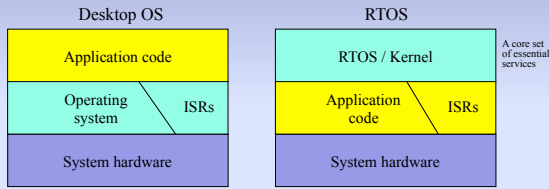


System organization with RTOS



Key: who writes software closest to hardware?



425 F16 4.1

“RTOS” vs. “kernel”

- For some, RTOS = kernel = real-time kernel (RTK)
- For others, RTOS > kernel
 - RTOS includes network support, debugging tools, memory management
 - Kernel includes only most basic services
- In our book and this class:
 - RTOS and kernel are synonymous



425 F16 4.2

RTOS application code: example

```
void main(void)
{
    /* Initialize (but don't start) the RTOS */
    InitRTOS();

    /* Tell the RTOS about our tasks */
    StartTask(RespondToButton, HIGH_PRIORITY);
    StartTask(CalculateTankLevels, LOW_PRIORITY);

    /* Actually start the RTOS. (This function never returns.) */
    StartRTOS();
}
```

- An RTOS is a set of functions called from application code.



425 F16 4.3

Example: Kernel services in μ C/OS

OSInit()	OSSemCreate()
OSIntEnter()	OSSemPend()
OSIntExit()	OSSemPost()
OSMboxCreate()	OSStart()
OSMboxPend()	OSTaskChangePrio()
OSMboxPost()	OSTaskCreate()
OSQCreate()	OSTaskDel()
OSQPend()	OSTimeDly()
OSQPost()	OSTimeGet()
OSSchedLock()	OSTimeSet()
OSSchedUnlock()	OSTimeTick()
OS_ENTER_CRITICAL()	OS_EXIT_CRITICAL()



425 F16 4.4

Comparison

RTOS

- Application code and RTOS are compiled and linked together.
- The application runs first and calls the RTOS.
- RTOS runs only when called by application.
- Dedicated to a single embedded application.
- Few if any extras.
- Needs to run forever without crashing.

Windows/Unix

- Application code and OS are separate.
- The OS runs first and it makes the application run.
- OS runs regularly as separate entity in separate mode.
- Multiple processes, general purpose.
- File system, I/O systems, user interface.
- Occasionally system hangs.



425 F16 4.5

Commercial RTOS Choices

Debugged, with nice tools

VxWorks	VRTX	pSOS
Nucleus	C Executive	LynxOS
QNX	MultiTask!	AMX



425 F16 4.6

So... why write an RTOS?

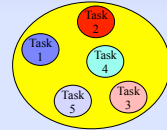
- For insight and experience
 - Understand most basic part of operating systems
 - Good design, coding, and debugging experience
 - Applicable to parallel programming: threads, multi-cores
- Great preparation for career in computing



425 F16 4:7

RTOS essentials: tasks

- The main building blocks of application software written for an RTOS.
- Tasks run independently from each other; execution order determined by RTOS.
- Each task has its own private **context**:
 - register values
 - program counter
 - stack
- All other data is **shared** by all tasks.
- Tasks are more like *threads* than *processes*.



425 F16 4:8

Multitasking

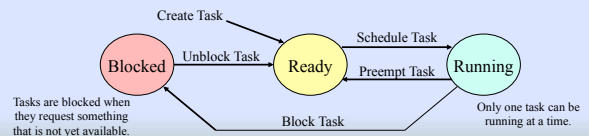
- Many applications best written with multiple tasks.
- Which of these architectures support **multitasking**?
 - Round-robin
 - Round-robin with interrupts
 - Function-queue scheduling
 - RTOS
- Hint:
 - Multitasking system needs a **scheduler** to pick the next task to run.



425 F16 4:9

Task management

- Each task has an associated **state** and a **priority**
- A **blocked** task is waiting for something before it runs again
- The **scheduler** always selects the highest priority ready task



Task States and Transitions



425 F16 4:10

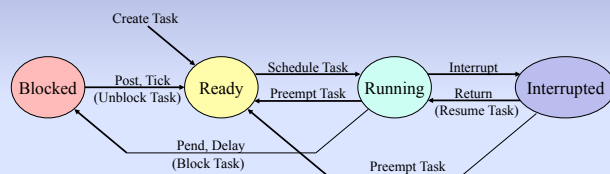
Preemption

- A **preemptive RTOS** will stop the execution of a lower priority task as soon as a higher-priority task becomes ready to run.
 - Example: the highest priority task may unblock when some ISR posts a message or releases a semaphore that task is waiting for.
- A **non-preemptive RTOS** will stop a task only when that task blocks or delays itself (e.g., by calling *pend* or *delay*).
- The kernel we implement this semester is preemptive.



