Keep learning.
It's the Washington Way.

EMBSYS 105
Programming with Embedded & Real-Time Operating Systems

Instructor: Nick Strathy, nstrathy@uw.edu

TA: Gideon Lee, gideonhlee@yahoo.com

Lecture 2 1/13/2020

PROFESSIONAL & CONTINUING EDUCATION

UNIVERSITY of WASHINGTON

Looking ahead

Date	Lecture number	Assignment
1/6	L1	A1 due* before L2
1/13	L2	A2 due before L3
1/20	Holiday (?)	
1/27	L3	A3 due before L4
2/3	L4	A4 due before L5
2/10	L5	A5 due before L7, Project due before L10
2/17	Holiday (?)	
2/24	L6	
3/2	L7	
3/9	L8	
3/16	L9	
3/23	L10 – Student presentations	

* Assignments are due Sunday night at 11:59 PM

Next week’s holiday(?)

Whereas:

- Next week is a holiday (MLK Day)
- We therefore have no class scheduled
- Which breaks momentum as our course is starting up
- And pushes our course end-date one week later

Be it resolved:

- **That we hold our lecture next week, anyway**
- That attendance not be required (you can watch the recording later)

All those in favor: you don’t need to say a word.

All those opposed: let me know by end-of-day tomorrow

Previous lecture (L1) overview

- Embedded Operating System Concepts Part 1 (Labrosse section p 45: *Real-Time Systems Concepts*)
  - Foreground/Background Systems
  - Shared resources
  - Critical sections
  - Multitasking
  - Tasks
  - Context switching
- ARM Review
  - States and Modes
  - Registers
  - Assembly language instructions
  - Interrupt semantics
- Assignment 1 – uDebugger tool

Current Lecture (L2) Overview

- Revisit LDREX, STREX
- Embedded Operating System Concepts Part 2 (Labrosse Chapter 2)
  - Multitasking OS concepts
    - The kernel, Preemptive and non-preemptive kernels, The scheduler, Scheduling algorithms, Task states, Parameters used in scheduling – quantum, priority, Priority inversion
  - Real time OS concepts
    - Definition, Hard/soft real time, Jitter, OS timer tick, Interrupt latency
- Context switching on ARM Cortex (Yiu 10.4, 10.5)
- Assignment 2: Context Switch (due in 1 week)

Assignment Grading Policy

- Canvas operates on a points system and does not provide a pass/fail option.
- However Canvas provides complete/incomplete, so your submissions will be graded as complete or incomplete with zero (0) points.
- You may resubmit any assignment to make it complete, except see below:
- Submissions will not be accepted after a solution has been released.
- Pass threshold corresponds roughly anything higher than C+.

Revisit LDREX/STREX

Exclusive memory access (Yiu Section 5.6.2, Table 5.21)

- The goal is to accomplish the following *exclusive access sequence* atomically:
  - Get the contents of memory location x
  - If contents==0
    - Success
    - Store 1 in location x
  - Else
    - Failed (try again or give up)

Revisit LDREX/STREX

Potential Problems:

- After memory location x has been read and before it has been written with value 1:
  - An interrupt may occur potentially allowing another thread to access memory location x.
  - An interrupt handler may exit potentially allowing another thread to access location x.
  - Or another *processor or bus master* may access location x

Revisit LDREX/STREX

What LDREX Rt, [Rn] does:

- Copies the contents of location [Rn] into Rt
- Sets the processor's Local Exclusive Access Monitor bit to indicate that an exclusive access is in progress.
- If a bus level Exclusive Access Monitor bit is present it will also be set.

Revisit LDREX/STREX

What STREX Rd, Rt, [Rn] does:

- Attempts to copy the contents of Rt to memory location [Rn], returns status in Rd (success: Rd==0, failure: Rd==1)
- Resets the processor's Local Exclusive Access Monitor bit.
- If a bus level Exclusive Access Monitor bit is present it will also be reset.

Revisit LDREX/STREX

STREX fails if:

- The bus level exclusive access monitor (if present) returns an exclusive fail response.
- The local exclusive access monitor is not set which can happen if:
  - Incorrect exclusive access sequence (LDREX wasn’t executed first)
  - An interrupt entry or exit occurred since LDREX was executed
  - Execution of a CLREX occurred which clears the local exclusive access monitor.

Embedded operating system concepts

The Kernel

- The part of a multitasking OS responsible for task management
- Schedules tasks – does context switching
- Provides task synchronization services like semaphores, task communication services like mailboxes and queues (more on that later), task bookkeeping, etc.
- Requires CPU time to perform its functions but considered a good trade off for the gain in system manageability
- Well designed kernel uses 2-5% of CPU time leaving 95-98% of CPU time for application tasks

Embedded operating system concepts

Two kernel types

- Non-preemptive
  - Tasks complete their steps before surrendering the CPU to another task
- Preemptive
  - the kernel may for various reasons preempt one task before it has completed its steps to allow another task to have control of the CPU

Embedded operating system concepts

Non-preemptive kernels

- A simple approach often sufficient for simple embedded systems running on limited hardware
- Similar to foreground-background systems in that tasks execute in the background and are interrupted by ISRs in the foreground
- More flexible than foreground-background systems in that tasks don't have to execute in the same order every time through the super loop
- The kernel can change the priority of tasks dynamically in an ISR so that the highest priority task runs next – but it still waits till the current task is done
- The current task is never preempted even if a higher priority task is next – it only relinquishes the CPU when it has finished its steps
- Consequently, response time is non-deterministic

Embedded operating system concepts

Preemptive kernels

- This type of kernel can preempt a task and switch context to another task even if the first task is not ready to relinquish the CPU
- In priority based scheduling a higher priority task that is ready to run should always preempt a lower priority task immediately
- Therefore execution of the highest priority task in the system is deterministic

Embedded operating system concepts

The Scheduler

- That part of the kernel that has the job of selecting the next task to run
- Various algorithms optimize for different scheduling goals
- All scheduling algorithms seek
  - Fairness – each task gets its fair share of CPU – no CPU “starvation”
  - “Fair” depends on the context and goals – e.g. priority based schemes need to account for varying task priority levels – higher priority tasks must have precedence over lower
  - Being fair is easier without priorities however real time constraints may not get met
  - Policy enforcement – need to be clear on the policies
  - Balance – keeping all parts of the system busy

Embedded operating system concepts

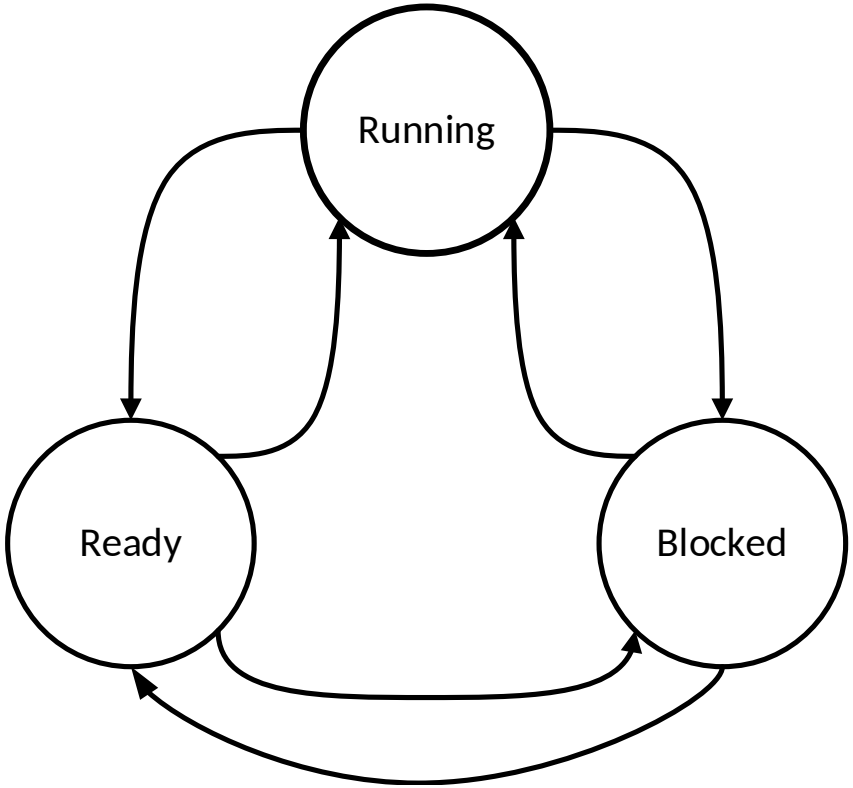
Scheduling algorithms

- Batch scheduling – example: processing big data
  - Throughput – maximize jobs per hour
  - Turnaround time – minimize time between submission and completion
  - CPU utilization – keep the CPU busy all the time
- Interactive scheduling – example: web server
  - Response time – respond to requests quickly
  - Proportionality – meet users' expectations/perceptions of how long a job "should" take
- Real time – example: streaming media – MP3 player
  - Meet deadlines
  - Predictability - deterministic

