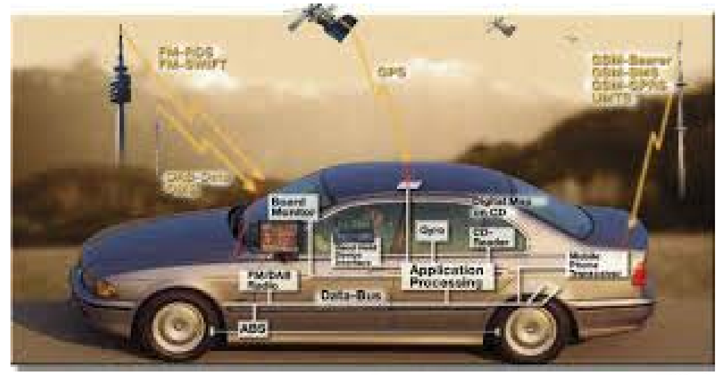
What Is an RTOS and Why Use One?

May, 2013

expresslogic

What is an Embedded System?

- Dedicated to a specific purpose
- Components:
  – Microprocessor
  – Application program
  – Real-Time Operating System (RTOS)
- RTOS and application programs usually stored in ROM
- Deterministic



The First Embedded System?

The first embedded system was probably developed in 1971 by the Intel Corporation which produced the 4004 microprocessor chip for a variety of business calculators. The same chip was used for all the calculators, but software in ROM provided unique functionality for each calculator.

- Maximum clock speed was 740 kHz
- Instruction cycle time: 10.8 µs[11] (8 clock cycles / instruction cycle)
- Instruction execution time 1 or 2 instruction cycles (10.8 or 21.6 µs), 46300 to 92600 instructions per second
- Separate program and data storage. Contrary to Harvard architecture designs, however, which use separate buses, the 4004, with its need to keep pin count down, used a single multiplexed 4-bit bus for transferring:
  - 12-bit addresses
  - 8-bit instructions
  - 4-bit data words
- Instruction set contained 46 instructions (of which 41 were 8 bits wide and 5 were 16 bits wide)
- Register set contained 16 registers of 4 bits each
- Internal subroutine stack 3 levels deep.

Real Time Systems

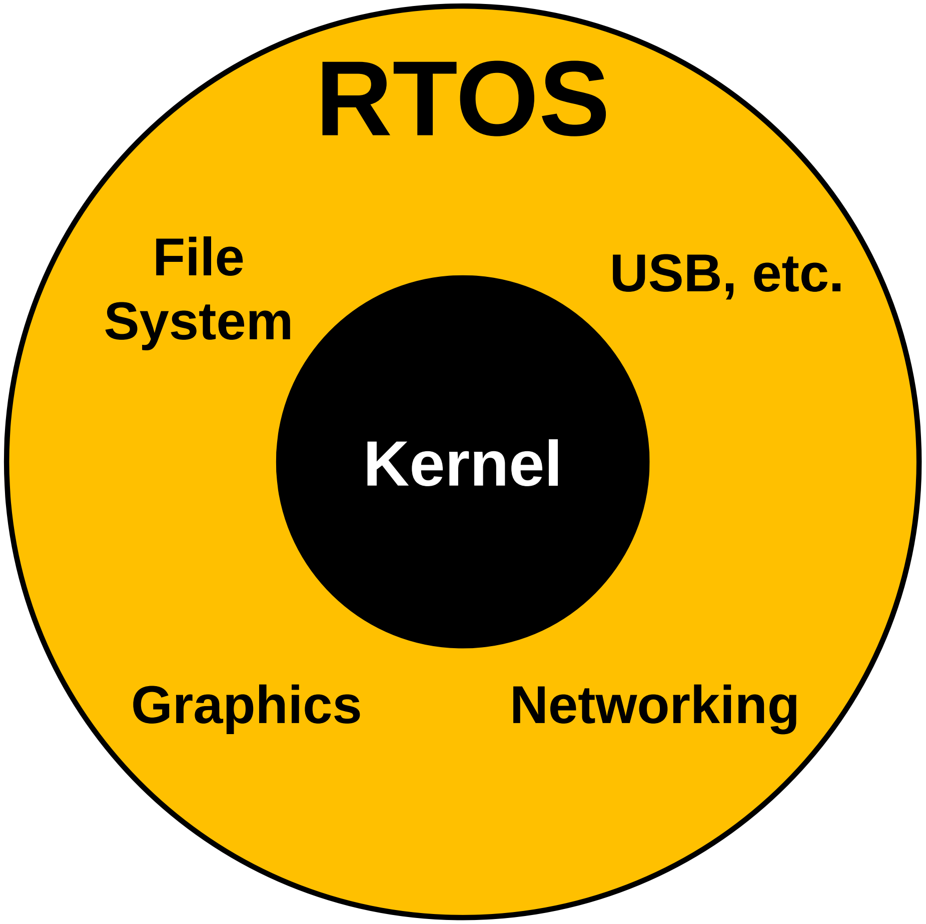
- Must respond to inputs or events within prescribed time limits
- Must operate within specified time constraints
- Important subclasses of Real Time:
  - *Hard Real Time*
    - Must meet deadlines 100% of the time
    - Generally true of safety-critical systems
  - *Soft Real Time*
    - Must meet deadlines under normal conditions
    - Generally true of consumer electronics