425 F16 4:11

Task states, transitions revisited



What is required for kernel to manage this?



425 F16 4:12

Internal task representation

- OS data structure for each task: **Task Control Block (TCB)**
 - priority
 - state
 - address of next instruction to execute in task code
 - address of current top of task stack
 - other context and status
- Kernel code initializes, maintains, and uses TCB contents.
 - Never accessed directly by task code.
- Important terminology:
 - **Ready list**: TCBs of all tasks in ready state
 - **Suspended list**: TCBs of all tasks in blocked state



425 F16 4:13

YAK

- The name of the RTOS kernel that you develop in labs 4-7.
 - Origin of name lost to antiquity
 - Yet Another Kernel? Y Academic Kernel? YAK Alternative Kernel?
- YAK specification defines the application code interface.
 - The set of functions that tasks, ISRs can call.
 - Function descriptions, details are on the class web pages.
 - Read the details carefully – and repeatedly!



425 F16 4:14

Sample YAK kernel functions

YKNewTask	Creates a new task
YKDelayTask	Task delays itself for a fixed time
YKInitialize	Initializes kernel data structures
YKRun	Starts the application: first task runs
YKEnterMutex	Disables interrupts
YKExitMutex	Enables interrupts
YKScheduler	Picks highest priority task to run



425 F16 4:15

Overview: Labs 4-7

- You are given application code, operational specs.
- You implement the required YAK functions.
 - Interface and functionality are specified
 - Lots of design, implementation options – with consequences!
- Many issues to consider; you should proceed carefully
 - **Must work in teams of two** (exceptions must be approved in advance)
- We'll discuss many details, challenges and options
 - Important to really understand issues before making design decisions



425 F16 4:16

Lab 4

- **Design** your YAK kernel, **implement** a core subset
 - Enough to run application code provided
 - Divided into four pieces, each requiring new functionality
- Some in C, some in assembly; total size not overwhelming
 - My code size for Lab 4, including comments, white space, etc.:
 - 354 lines of C code
 - 175 lines of assembly
- Modify your ISRs and interrupt handlers from Lab 3
 - Required changes are minor, usually adding a few function calls
- Familiarize yourself with debugging capabilities of simulator
 - Goal: when it doesn't work, discover *why* quickly



425 F16 4:17

Lab 4a: design

- After
 - carefully reading all the information available,
 - studying the application code (for parts b-d), and
 - thinking about all issues involved;
 - You submit (via email)
 - your pseudo-code for all required functions, and
 - answers to 21 questions about how you will implement your kernel.
- Examples:
- Where and how will contexts be **saved**?
 - When and how will contexts be **restored**?
 - How will **scheduler** and **dispatcher** really work?
 - What will your **TCB** and associated **data structures** look like?
 - How will you handle a variety of **special cases**?



425 F16 4:18

Purpose of Lab 4a

- Make you think through the implementation issues before you code.
- Required detail goes beyond lectures, slides, book.
- Your submission should demonstrate that you have carefully considered the tricky implementation issues in your kernel.
- Observation: **TA can't catch all potential problems in your submission.**
 - Ternary feedback: (1) on-target, (2) a few loose ends, (3) major problems.
 - You benefit doing a careful design: effort here saves you time later on.

8 hours of programming can save you 10 minutes with pencil and paper.
Mike Goodrich



425 F16 4.19

Lab 4b application

- Source code on next slides has 3 tasks
 - main() creates Task A
 - Task A creates Task B (lower priority) and Task C (higher priority)
 - Once Task C is created, only Task C should run
- Easy to see from output when it works, when it doesn't.
- It must run correctly with your YAK code, without crashing from keypress and timer interrupts, etc.
- Application code will help you understand what your kernel must do.