Embedded operating system concepts

Task states

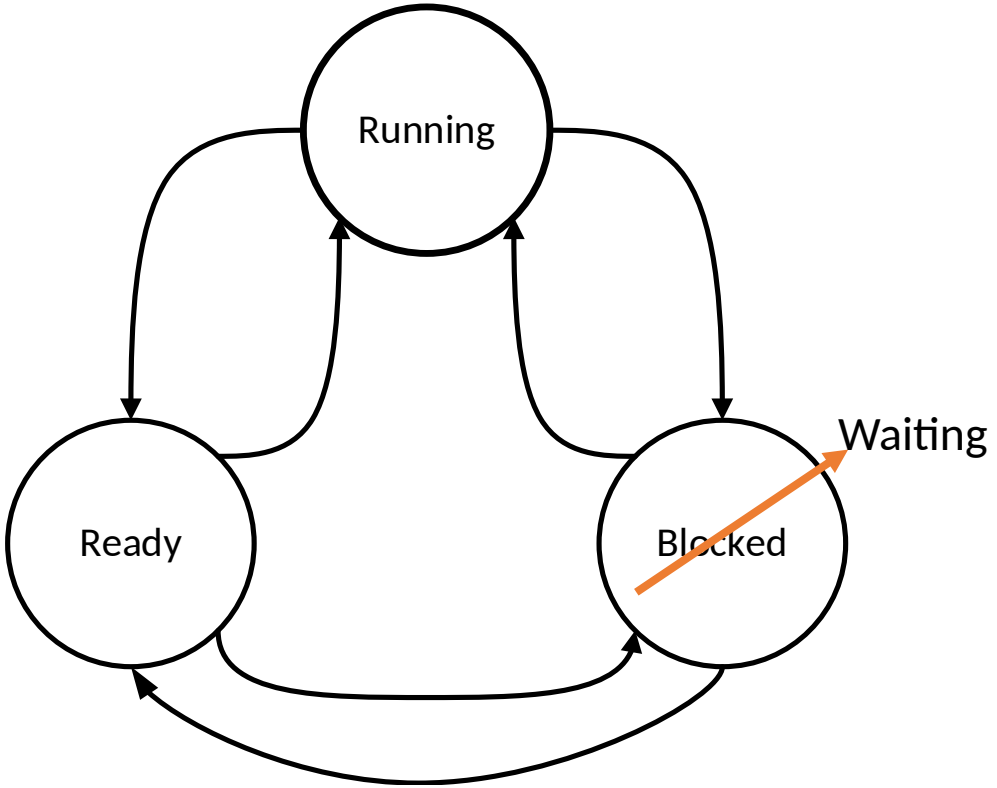
- Simplest preemptive kernel has 3 task states – Running, Ready, Blocked
- Running - only 1 task at a time is in this state (on a single-core system) and it has control of the CPU
- Blocked – tasks in this state are not currently capable of running – e.g. waiting for a shared resource to become available
- Ready state – tasks in this state are currently capable of running, just waiting for control of the CPU



Embedded operating system concepts

MicroC/OS-II terminology note on Task states

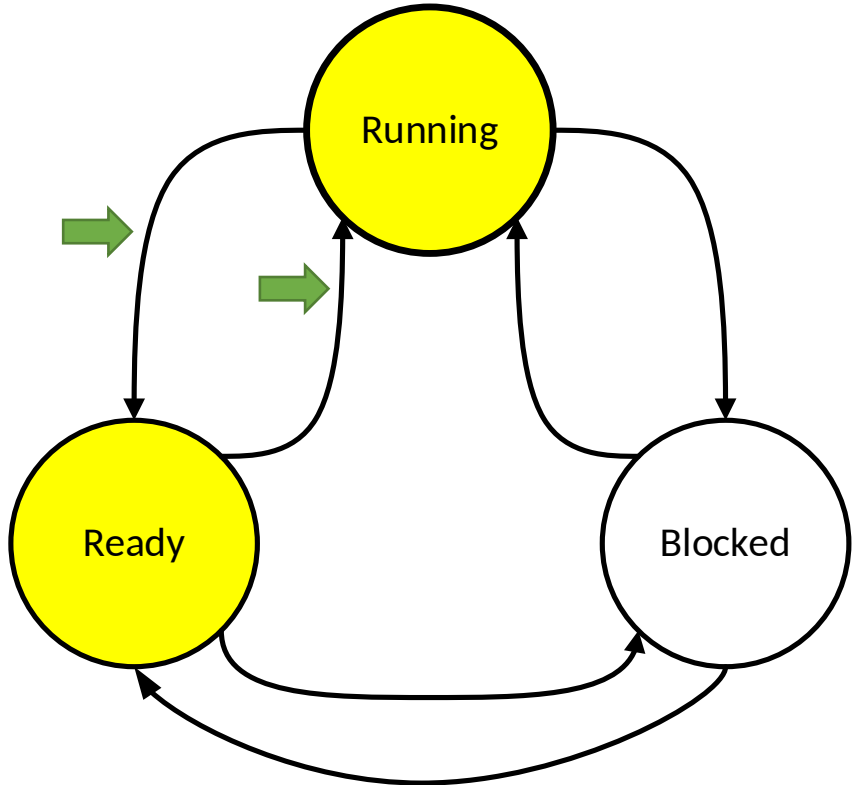
- uCOS textbook calls the “Blocked” state the “Waiting” state
- Could be confusing: both Ready and Blocked tasks are “waiting” but for different things
- Blocked tasks are waiting for something to make them capable of running
- Ready tasks are capable of running and are just waiting to be assigned control of the CPU



Embedded operating system concepts

Some reasons for state transitions

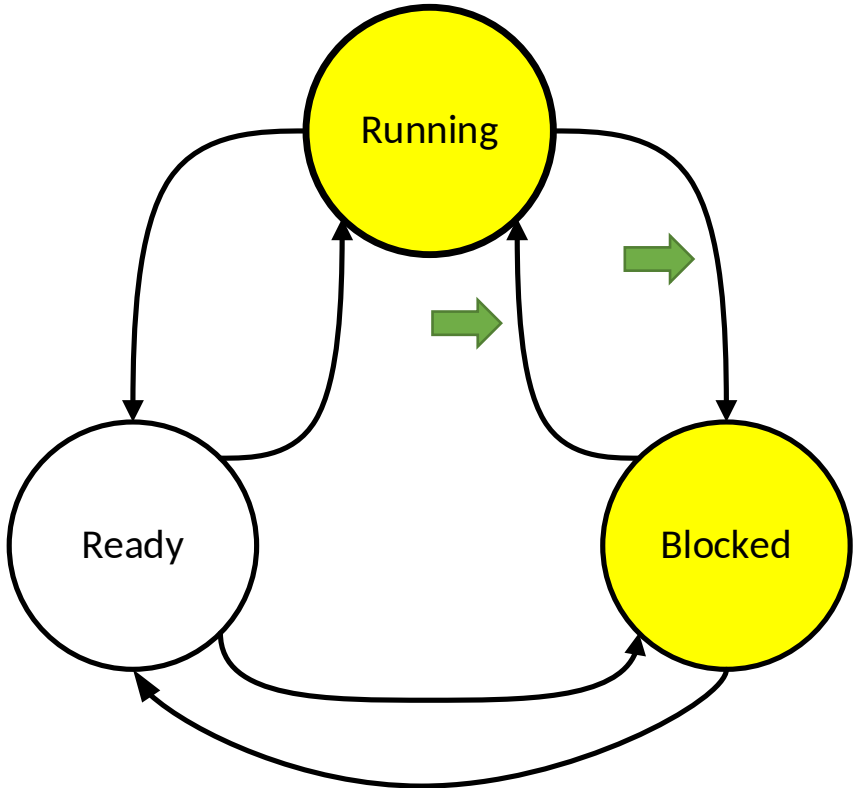
- Ready Running
  - Task gets its “turn” for control of CPU
- Running Ready
  - Task’s “time slice” expires
  - Task currently has no work to do and “yields” the CPU
  - Task is preempted by a higher priority task