Determinism

- Time required to complete any function must be finite and predictable
- Maximum response time must be calculable and guaranteed
- Number of cycles required to execute a given operation must always be the same
- Execution can be interrupted, but interrupt latency and processing time must be bounded
- Not all systems require it
- Not all RTOSes deliver it

Real-Time Operating System

- **What is an RTOS?**
  - Kernel
  - File System
  - Networking
  - USB
  - Graphics
- **Kernel Components**
  - Scheduler
  - Thread Management
  - Message Queues
  - Semaphores
  - Mutexes
  - Timers
  - Memory Pools



expresslogic

What Does An RTOS Do?

- **Simplifies use of hardware resources**
  – Allocate memory
  – Service interrupts
  – Interface to devices
- **Provides library of services**
  – Schedule application (activate/suspend/resume)
  – Send/Receive “messages”
  – Get/Release “semaphores”
  – React to “events”

expresslogic

Why Use An RTOS?

- **RTOS Benefits**
  1. Better Responsiveness and Lower Overhead
  2. Simplified Resource Sharing
  3. Easier Development and Debugging
  4. Enabled Use of Layered Products
  5. Increased Portability and Maintenance
  6. Faster Time To Market

Better Responsiveness and Lower Overhead

- **Non-RTOS application program must “loop” or “poll” to check for need to perform a function – ie: process a received message**
  – Number of application functions determines time to poll
  – Response to need for service depends on polling time
  – Looping, checking, polling, state machine tracking all consume processor cycles and add to overhead
- **RTOS can “context switch” processor to required function and back**
  – RTOS performs context switch transparently to application
- **RTOS makes use of processor when application is waiting**
  – Multithreading
- **RTOS enables processor to spend more time in application**
  – Less time inefficiently managing application

Simplified Resource Sharing

- **Some processor resources must be shared among functions**
  - Memory
  - I/O Ports
  - Critical Sections of code

- **RTOS provides centralized mechanisms for arbitrating requests for resources**
  - Memory allocation/de-allocation at run-time
  - Semaphores and Mutexes to control single-use hardware or critical sections of software
  - Preemption-Threshold™ to help manage access to Critical Sections

Easier Development and Debugging

- **Development team members can operate asynchronously**
- **Application can be maintained more easily**
- **Debugger can display RTOS kernel objects for increased visibility into application behavior**
- **Applications can call service functions to perform operations rather than write and debug new code**
- **Application developers can avoid dealing with many interrupt details, timers, and other hardware resources**

Enabled Use of Layered Products

- **Layered products often depend on RTOS services for their operation**
  - File System
  - TCP/IP Network Stack
  - USB Stack
  - Graphics
  - 3^{rd} Party Products

Increased Portability and Easier Maintenance

- **Application talks to RTOS API, not specific hardware**
- **Application runs wherever RTOS runs**
- **Modular applications easily expanded and modified**
- **Commercial support for RTOS service functions**