425 F16 4.20

```
#include "clib.h"

#define ASTACKSIZE 256 /* Size of stack */
#define BSTACKSIZE 256
#define CSTACKSIZE 256

int AStk[ASTACKSIZE]; /* Stack space */
int BStk[BSTACKSIZE];
int CStk[CSTACKSIZE];

void ATask(void); /* Function prototypes */
void BTask(void);
void CTask(void);

void YKInitialize(void);
void YKRun(void);
void YKNewTask(void (* task)(void), void *taskstack, unsigned char priority);
void YKEnterMutex(void);
void YKExitMutex(void);

extern unsigned YKCtxSwCount;

void main(void)
{
    YKInitialize();
    printString("Creating task A...\n");
    YKNewTask(ATask, (void *) &AStk[ASTACKSIZE], 5);
    printString("Starting kernel...\n");
    YKRun();
}
```

Lab 4b application code (must be run without modification)



425 F16 4.21

Lab 4b application code

```
void ATask(void)
{
    printString("Task A started!\n");
    printString("Creating low priority task B...\n");
    YKNewTask(BTask, (void *)
        &BStk[BSTACKSIZE], 7);
    printString("Creating task C...\n");
    YKNewTask(CTask, (void *)
        &CStk[CSTACKSIZE], 2);

    printString("Task A is still running! Oh no! Task
        A was supposed to stop.\n");
    exit(0);
}

void BTask(void)
{
    printString("Task B started! Oh no! Task B
        wasn't supposed to run.\n");
    exit(0);
}
```

```
void CTask(void)
{
    int count;
    unsigned numCtxSwitches;
    YKEnterMutex();
    numCtxSwitches = YKCtxSwCount;
    YKExitMutex();

    printString("Task C started after ");
    printUInt(numCtxSwitches);
    printString(" context switches!\n");

    while (1) {
        printString("Executing in task C.\n");
        for (count = 0; count < 5000; count++);
    }
}
```



425 F16 4.22

```
Creating task A...
Starting kernel...
Task A started!
Creating low priority task B...
Creating task C...
Task C started after 2 context switches! Executing
in task C.

TICK 1
Executing in task C.

TICK 2

TICK 3
Executing in task C.

TICK 4
Executing in task C.
.
.
.
```

Lab 4b output



425 F16 4.23

Lab 4c: new features

- The only task defined in the application code delays itself
 - What runs while task is delayed?
- In YAK, a low priority background task is always ready to run.
 - YAK's **idle task**: created by kernel
 - Once initialized, ready list is never empty; simplifies scheduler, list handling.
 - Idle task never blocks; just increments YKIdleCount in a loop.



425 F16 4.24

Lab 4c application code

```
#include "clib.h"
#define STACKSIZE 256
int TaskStack[STACKSIZE]; /* Space for stack */
void Task(void); /* Function prototypes */

void YKInitialize(void);
void YKRun(void);
void YKNewTask(void (* task)(void), void *taskStack,
                unsigned char priority);
void YKDelayTask(int count);
void YKEnterMutex(void);
void YKExitMutex(void);

extern unsigned YKIdleCount;
extern unsigned YKCtxSwCount;

void main(void)
{
    YKInitialize();
    printf("Creating task...\n");
    YKNewTask(Task, (void *) &TaskStack[STACKSIZE], 0);
    printf("Starting kernel...\n");
    YKRun();
}

void Task(void)
{
    unsigned idleCount;
    unsigned numCtxSwitches;

    printf("Task started.\n");
    while (1)
    {
        printf("Delaying task...\n");
        YKDelayTask(2);

        YKEnterMutex();
        numCtxSwitches = YKCtxSwCount;
        idleCount = YKIdleCount;
        YKIdleCount = 0;
        YKExitMutex();

        printf("Task running after ");
        printf("%u", numCtxSwitches);
        printf(" context switches! YKIdleCount is %u",
              idleCount);
        printf("\n");
    }
}
```

```
Creating task...
Starting kernel...
Task started.
Delaying task...

TICK 1

TICK 2
Task running after 3 context switches! YKIdleCount is 2338.
Delaying task...

TICK 3

TICK 4
Task running after 5 context switches! YKIdleCount is 2328.
Delaying task...

TICK 5

TICK 6
Task running after 7 context switches! YKIdleCount is 2328.
Delaying task...
```

Lab 4c output

Lab 4d: new features

- main() defines four tasks
 - Each delays itself by a different amount
- Results in lots of context switches
 - More extensive testing of context save and restore mechanisms