Embedded operating system concepts

Some reasons for state transitions

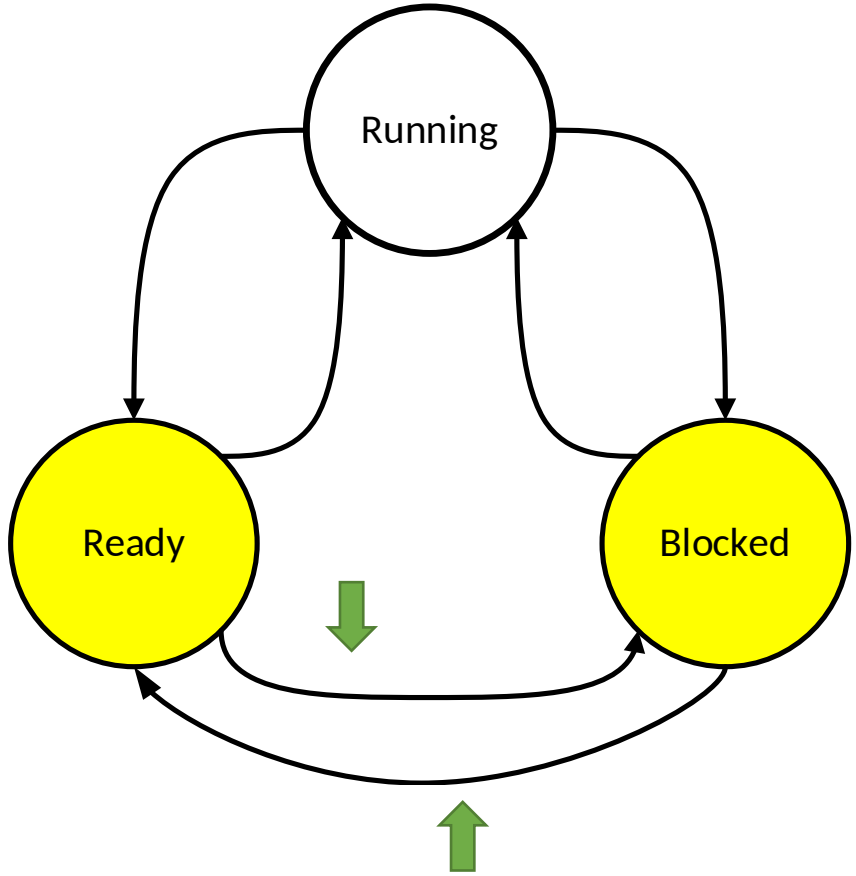
- Running Blocked
  - Task attempts to enter a critical section currently occupied by another task
  - Task chooses to delay itself
- Blocked Running
  - A task exits a critical section allowing a top priority task to enter the CS
  - The blocked task is in a “priority inversion” situation which requires it to be unblocked and run (more on that later)



Embedded operating system concepts

Some reasons for state transitions

- Ready Blocked
  - Another task such as a shell suspends the task, e.g. Linux “suspend” command
- Blocked Ready
  - One task exits a critical section which unblocks a waiting task which if it is not top priority will become ready
  - A task’s self imposed delay expires



Embedded operating system concepts

Parameters used in scheduling

- Quantum – or time slice
  - Round Robin Scheduling:
    - Each task gets to control the CPU for a predetermined time period or quantum before the scheduler assigns the CPU to the next ready task
  - Too short quantum
    - results in frequent context switches
    - Context switches are time consuming so can result in too high a percentage of CPU time spent on unproductive work
  - Too long quantum
    - Tasks can get a lot done in their quantum but
    - Tasks may miss deadlines if they have to wait too long for their turn

Embedded operating system concepts

Parameters used in scheduling

- Priority
  - With “pure” priority scheduling, the highest priority ready task *always* gets immediate control of the CPU
  - Requires that hi-pri tasks get blocked or self-delay to allow lower priority tasks to run
  - Otherwise the globally highest priority task will run 100% of the time
  - Pure priority scheduling can result in **priority inversion …**

Embedded operating system concepts

Priority inversion

- Occurs when a higher priority task ends up blocked indirectly by a lower priority task
- Minimum conditions required:
  - Represent a task as an ordered pair:
    (name, priority)
  - 3 tasks (A,1), (B,2), (C,3)
  - Priority 1 > 2 > 3
  - 1 shared resource: R

- A priority inversion sequence:
  - (C,3) is running and acquires R
  - (A,1) preempts (C,3), then blocks waiting for R held by (C,3) (this is not yet priority inversion)
  - (B,2) becomes the highest priority ready task and therefore resumes executing
  - We now have priority inversion:
  - Usually OK: (B,2) prevents (C,3) from running
  - Not OK this time: By preventing (C,3) from running, (B,2) indirectly prevents (A,1) from running

Embedded operating system concepts

Priority inversion

Time -->	t0	t1	t2	t3	t4	t5
Task A, pri=1	blocked	blocked	unblocked by some event	running, try to acquire R	blocked waiting for R	blocked waiting for R
Task B, pri=2	blocked	blocked	unblocked by some event	ready	running: priority inversion	running: priority inversion
Task C, pri=3	running	acquire R	ready	ready	ready	ready
Resource R	available	owner=C	owner=C	owner=C	owner=C	owner=C

Embedded operating system concepts

Priority inversion solution

Priority inheritance

- When (A,1) attempts to acquire resource R held by (C,3), the priority of (C,3) is temporarily boosted to higher than (A,1)

- Now the sequence becomes:
  - (C,3) is running and acquires resource R
  - (A,1) preempts (C,3), then blocks waiting for R held by (C,3) however C's priority is temporarily raised higher than (A,1) thus (C,~~3~~ 0)
  - (B,2) is no longer the highest priority ready task therefore (C,0) continues to run until it releases resource R whereupon its priority is reduced back to 3 (C,~~0~~ 3)
  - When R becomes available, (A,1) is unblocked and resumes since it is the highest priority ready task.

Embedded operating system concepts

Time -->	t0	t1	t2	t3	t4	t5
Task A, pri=1	blocked	blocked	unblocked by some event	running, try to acquire R	blocked waiting for R	blocked waiting for R
Task B, pri=2	blocked	blocked	unblocked by some event	ready	running: priority inversion	running: priority inversion
Task C, pri=3	running	acquire R	ready	ready	ready	ready
Resource R	available	owner=C	owner=C	owner=C	owner=C	owner=C

Priority inversion

Time -->	t0	t1	t2	t3	t4	t5
Task A, pri=1	blocked	blocked	unblocked by some event	running, try to acquire R	blocked waiting for R	running, acquire R
Task B, pri=2	blocked	blocked	unblocked by some event	ready	ready	ready
Task C, pri=3	running	acquire R	ready	ready, pri=0	running, release R, pri=3	ready
Resource R	available	owner=C	owner=C	owner=C	owner=C	owner=A

Priority inversion avoidance via *priority inheritance*

Embedded operating system concepts

- Scheduling with a blend of quantum and priority
  - Pure Round Robin – no priorities – just use time slices
  - Priority tiers - at start of a new quantum, current highest priority runs, else Round Robin for equal priority tasks (priority tiers)
  - Non-preemption with priority - at start of new quantum, current highest priority runs, but no preemption till end of quantum even if higher priority task becomes ready during the quantum (via ISR)
  - Pure priority scheduling – highest priority always runs, period.