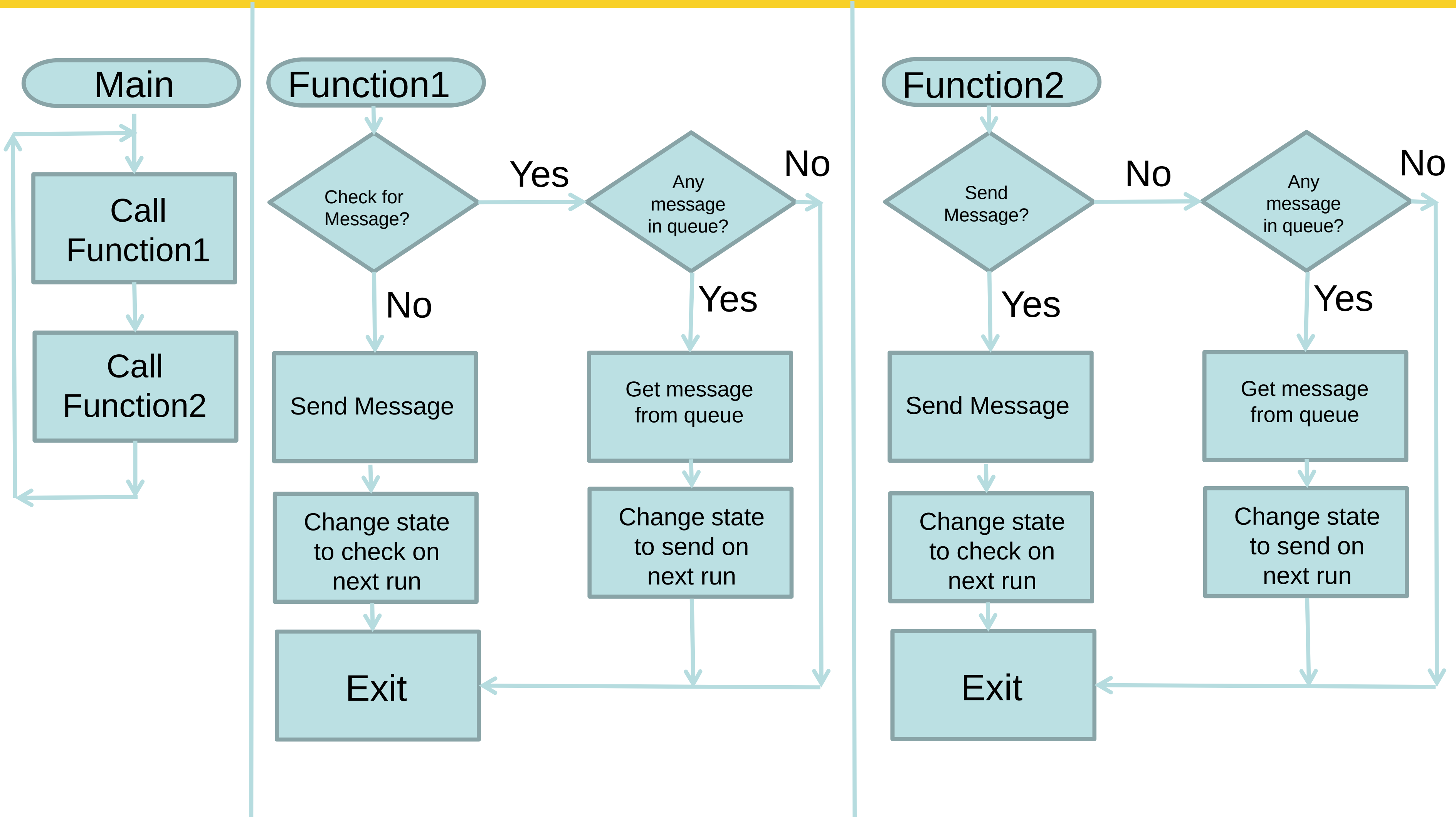
Who Doesn’t Need An RTOS?

- **Single-purpose applications**
- **Simple looping or polling applications**
- **Foreground-background applications**
- **Typically, <32KB applications**
- **No reason to overkill a solution**

Example: Non-RTOS

- **Simple application with two functions**
  - One sends a message to a buffer and checks for reply
  - Second one checks for message and if found, replies
- **Main loop to sequence back and forth between the functions to see if they have any messages to send or have received a message**
- **Each function must remember where it left off last time it ran (sent or received a message?)**
- **Nothing else can be done while routines are working or waiting**