Lab 4d application code

```
#include "clib.h"
#define ASTACKSIZE 256
#define BSTACKSIZE 256
#define CSTACKSIZE 256
#define DSTACKSIZE 256

int ASH[ASTACKSIZE]; /* Space for each stack */
int BSH[BSTACKSIZE];
int CSH[CSTACKSIZE];
int DSH[DSTACKSIZE];

void ATask(void); /* Function prototypes */
void BTask(void);
void CTask(void);
void DTask(void);

void YKInitialize(void);
void YKRun(void);
void YKNewTask(void (* task)(void), void *taskStack,
                unsigned char priority);
void YKDelayTask(int count);

void main(void)
{
    YKInitialize();
    printf("Creating tasks...\n");
    YKNewTask(ATask, (void *) &ASH[ASTACKSIZE], 3);
    YKNewTask(BTask, (void *) &BSH[BSTACKSIZE], 5);
    YKNewTask(CTask, (void *) &CSH[CSTACKSIZE], 7);
    YKNewTask(DTask, (void *) &DSH[DSTACKSIZE], 9);
    printf("Starting kernel...\n");
    YKRun();
}

void ATask(void) {
    printf("Task A started.\n");
    while (1) {
        printf("Task A, delaying 2.\n");
        YKDelayTask(2);
    }
}

void BTask(void) {
    printf("Task B started.\n");
    while (1) {
        printf("Task B, delaying 3.\n");
        YKDelayTask(3);
    }
}

void CTask(void) {
    printf("Task C started.\n");
    while (1) {
        printf("Task C, delaying 5.\n");
        YKDelayTask(5);
    }
}

void DTask(void) {
    printf("Task D started.\n");
    while (1) {
        printf("Task D, delaying 10.\n");
        YKDelayTask(10);
    }
}
```

Lab 4d output

```
Creating tasks...
Starting kernel...
Task A started.
Task A, delaying 2.
Task B started.
Task B, delaying 3.
Task C started.
Task C, delaying 5.
Task D started.
Task D, delaying 10.

TICK 1

TICK 2
Task A, delaying 2.

TICK 3
Task B, delaying 3.

TICK 4
Task A, delaying 2.

TICK 5
Task C, delaying 5.

TICK 6
Task A, delaying 2.

TICK 7

TICK 8
Task A, delaying 2.

TICK 9
Task B, delaying 3.

TICK 10
Task A, delaying 2.
Task C, delaying 5.
Task D, delaying 10.

TICK 11
.
```

Lab 4 kernel components

Kernel functions:

- YKInitialize** Initializes global variables, kernel data structures
- YKRun** Starts actual execution of user code (tasks)
- YKEnterMutex** Disables interrupts
- YKExitMutex** Enables interrupts
- YKEnterISR** Called on entry to ISR
- YKExitISR** Called on exit from ISR
- YKScheduler** Determines the highest priority ready task
- YKDispatcher** Causes the designated task to execute
- YKNewTask** Creates a new task
- YKDelayTask** Delays a task for specified number of clock ticks
- YKTickHandler** The kernel's timer tick interrupt handler
- YKIdleTask** Lowest priority task, never blocks

Kernel variables:

- YKCtxSwCount** Number of context switches
- YKIdleCount** Incremented in idle task

RTOS essentials: scheduler

- The **scheduler** is the kernel routine that decides what task to run next
- Each task has:
 - A unique priority
 - A state (e.g., running, ready, blocked, interrupted, ...)
- The scheduler always selects the highest priority ready task
- Once the next task is selected, the scheduler calls the dispatcher to make the task run



425 F16 4.31

A simple scheduler

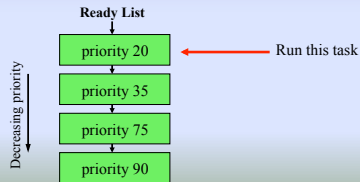
- Schedulers in an RTOS are simple-minded.
 - In YAK, number of ready tasks always ≥ 1 .
 - Not hard to find the one with highest priority.
- Unlike schedulers in Windows/Linux/Unix, the RTOS scheduler makes *no* attempt to be fair.
 - Low priority tasks can be starved; CPU can be hogged.
 - Responsibility of application designer (not OS!) to make sure all tasks get the CPU time they need.



