the invariants. This is generally referred to as **lock-free programming** and is difficult to get right. If we're working at this level, the nuances of the **memory model** and identifying which threads can potentially see which set of values can get complicated.

Another way of dealing with race conditions is to handle the updates to the data structure as a transaction, just as updates to a database are done within a transaction. The required series of data modifications and reads is stored in a transaction log and then committed in a single step. If the commit can't proceed because the data structure has been modified by another

thread, the transaction is restarted. This is termed **software transactional memory (STM)**, and currently, it's an active research area.

The most basic mechanism for protecting shared data is probably the **mutex**.

Suppose, two person **A** and **B** share the same account and try to deposit roughly at the same time. In the following example, they deposited 1$ million times. So, we expect the balance would be $2,000,000. However, it turns out that's not the case as we see from the output:

```
// global1.c
#include <stdio.h>
#include <pthread.h>

static volatile int balance = 0;

void *deposit(void *param)
{
    char *who = param;

    int i;
    printf("%s: begin\n", who);
    for (i = 0; i < 1000000; i++) {
        balance = balance + 1;
    }
    printf("%s: done\n", who);
    return NULL;
}

int main()
{
    pthread_t p1, p2;
    printf("main() starts depositing, balance = %d\n", ba
    pthread_create(&p1;, NULL, deposit, "A");
```

```
    pthread_create(&p2;, NULL, deposit, "B");

    // join waits for the threads to finish
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf("main() A and B finished, balance = %d\n", bal
    return 0;
}
```

Output:

```
$ ./global
main() starts depositing, balance = 0
B: begin
A: begin
B: done
A: done
main() A and B finished, balance = 1270850
----
$ ./global
main() starts depositing, balance = 0
B: begin
A: begin
B: done
A: done
main() A and B finished, balance = 1423705
```

Note that both threads accessing the same function and share global variable **balance**.

Put simply, what could happened is this: the code to increment balance has been run twice by A and B, in both cases, however, the balance might started at 0 without noticing the other person was in the process of putting money into the same account.

We need to make the balance updating code (**critical section**) as atomic:

```
// global2.c
#include <stdio.h>
#include <pthread.h>

static volatile int balance = 0;
pthread_mutex_t myMutex;

void *deposit(void *param)
{
    char *who = param;

    int i;
    printf("%s: begin\n", who);
    for (i = 0; i < 1000000; i++) {

        // critical section
        pthread_mutex_lock(&myMutex;);
        balance = balance + 1;
        pthread_mutex_unlock(&myMutex;);

    }
    printf("%s: done\n", who);
    return NULL;
}

int main()
{
    pthread_t p1, p2;
    printf("main() starts depositing, balance = %d\n", ba
    pthread_mutex_init(&myMutex;,0);
    pthread_create(&p1;, NULL, deposit, "A");
    pthread_create(&p2;, NULL, deposit, "B");

    // join waits for the threads to finish
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    pthread_mutex_destroy(&myMutex;);
    printf("main() A and B finished, balance = %d\n", bal
    return 0;
}
```

Makefile:

```
global2: global2.o
        gcc -D_REENTRANT -o global2 global2.o -Wall -lpth
global2.o: global2.c
        gcc -c global2.c
clean:
        rm -f *.o global2
```

Now, our output looks desirable:

```
$ make
gcc -c global2.c
gcc -D_REENTRANT -o global2 global2.o -Wall -lpthread
$ ./global2
main() starts depositing, balance = 0
B: begin
A: begin
B: done
A: done
main() A and B finished, balance = 2000000
```

We can detect the potential race condition using **valgrind --tool=helgrind program_name**.

Here are some of the main components of an RTOS. The SimpleLink SDK offers all of these features for both TI-RTOS and FreeRTOS.

- **Scheduler**: Preemptive scheduler that guarantees the highest priority thread it running.
- **Communication Mechanism**: Semaphores, Message Queues, Queues, etc.
- **Critical Region Mechanisms**: Mutexes, Gates, Locks, etc.
- **Timing Services**: Clocks, Timers, etc.
- **Power Management**: For low power devices, power management is generally part of the RTOS since it knows the state of the device.
- **Memory Management**: Variable-size heaps, fixed-size heaps, etc.
- **Peripheral Drivers**: UART, SPI, I2C, etc.
- **Protocol stacks**: BLE, WiFi, etc.
- **File System**: FatFS, etc.
- **Device Management**: Exception Handling, Boot, etc.