Non-RTOS Example

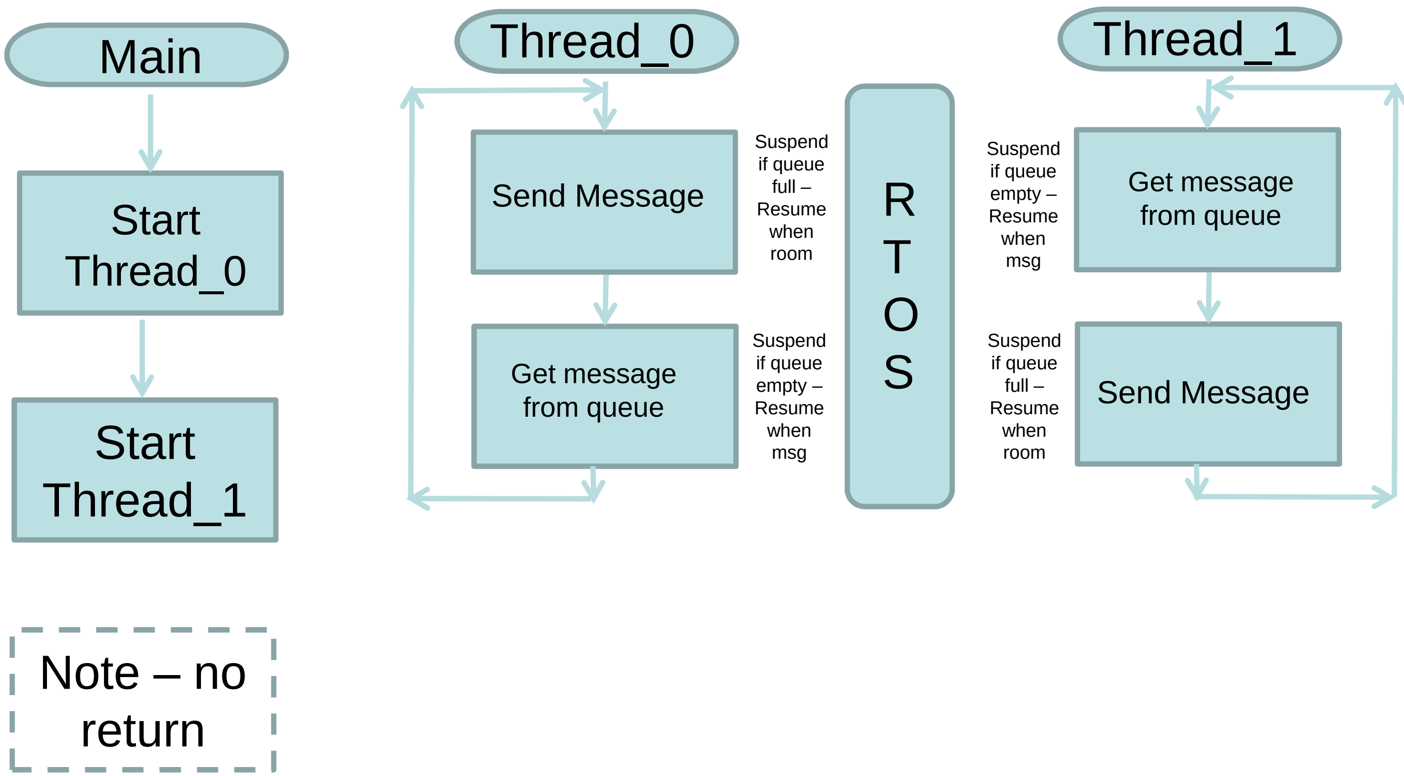


expresslogic

Example: RTOS

- **Routines become RTOS “threads”**
- **Messages get sent to, and retrieved from a “queue” managed by the RTOS**
- **Threads “suspend” if queue is empty**
- **Processor is free to do other work while queue remains empty**
- **Threads automatically “resumed” at point of suspension when message arrives in queue**

RTOS Example

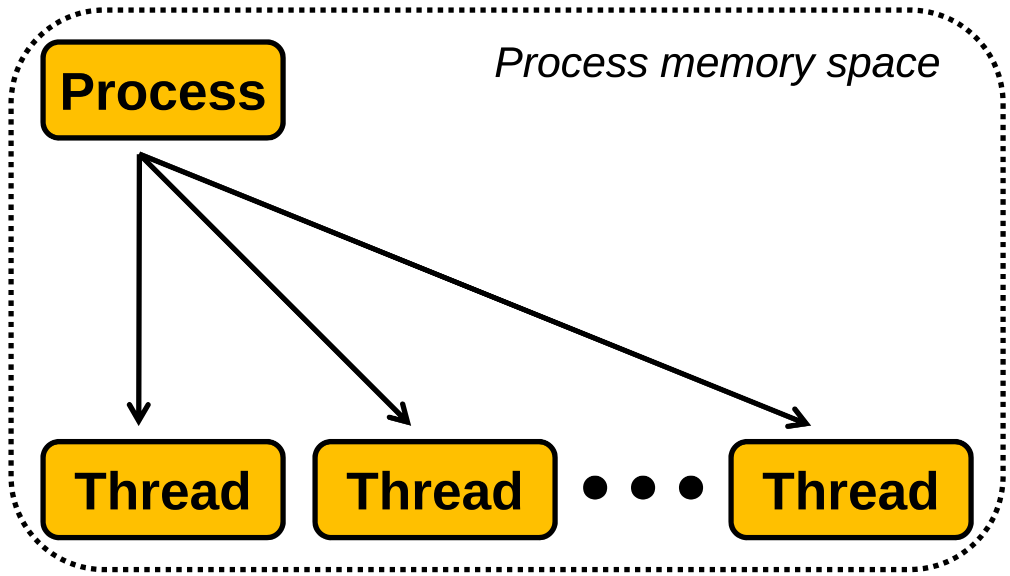


Processes, Tasks, and Threads

- **Process/Task**
  - Independent executable program with its own memory space
  - Multitasking: running several tasks/processes "concurrently"
  - A Process can have multiple threads
- **Thread**
  - Semi-independent program segment; multiple threads can share the same memory space
  - Multithreading: running several threads "concurrently"
  - Some RTOSes use "task" to mean "thread"

Threads and Priorities

- **Threads**
  - What is a thread?
    - Semi-independent program segment
    - Share same memory space
    - Run “concurrently”
  - How are threads used?
    - Modularize a program
    - Minimize stalls
  - Thread Services
    - Create, Suspend, Relinquish, Terminate, Exit, Prioritize
  - Thread States
    - READY, RUNNING, SUSPENDED, TERMINATED