425 F16 4.32

An efficient approach

- If the TCBs of ready tasks are kept in priority order in a queue, the scheduler's job is trivial:
 - Always pick the task at the front of the queue.



425 F16 4.33

The dispatcher

- The scheduler's work is easy; it calls the dispatcher to do the hard part:
 - Actually cause the selected task to run
 - Possibly save context of previously running task
- Tricky because it must handle all of the low-level details:
 - Saving and restoring context, including IP and SP
 - Stack frame, TCB manipulation
- Must be written in assembly
 - You can't save/restore registers or manipulate stack frames in C



425 F16 4.34

Task dispatch

- How does dispatcher actually cause task to run?
- What 8086 instruction(s) could be used to transfer control?
 - Instructions that modify the instruction pointer in 8086:
 - `call`
 - `ret`
 - `int`
 - `iret`
 - `jmp`
 - `jxx` (conditional jumps)
 - Which is best candidate? Does interrupt status matter?



425 F16 4.35

Task dispatch, cont.

- Interrupt status is **crucial**.
 - Interrupts almost certainly off in scheduler & dispatcher
 - Critical section: bad places to get an interrupt, do possible context switch
 - Need to be turned back on *simultaneously* with transfer of control to task
 - Think through what can go wrong if they are *not* coincident
- Dispatcher is tricky, but not lengthy:
 - My function is just 19 instructions long
 - It conditionally calls a subroutine to save context (21 instructions long, also called by ISRs)



425 F16 4.36

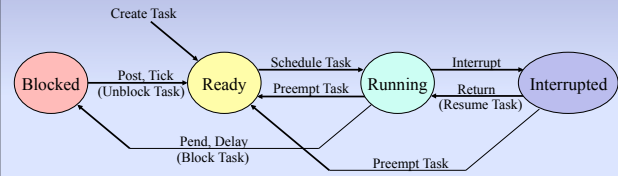
Calling the scheduler

- **Key idea: must be called in *all* kernel code which could change the state of *any* task – before returning to task code.**
 - This ensures that highest priority ready task will always be the next task to run; this provides preemption
- The YAK scheduler must be called:
 1. In `YKRun` to get the first task running
 2. At end of every kernel function that a task can call to become blocked, including `YKDelayTask`, `YKSemPend`, `YKQPend`, etc.
 3. At end of every function that a task can call to cause another task to unblock, including `YKNewTask`, `YKSemPost`, `YKQPost`, etc.
 4. In `YKExitISR`, called near end of each ISR. Handler may have unblocked a task by calling `YKSemPost`, `YKQPost`, `YKTickHandler`, etc.



425 F16 4.37

Task states revisited



425 F16 4.38

Scheduler questions

- How does the scheduler know when the running task blocks?
- How does the scheduler know when a task is unblocked?
- What happens if all tasks are blocked?
- What if two tasks with the same priority are ready to run?
- If one task is running, and another higher-priority task is unblocked, does the running task get stopped right away?

(From text, pp. 141-142)



425 F16 4.39

RTOS essentials: Possible TCB entries

- Task name or ID
- Task priority
- Stack pointer (top of stack) for this task
- Program counter (address of next task instruction to run)
- Task state (running, delayed, suspended, etc.)
- Space to store task's context
- Pointers to link TCBs in lists
- Delay count



425 F16 4.40

Context

- Each task has its own private context
 - Register values
 - Stack (including all stack-based variables)
 - Program counter
- Context must be saved whenever a task stops running, and it must be restored by dispatcher when it runs again.
- The tricky part: **context must be saved and restored consistently regardless of what caused the task to be suspended.**
- What events can cause task to stop running?
 - Something the task did.
 - Something done by something else in the system. Task? ISR?
- How/where might you save context?
 - Could be in TCB or on stack
 - Observation: the code to do this must be written in assembly



425 F16 4.41

Saving private context

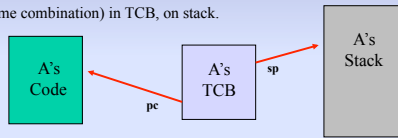
- **When** (in call sequence) do you save a task's context?
 - If suspended by ISR?
 - If suspended by task's own action?
- How do you obtain a **return address** and where do you save it?
- What **stack pointer value** do you save in the TCB?
- **You have to find an answer to these and many other questions for your design.**
 - Devising a way to save context consistently, effectively is probably the hardest part of lab 4.
- Suggestions:
 - **Think it through carefully!**
 - **Draw pictures!** Track stack frame progress, TCB state.
 - **Document your design!** Put something down in writing.
 - **Convince your skeptical partner that your approach will work!**



425 F16 4.42

Challenges in saving context

- Context is a consistent snapshot of all values associated with a task at some instant in time.
- Consider saved context for task A.
 - Stored (in some combination) in TCB, on stack.