Embedded operating system concepts

Real-Time OSes

- **Definition**: An RTOS is an operating system intended to serve real-time applications that process data as it comes in, typically without buffering delays. Processing time requirements (including any OS delay) are measured in tenths of seconds or shorter. (Wikipedia)
- *Hard* real-time OS: meets deadlines deterministically – guaranteed response times – must determine worst case performance
- *Soft* real-time OS: can usually or generally meet deadlines – extreme delays are occasionally possible but are considered acceptable

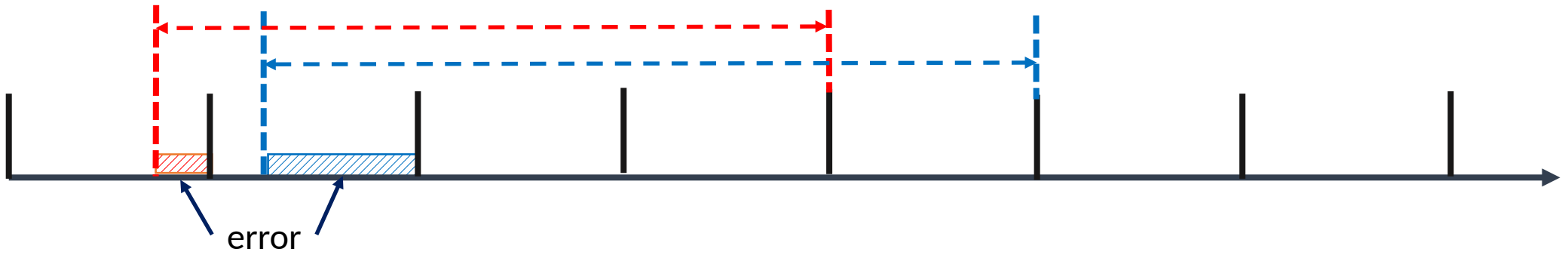
Embedded operating system concepts

Real-Time OSes

- **Jitter**: the variability in time taken to accept and process a given unit of work – inherent in all digital systems. Worst case jitter delays have to meet real time specifications of the application.
- **The OS timer tick**:
  - The “heartbeat” of the OS
  - OS bookkeeping is updated – CPU, memory usage per task, etc.
  - Update the state of tasks that are waiting for events
  - Call scheduler to determine which task should be the “current” task
  - Turn control of CPU over to current task

Embedded operating system concepts

Example: timer jitter



- Tick interrupt frequency, say 1 kHz
- If we make a 3 msec timer using ticks from this clock there will be an error between 0 and 1 msec depending on where in the tick interval we start counting ticks.
- To keep the relative error small, either time larger intervals (say > 1 sec) or use a higher frequency clock.

Embedded operating system concepts

Real-Time OSes

Interrupt latency

Latency for a particular interrupt depends on (quoting Simon)

1. The longest period of time during which that interrupt (or all interrupts) are disabled.
2. The time it takes to execute any interrupt routines for interrupts that are of higher priority than the one in question.
3. How long it takes the processor to stop what it is doing, do the necessary bookkeeping, and start executing instructions within the interrupt routine.
4. How long it takes the interrupt routine to save the context and then do enough work that what it has accomplished counts as a “response”

Embedded Operating System Concepts

Summary

- Multitasking OS concepts
  - The kernel
  - Preemptive and non-preemptive kernels
  - The scheduler, Scheduling algorithms
  - Task states
  - Parameters used in scheduling – quantum, priority
  - Priority inversion
- Real time OS concepts
  - Definition
  - Hard/soft real time
  - Jitter
  - OS timer tick
  - Interrupt latency

“Don’t just do something – sit there!”

Jean Labrosse

- Break
- Quiz

After the break

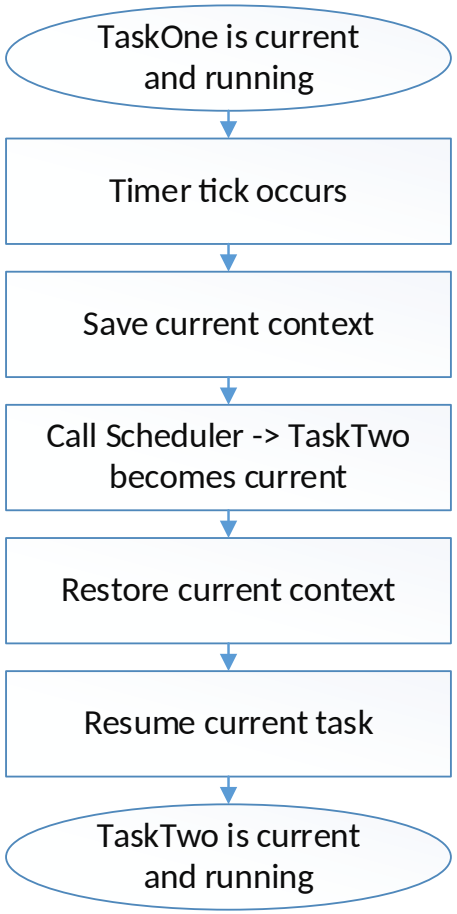
- Revisit Assignment 1
- Context switch

Context switching on Cortex-M4

Context switching (Assignment 2)

- Let’s look at a basic context switch scenario:
  - 2 tasks: taskOne, taskTwo
  - Timer tick interrupt occurs every 200 msec
  - On each tick,
    - Save the context of the current task in its own stack
    - Call a simple scheduler that will select the other task and make it the current task
    - Restore the context of the current task from its stack
    - Resume executing the current task
  - Tasks will run in Thread Mode, use Main SP, privileged level
  - Timer tick interrupt occurs in Handler mode, uses Main SP, privileged level (as do all interrupts)

Context switch flow diagram



Current Task:

Only one task at a time can run.

This is the **current task**.

Code for tasks to execute

```
// taskOne counts up from 0.
// Never exits.
//
void taskOne(void)
{
    int count = 0;
    while(1)
    {
        printString("task one: ");
        print_uint32(count++);
        printString("\n");
        int i;
        for(i=0;i<10000;i++);  // delay
    }
}
```

```
// taskTwo counts down from 0xFFFFFFFF
// Never exits.
//
void taskTwo(void)
{
    int count = 0xFFFFFFFF;
    while(1)
    {
        printString("task two: ");
        print_uint32(count--);
        printString("\n");
        int i;
        for(i=0;i<10000;i++);  // delay
    }
}
```

One stack per task

```
// Default size of stacks.
#define STACKSIZE 256

// Allocate space for two stacks
int stackOne[STACKSIZE];
int stackTwo[STACKSIZE];

int stackOneSP;         // Value of task 1's stack pointer
int stackTwoSP;         // Value of task 2's stack pointer
```

Key variables in Context Switch Homework

- **taskID**: an integer, either 1 or 2 indicating the current task ID.
- **stackOneSP**: 32-bit word to store the SP register value for taskOne when it is switched out (not running)
- **stackTwoSP**: 32-bit word to store the SP register value for taskTwo when it is switched out (not running)
- **currentSP**: 32-bit word which gets set to one of the above values by the Scheduler when switching to a new context.