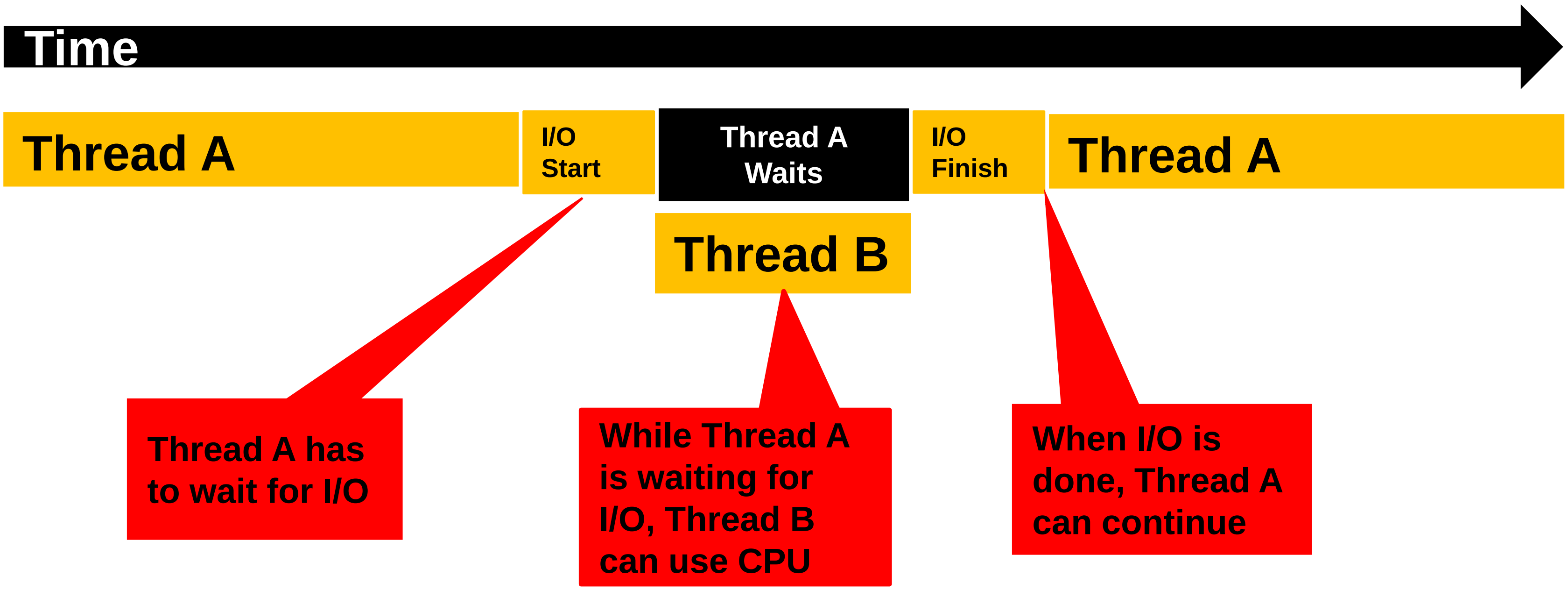


- **Thread Priorities**
  - Often 0-n, with 0 highest
  - Dynamic or Static
  - Equal priorities
    - Multiple threads at same priority
  - Unique priorities
    - Each thread has unique priority

Priority	
0	*Highest*
1	
2	
…	
n	*Lowest*

Multithreading

- Enables one part of an application to use the CPU while another part must wait
- Makes more efficient use of CPU than “waiting”
- Foreground/Background or Multiple threads



Context Switch

- **Thread Context**
  - Information critical to thread's operation
  - Register Contents, Program Counter, Stack Pointer
  - Saved when thread is preempted
  - Restored when thread is resumed

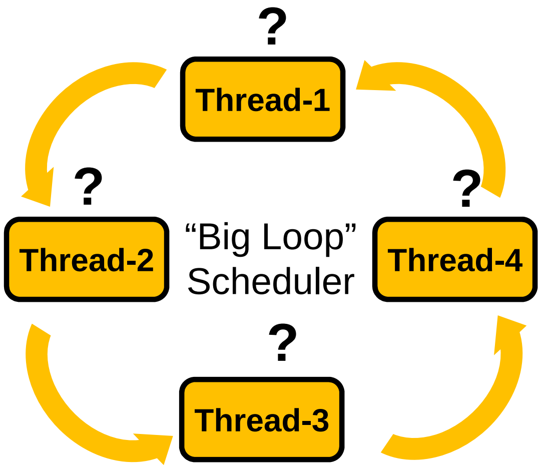
Registers	PC
	SP

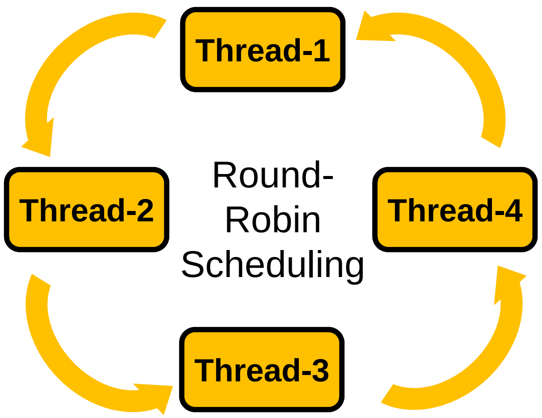
- **Context Switch**
  - Interrupt running thread and do something else
  - Result of preemption, interrupt, or cooperative service
- **What's involved in a context switch?**
  - See →