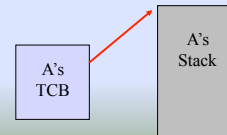
- Must be consistent regardless of why A last stopped running – essential when A runs again.
 - Why is this challenging?
 - Let's consider three cases.



425 F16 4.43

Saving context: case 1

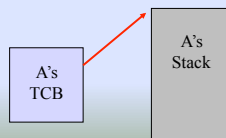
- What happens when a task **runs for the first time**?
 - Is there a context to restore?
- Consider two options:
 - Treat as special case (e.g. use flag in TCB) with special dispatcher
 - Treat same as other cases by storing initial context before task runs the first time (key values: SP, IP, flags)



425 F16 4.44

Saving context: case 2

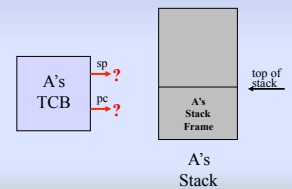
- What happens when a task **is interrupted**?
 - Suppose A is running, tick interrupt makes higher priority task B ready. (Assume B called YKDelayTask() when it last ran.)
 - How will A's context be saved?
 - What did ISR already do?



425 F16 4.45

Case 2: piece by piece

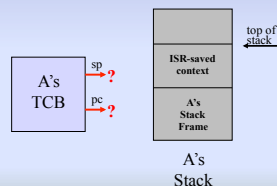
- A is running.



425 F16 4.46

Case 2: piece by piece

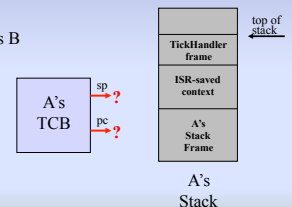
- A is running.
- A is interrupted by tick ISR.



425 F16 4.47

Case 2: piece by piece

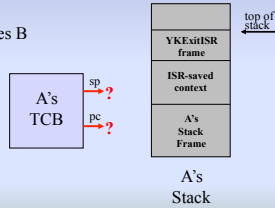
- A is running.
- A is interrupted by tick ISR.
- Tick ISR calls YKTickHandler, makes B "ready".



425 F16 4.48

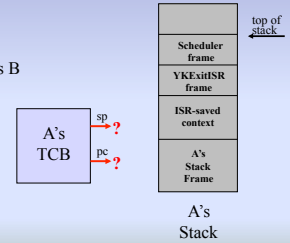
Case 2: piece by piece

- A is running.
- A is interrupted by tick ISR.
- Tick ISR calls YKTickHandler; it makes B “ready”.
- Tick ISR calls YKExitISR.



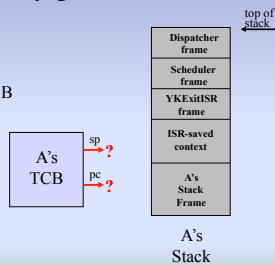
Case 2: piece by piece

- A is running.
- A is interrupted by tick ISR.
- Tick ISR calls YKTickHandler, makes B “ready”.
- Tick ISR calls YKExitISR.
- YKExitISR calls Scheduler.



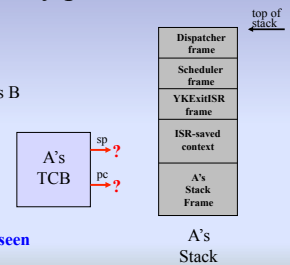
Case 2: piece by piece

- A is running.
- A is interrupted by tick ISR.
- Tick ISR calls YKTickHandler, makes B “ready”.
- Tick ISR calls YKExitISR.
- YKExitISR calls Scheduler.
- Scheduler calls Dispatcher.



Case 2: piece by piece

- A is running.
- A is interrupted by tick ISR.
- Tick ISR calls YKTickHandler, makes B “ready”.
- Tick ISR calls YKExitISR.
- YKExitISR calls Scheduler.
- Scheduler calls Dispatcher.
- Dispatcher causes B to run.
- **What view of A's context will be seen when A runs next?**
 - When do SP and PC get updated?