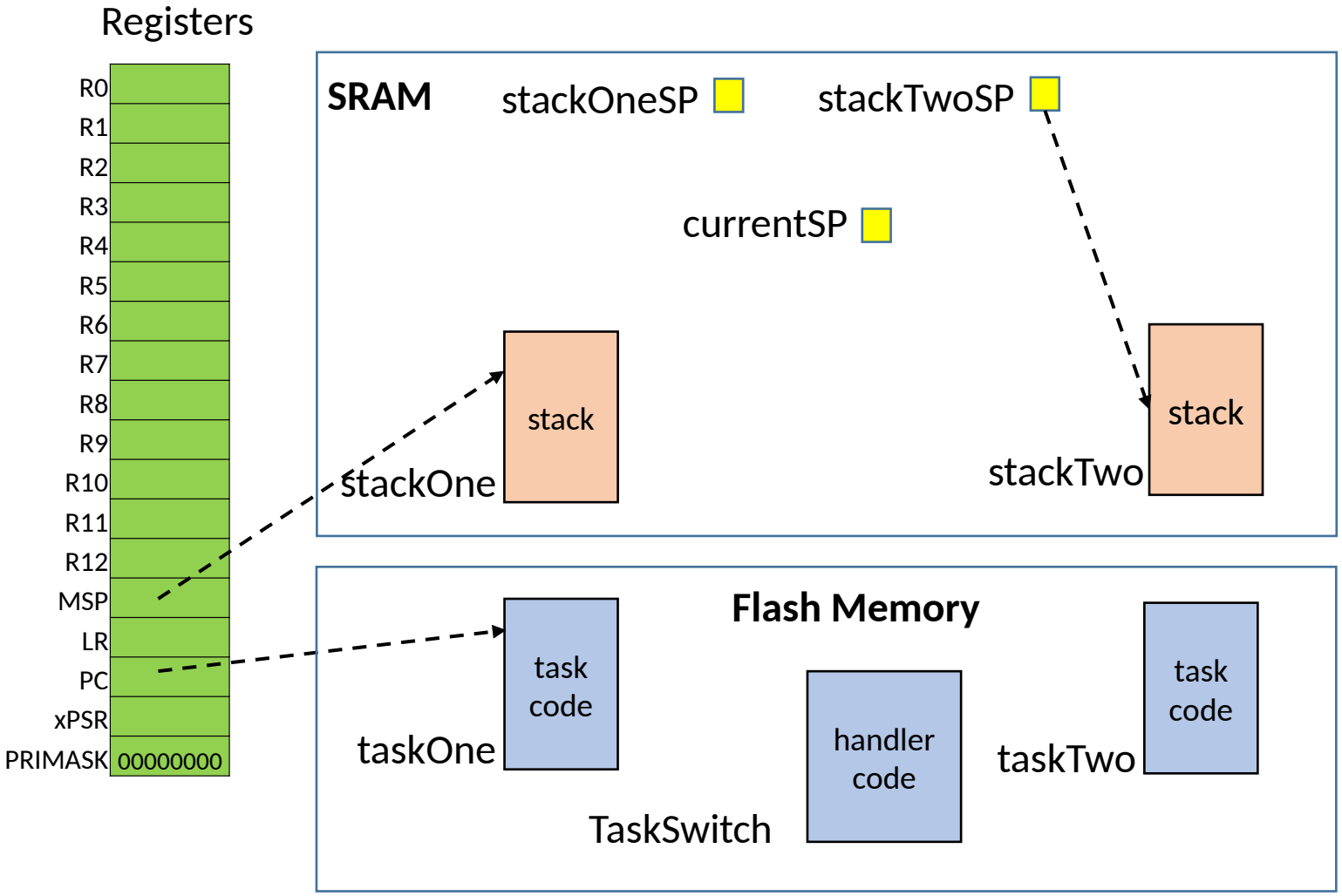
PendSV – a special Cortex-M4 exception

- PendSV is a special Cortex-M exception intended for context switching
- Always triggered explicitly *by software* (in SysTick_Handler)
- Convention requires that PendSV priority be set to lowest value (0xFF)
- The reason a context switch should have lowest priority is that it should never happen while any interrupt other than PendSV is being handled
- In other words, all other interrupts must finish being serviced before a context switch can take place

Context switch trace (Cortex-M4)

1 **taskOne is current and running**

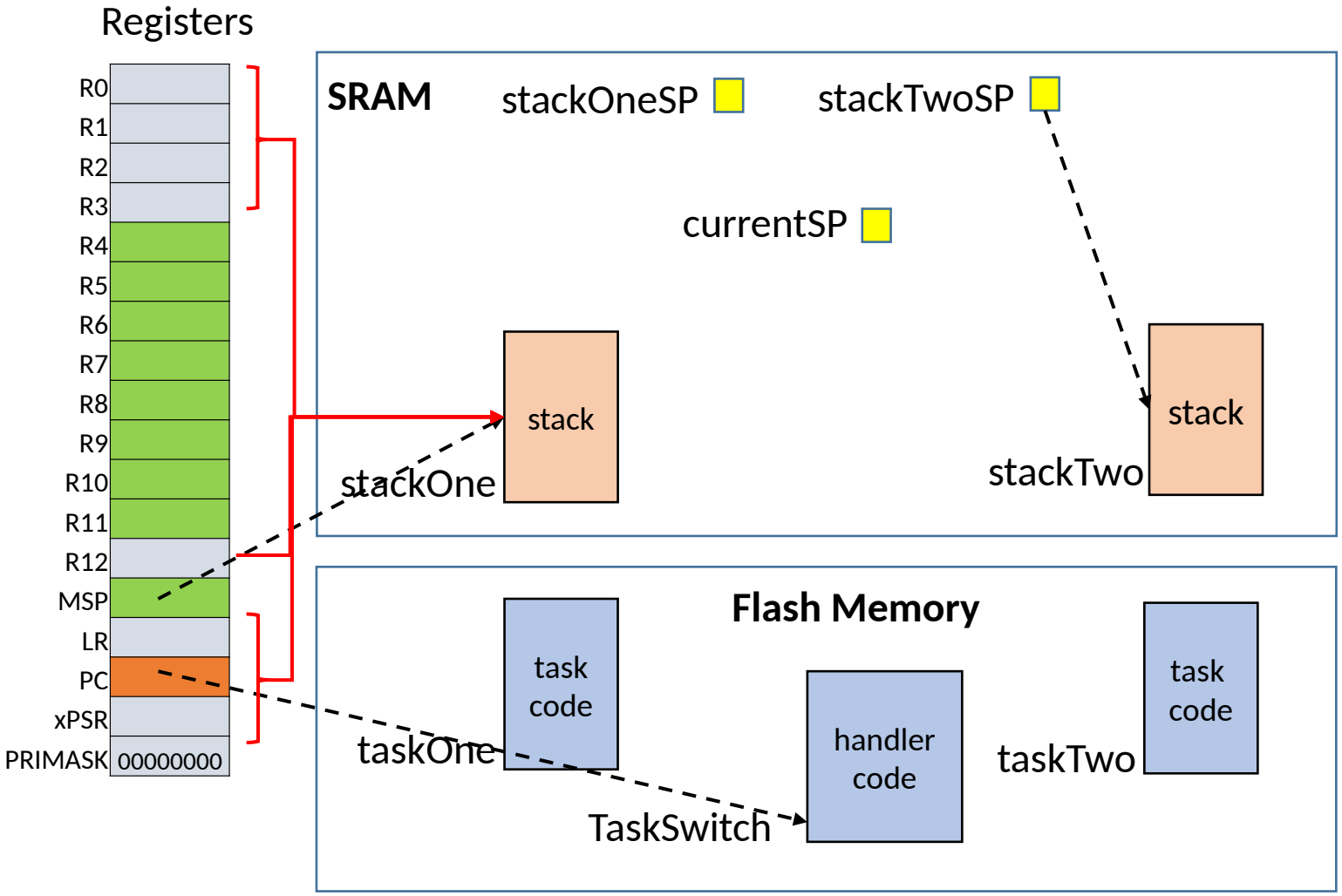
- In Thread mode
- MSP points somewhere in stackOne
- PC points somewhere in taskOne code
- PRIMASK is 0 (interrupts enabled)
- stackTwoSP points to saved context of taskTwo



Context switch trace (Cortex-M4)