Step	Operation	Cycles
1	Save the current thread's context (ie: GP and FP register values and PC) on the stack.	20 - 100
2	Save the current stack pointer in the thread's control block.	2 - 20
3	Switch to the system stack pointer.	2 - 20
4	Return to the scheduler.	2 - 20
5	Find the highest priority thread that is ready to run.	2 - 50
6	Switch to the new thread's stack.	2 - 50
7	Recover the new thread's context.	20 - 100
8	Return to the new thread at its previous PC.	2 - 40
9	Other processing	0 - 100
	TOTAL	**50 - 500**

Types of Schedulers

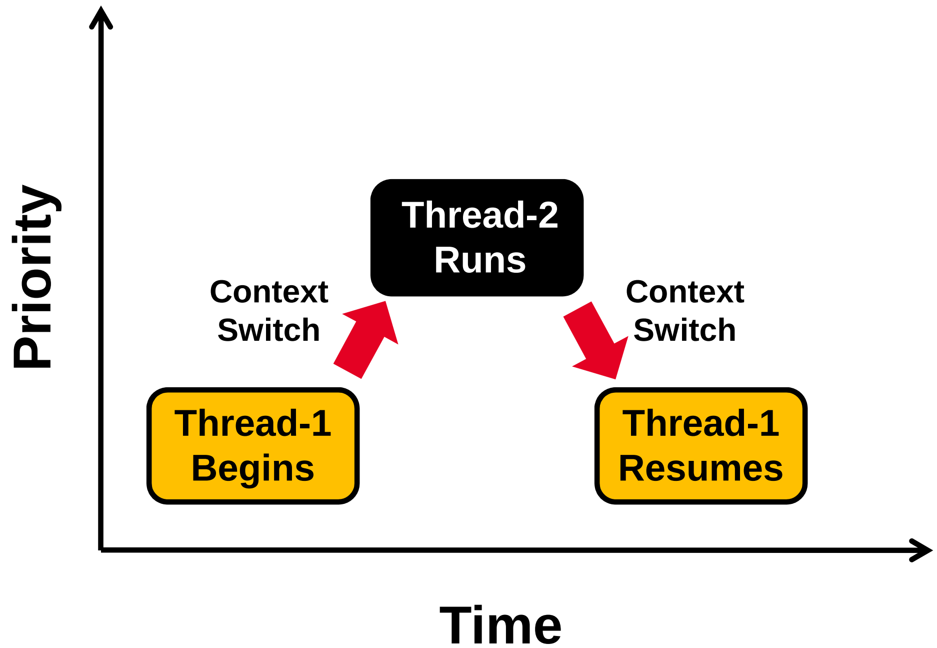
- **Big Loop Scheduling**
  - Each thread is polled to see if it needs to run
  - Polling proceeds sequentially, or in priority order
  - Inefficient, lacks responsiveness
- **Round-Robin Scheduling**
  - Cycle through multiple "READY" threads
  - Threads run to completion or blockage
  - May impose "time-slice" for each thread
- **Preemptive Scheduling**
  - Based on priority
  - Performs context switches
  - Manages thread states
    - Ready/Running
    - Suspended (Blocked/Sleeping/Relinquished)
    - Terminated





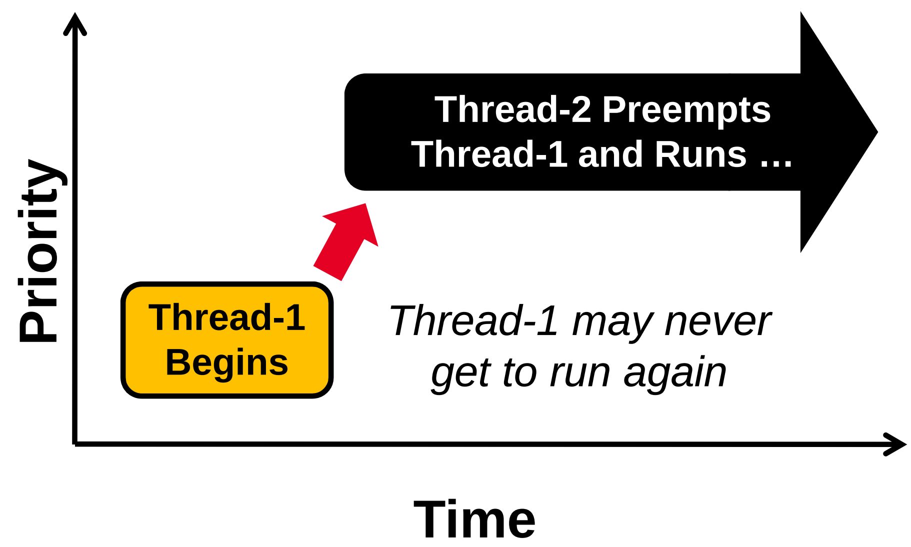
Preemptive Scheduling

- **Preemption**
  - Interruption for higher-priority activity
    - Interrupt
    - Thread
- **Preemptive Scheduling**
  - Always run highest priority thread that is READY to run
  - Maximum responsiveness
  - No Polling, so more efficient
  - Always results in a context switch



Preemptive Problems

- **Thread Starvation**
  - If a higher-priority thread is always ready, the lower priority threads never execute
- **Excessive Overhead**
  - From context switching
  - See example
- **Priority Inversion**
  - Higher-priority thread is suspended because a lower-priority thread has a needed resource

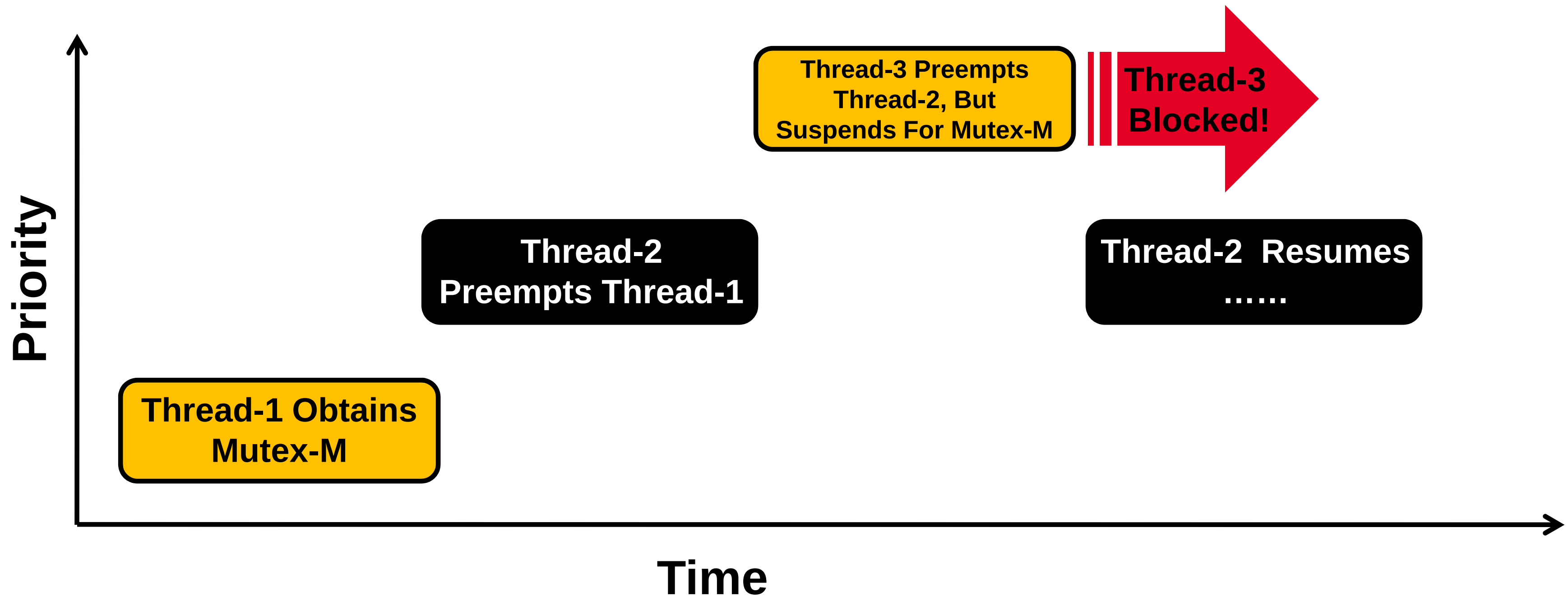


Priority Inversion

- **Occurs when higher priority thread is suspended because a lower priority thread has a needed resource**
- **May be necessary for 2 threads of different priorities to share a common resource**
- **Priority inversion time may become un-deterministic and lead to application failure**