2 **Timer tick occurs**

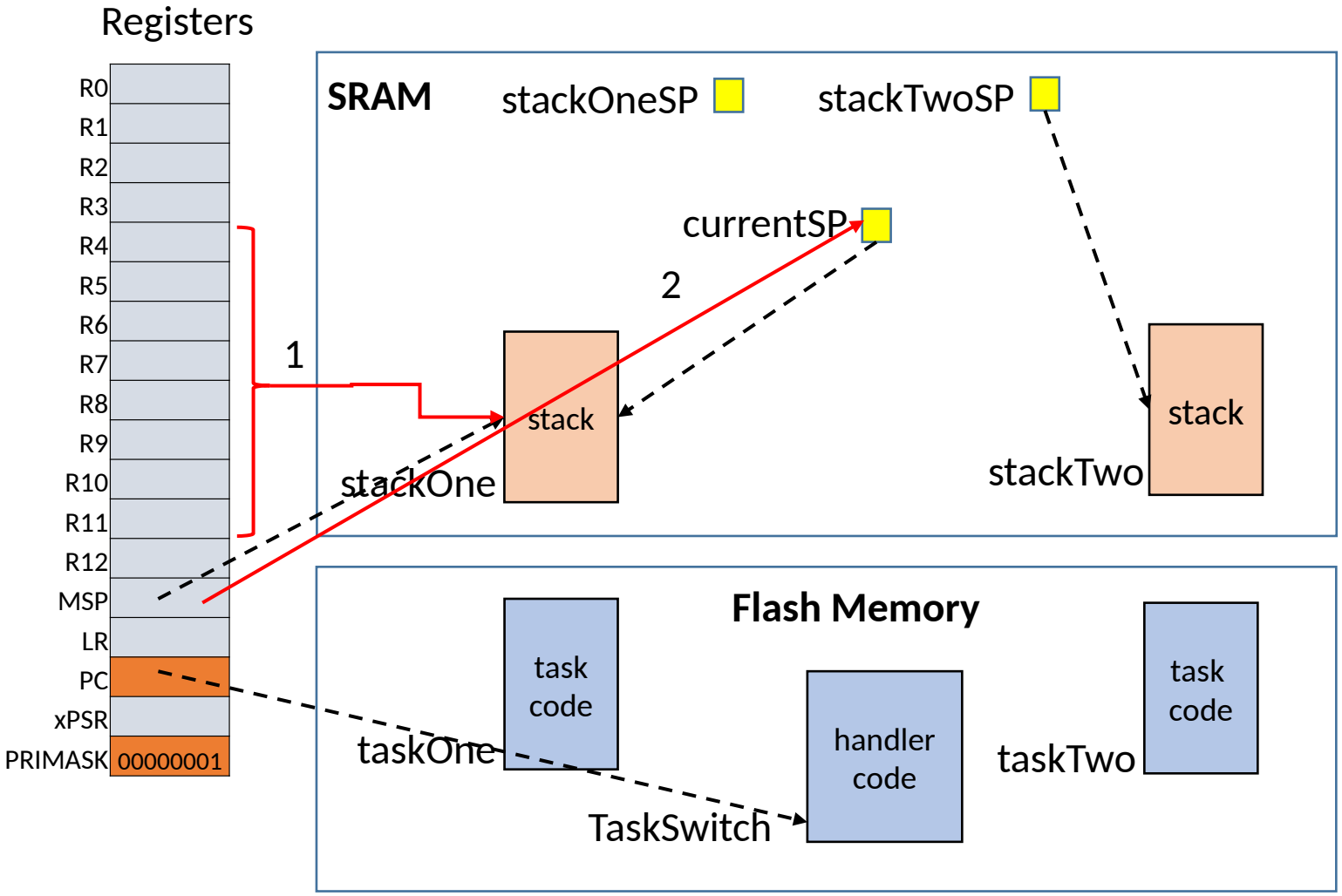
- R0-R3, R12, LR, PC, xPSR are stacked by NVIC
- SysTick_Handler triggers a PendSV interrupt and returns
- PendSV immediately enters TaskSwitch, “re-using” the stack frame already saved for the timer interrupt (“tail-chaining”)



Context switch trace (Cortex-M4)

3 **Save remaining context**

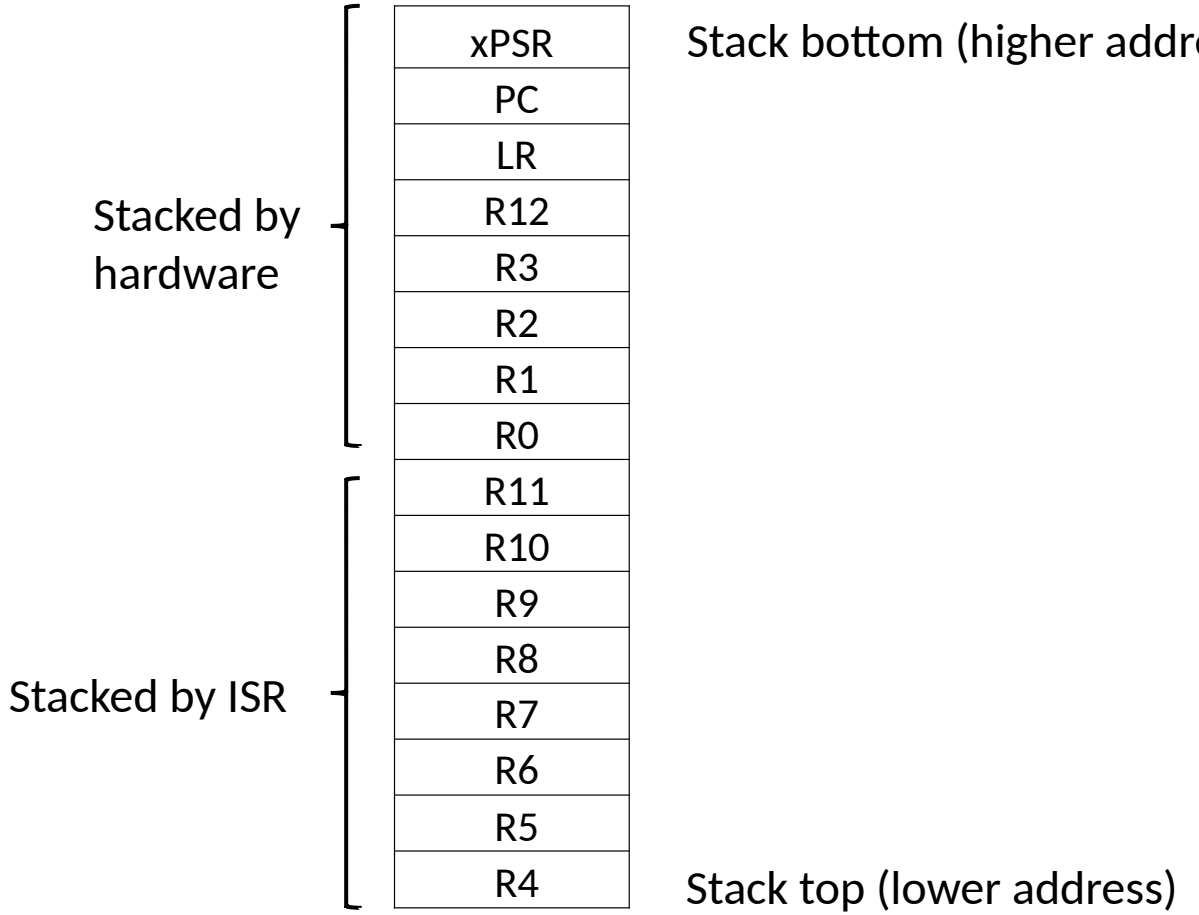
- Disable interrupts with CPSID I (PRIMASK = 1)
- R4-R11 are stacked by TaskSwitch handler
- MSP points to saved context of taskOne
- MSP is saved to currentSP