expresslogic

Undeterministic Priority Inversion

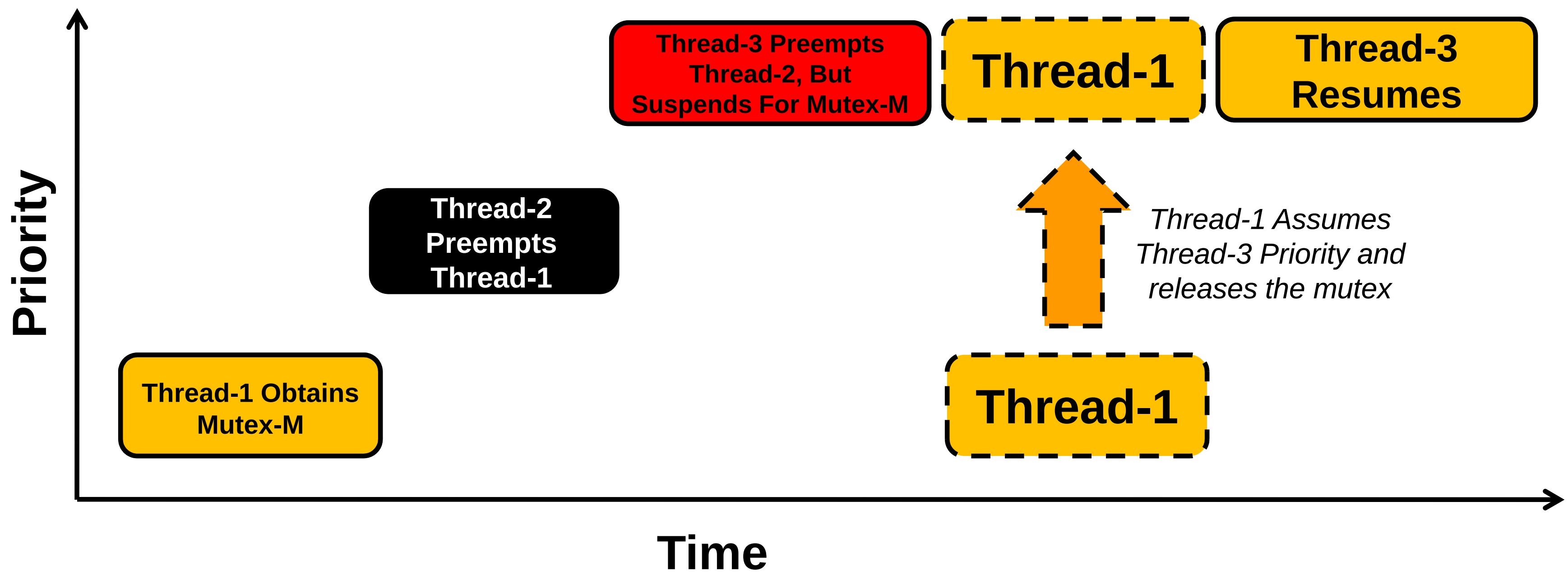


Even though Thread-3 has the highest priority, it must wait for Thread-2.

Preventing Priority Inversion

- **Proper design** of application run-time behavior and appropriate priority selections
- Lower priority threads can use **Preemption-Threshold** to block preemption from intermediate threads while sharing resources with higher priority threads
- Threads using mutex objects may utilize **priority inheritance** to eliminate un-deterministic priority inversion

Priority Inheritance



Thread-1 assumes the priority of Thread-3 until it is finished with Mutex-M

Preemption-Threshold™

- Another technique to avoid priority-inversion and reduce context switches
- Preemption-Threshold establishes a priority ceiling for disabling preemption – preemption requires a priority higher (lower number) than the ceiling
- For example, assume a thread's priority is 20, and its preemption threshold is set to 15
- Threads with priority lower than (larger number) 14 , even if higher than (smaller number) the running thread's priority (20), will not preempt the running thread

Preemption-Threshold

Priority

Priority	Comment
0	Preemption allowed for threads with priorities from 0 to 14 (inclusive)
:	
14	
15	Thread is **assigned Preemption-threshold = 15** [this has the effect of disabling preemption for threads with priority values from 15 to 19 (inclusive)]
:	
19	
20	Thread is assigned **Priority = 20**
:	
31	

Message **Queues**

- **What is a Message Queue?**
  - Data structure that holds messages
  - Means of message-passing among threads
  - Messages usually are inserted at rear of queue (FIFO) but can be inserted at front of queue if desired (LIFO)
  - Messages are removed from front of queue
  - Public resource—any thread can access any queue
  - Threads will suspend on queue full and queue empty

Semaphore

- **Efficient means of inter-thread communication**
  – Binary Semaphores
    - (0 or 1)
    - Only one occurrence (single-use resource)
  – Counting Semaphores
    - (0 - 0xFFFFFFFF)
    - Many occurrences (multiple-use resource)
- **Very Low Overhead**
- **Suspension on 0**
  – Suspended Threads Resumed in FIFO Manner
  – Optional Time-out on Suspension
- **No Maximum Number of Semaphores**

Semaphore Management API

- **Semaphore Create**
  - UINT **tx_semaphore_create** (TX_SEMAPHORE *semaphore_ptr, CHAR *name_ptr, ULONG initial_count);

- **Semaphore Get**
  - UINT **tx_semaphore_get** (TX_SEMAPHORE *semaphore_ptr, ULONG wait_option);

- **Semaphore Put**
  - UINT **tx_semaphore_put** (TX_SEMAPHORE *semaphore_ptr);

Mutex

- **Used to control thread access**
  - To critical sections
  - Or exclusive-use resources
  - Prevents interference with exclusive use
- **Similar to binary semaphore, but used solely for mutual exclusion, and not for event notification**
- ***"Get"* operation obtains mutex not owned by another thread**
  - Suspension if already owned by another thread
- ***"Put"* operation releases previously obtained mutex**

Mutex Management API

- **Mutex Create**
  - UINT **tx_mutex_create** (TX_MUTEX *mutex_ptr, CHAR *name_ptr,
    UINT priority_inherit);

- **Mutex Get**
  - UINT **tx_mutex_get** (TX_MUTEX *mutex_ptr, ULONG wait_option);

- **Mutex Put**
  - UINT **tx_mutex_put** (TX_MUTEX mutex_put);

Summary And Conclusions

- **What Is An RTOS**
  - Facility for managing application threads
- **Why Use An RTOS?**
  - Achieve more efficient use of CPU through multithreading
  - Modularize application development and maintenance
  - Simplify application porting