Contents of saved context in stackOne

- R13 (MSP) is not saved in the stack – we save it to currentSP

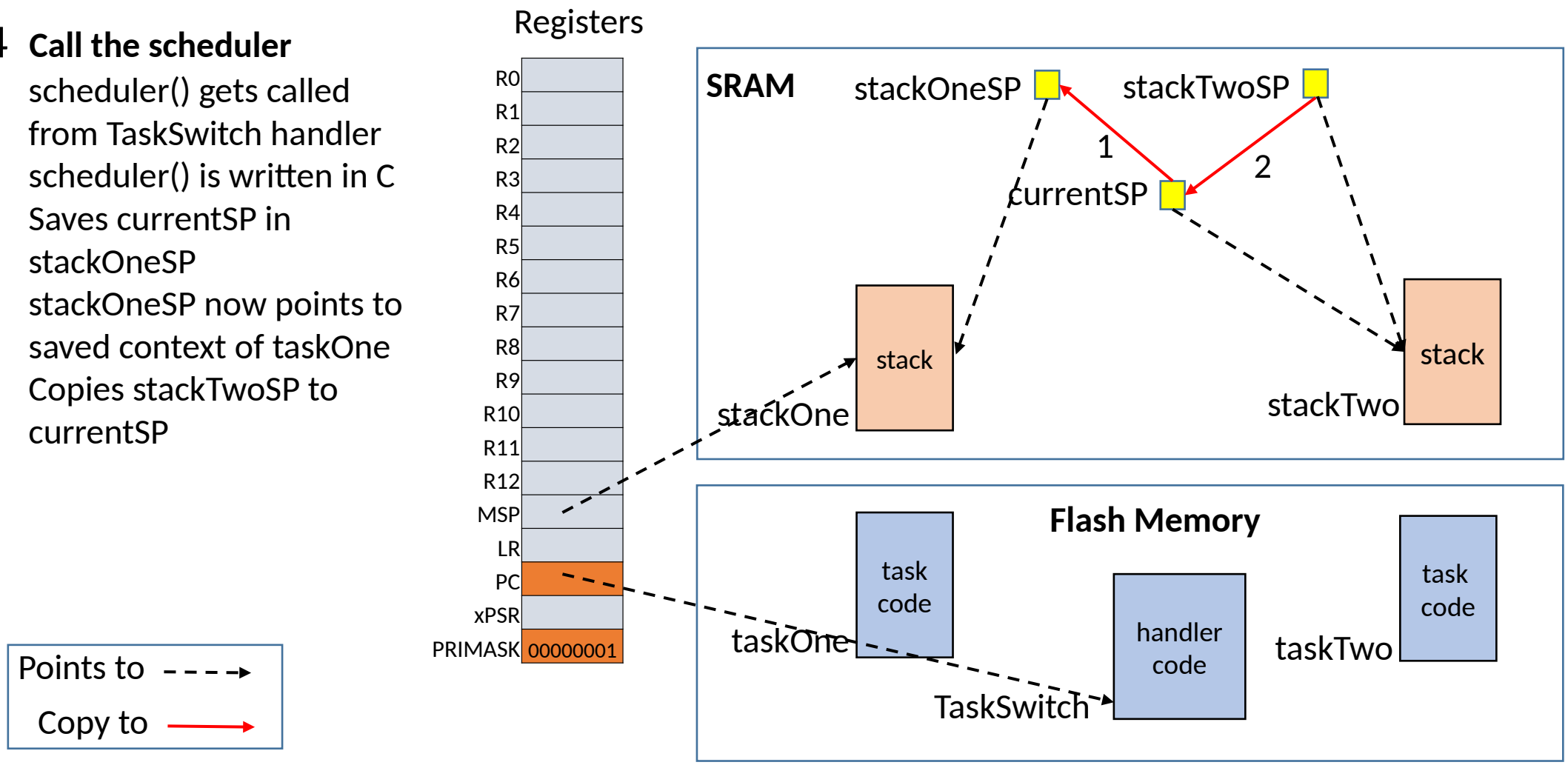
- Next step is to call scheduler to select next task



Context switch trace (Cortex-M4)

4 **Call the scheduler**

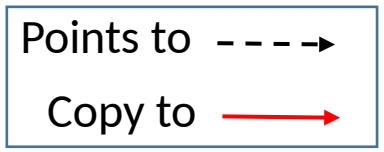
- scheduler() gets called from TaskSwitch handler
- scheduler() is written in C
- Saves currentSP in stackOneSP
- stackOneSP now points to saved context of taskOne
- Copies stackTwoSP to currentSP

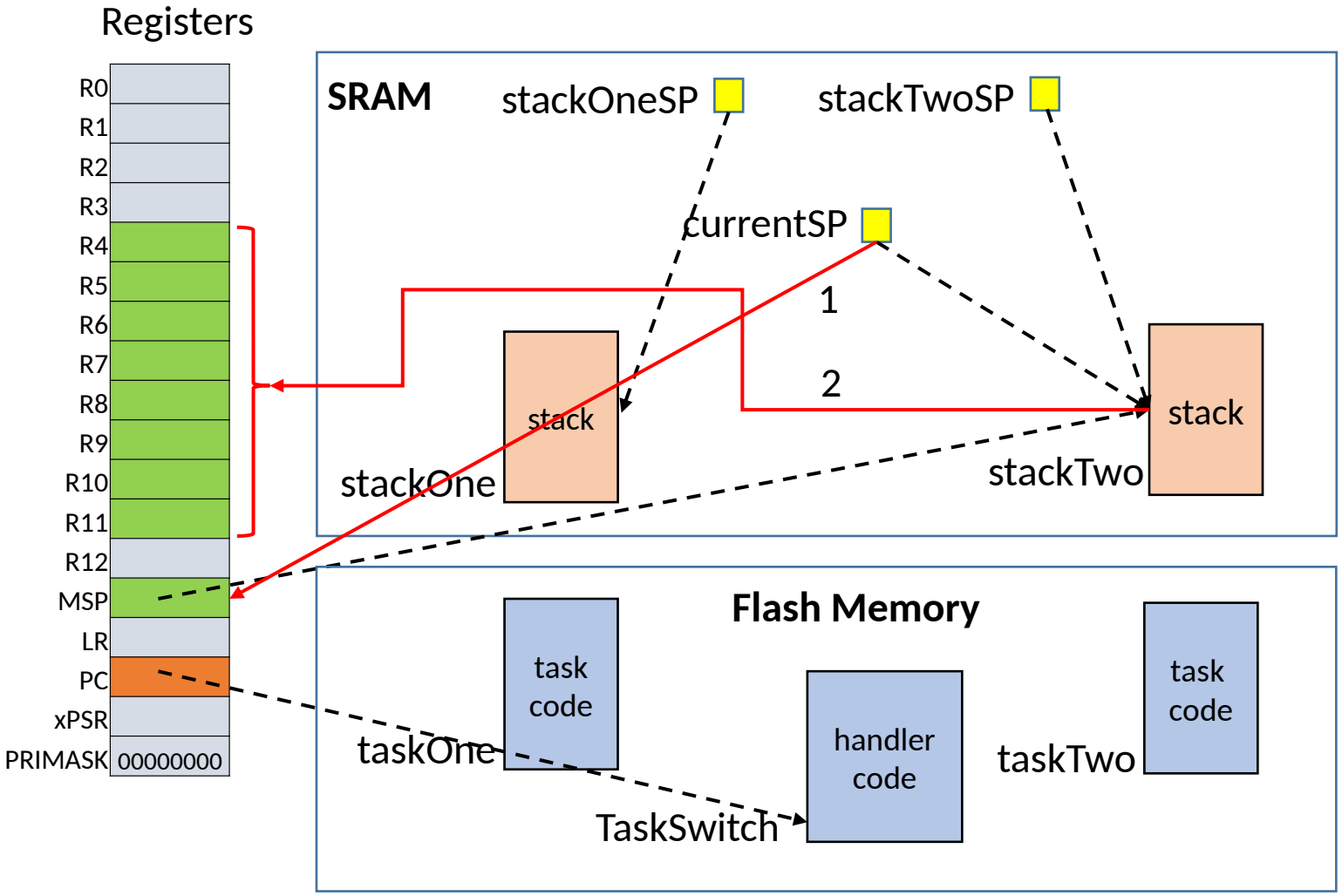


Context switch trace (Cortex-M4)

5 **Resume current task (taskTwo)**

- Copy currentSP to MSP
- MSP now points to saved context of taskTwo
- Restore R4-R11 using MSP
- Enable interrupts by setting PRIMASK=0
- Execute a return from interrupt which will restore R0-R3, R12, LR, PC, xPSR from stackTwo

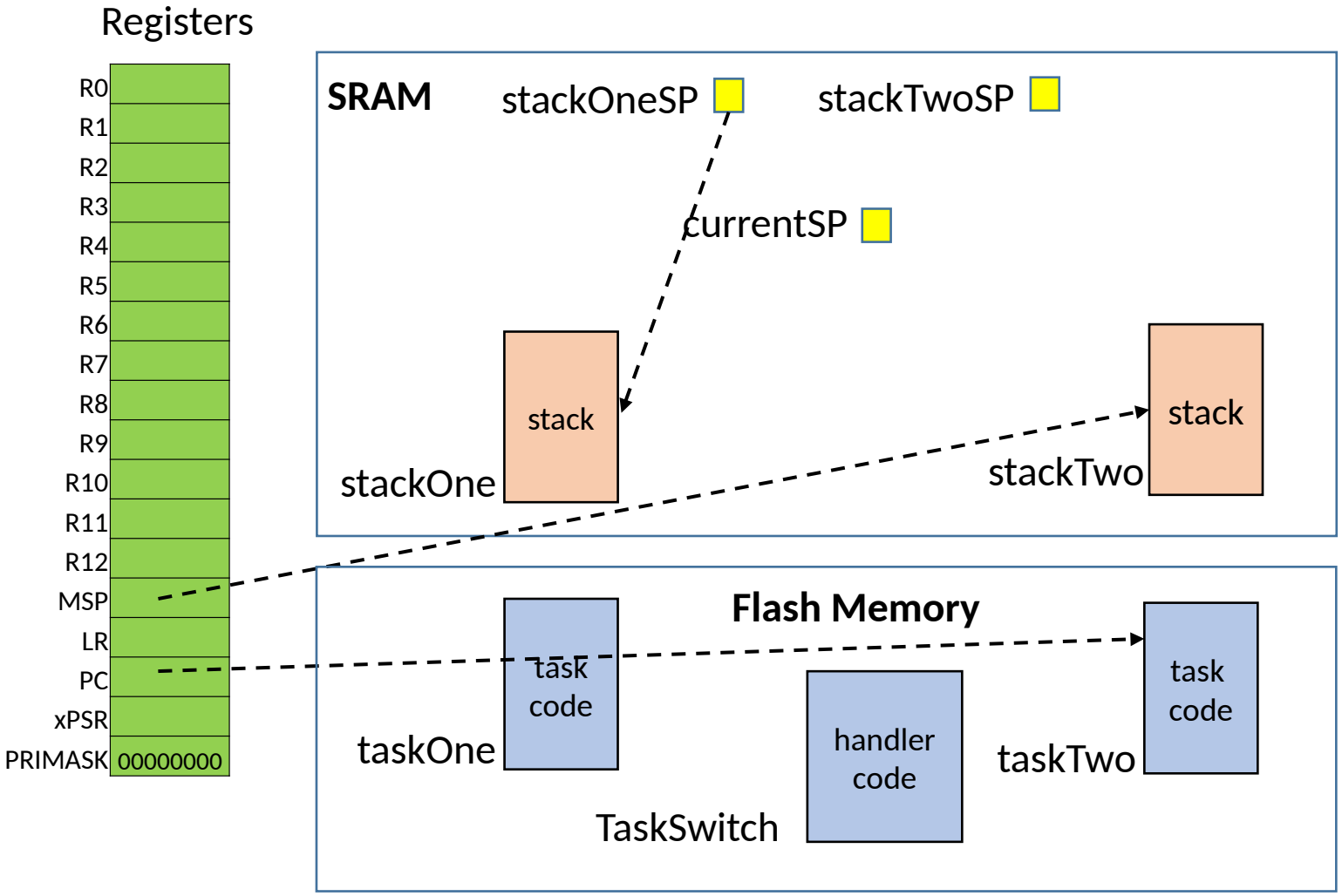




Context switch trace (Cortex-M4)

6 taskTwo is current and running

- In Thread mode
- MSP points somewhere in stackTwo
- PC points somewhere in taskTwo code
- FAULTMASK is 0 (interrupts enabled)
- stackOneSP points to saved context of taskOne



scheduler

```
void scheduler(uint32_t sp)
{
    if (taskID == 1) {
    stackOneSP = sp;                // Store stack pointer for task 1
    taskID = 2;                     // Set the taskID to task 2
    currentSP = stackTwoSP;         // Set the current stack pointer
                                                // to task 2's stack pointer

    } else  {
    stackTwoSP = sp;                // Store stack pointer for task 2
    taskID = 1;                     // Set the taskID to task 1
    currentSP = stackOneSP;         // Set the current stack pointer
                                                // to task 1's stack pointer

    }
}
```

Initializing our multitasking system

- Ensure that PendSV priority is set to lowest
- Need to set up the initial context for a task on its stack
  - xPSR initial value is 0x01000000 i.e. bit 24 indicates Thumb mode.
  - PC is next and gets set to the entry point of the task code
  - LR doesn’t matter because the task should never return
  - All the other registers can have any values we choose
- Set taskID to 1
- Start multitasking by explicitly calling taskOne()
- taskOne will execute until timer tick triggers PendSV which invokes ContextSwitch.
- ContextSwitch will switch control to the initial context of taskTwo

Stack initialization

```
*(--stkptr) = 0x01000000uL;  // xPSR
*(--stkptr) =
    (uint32_t)task_address;  // Entry Point
                             // of task code
*(--stkptr) = 0x14141414uL;  // R14 (LR)
*(--stkptr) = 0x12121212uL;  // R12
*(--stkptr) = 0x03030303uL;  // R3
*(--stkptr) = 0x02020202uL;  // R2
*(--stkptr) = 0x01010101uL;  // R1
*(--stkptr) = 0x00000000uL;  // R0
```

```
*(--stkptr) = 0x11111111uL;      //
R11
*(--stkptr) = 0x10101010uL;      //
R10
*(--stkptr) = 0x09090909uL;      // R9
*(--stkptr) = 0x08080808uL;      // R8
*(--stkptr) = 0x07070707uL;      // R7
*(--stkptr) = 0x06060606uL;      // R6
*(--stkptr) = 0x05050505uL;      // R5
*(--stkptr) = 0x04040404uL;      // R4
```

Explore Assignment 2 in the debugger

- Download ContextSwitch.zip, unzip, load the workspace in EWARM
- Things to notice in **main.c**:
  - stackOne, stackTwo, taskOne(), taskTwo(), scheduler()
  - Multitasking kickoff steps in main()
- Things to notice in **stack.c**:
  - Only the xPSR and PC values are important. The rest of the registers values don’t really matter
- Things to notice in **hw_init.c**:
  - Configure SysTick to interrupt every 200 msec
  - Configure PendSV interrupt priority to lowest
- Things to notice in **startup.s**:
  - PendSV handler set to TaskSwitch
- Things to notice in **interrupt.c**:
  - Trigger PendSV exception in SysTick_Handler()
- Things to notice in **switch.s**:
  - This is where you need to add code to perform the context switch

Explore Assignment 2 in the debugger

- Startup TeraTerm
- Build, upload and run the project
- Break into the program and notice that the TeraTerm output consists of output from the loop in taskOne()
- This is expected since TaskSwitch has not been implemented
- Resume running the program
- Set breakpoints and observe register values:
  - Breakpoint in taskOne()
  - Breakpoint in TaskSwitch

Explore Assignment 2 in the debugger

- We'll then look at the output with the context switch implemented
- Notice we now get output from both taskOne and taskTwo
- Notice some loss of data due to shared buffer not protected by critical section
- Breakpoints in
  - taskOne()
  - taskTwo()