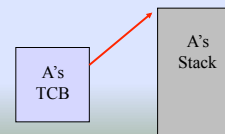


Case 2: discussion

- A complete context was already saved on stack by ISR.
 - Has all register values of interrupted task code.
 - Saved IP is desired “return address”.
- Makes sense to use it – what is required to do this?
 - TCB must be updated with SP value for task.
 - But at what precise point?
 - Should SP in some TCB be updated by every ISR?
 - Are nested interrupts an issue here?
- Consider (likely) case of task resuming after ISR.
 - Better to resume with scheduler/dispatcher or simply return?

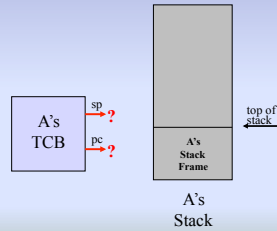
Saving context: case 3

- What happens when a task **delays itself**?
 - Suppose A is running, and it calls YKDelayTask().
 - How will A's context be saved?
 - Similar scenarios: calls to kernel functions that **cause task to block**.



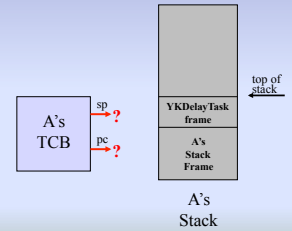
Case 3: piece by piece

- A is running.



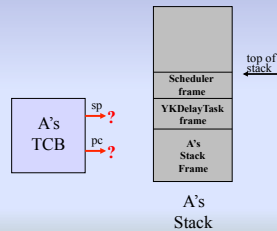
Case 3: piece by piece

- A is running.
- A calls YKDelayTask.



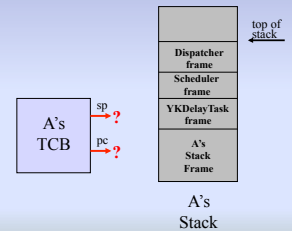
Case 3: piece by piece

- A is running.
- A calls YKDelayTask.
- YKDelayTask calls Scheduler.



Case 3: piece by piece

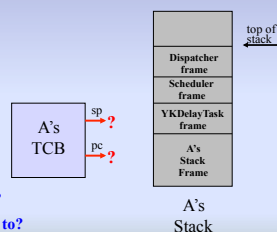
- A is running.
- A calls YKDelayTask.
- YKDelayTask calls Scheduler.
- Scheduler calls Dispatcher.



Case 3: piece by piece

- A is running.
- A calls YKDelayTask.
- YKDelayTask calls Scheduler.
- Scheduler calls Dispatcher.
- Dispatcher causes another task to execute.

- What context for A is saved?
 - When in call sequence is it saved?
 - What will saved SP and PC point to?



Case 3: other possible solutions

- Would any of these alternative approaches work?
 - At beginning of YKDelayTask, call assembly function to save context.
 - At beginning of YKDelayTask, use inline assembly to save context.
 - Write YKDelayTask entirely in assembly; save context at start.
 - Make YKDelayTask an assembly wrapper function that saves context, calls C code.
 - In dispatcher, use back trail of saved bp values: determine sp and pc for return to YKDelayTask (or task), save context with those values.
 - Save context in dispatcher: use return address to scheduler, which will return to YKDelayTask, then task code.
 - In YKDelayTask, change state, then generate software interrupt. That ISR saves context, updates TCB, calls scheduler.

Solutions: discussion

- Things to consider:
 - If I call a function or use inline assembly, are some registers changed?
 - I may obtain return address and stack pointer for some previous point of execution, but how would I get the corresponding register values?
 - Seldom a good idea to use assembly code if it can be avoided.
 - Many kernel functions that *can* cause a task to block will not *always* do so. Examples: depends on state of semaphore, or whether queue is empty.
 - Dangerous to reach into previous stack frames for values.
 - For every frame allocated on stack (regular stack frame created by function, or frame storing context), there must be corresponding code to remove it.
 - If execution resumes in scheduler, what will that code do?



425 F16 4.61

Saving context: overall options

- Option 1: save context in same way in all cases
 - Scheduler calls single dispatcher, fires up task in same way regardless of what caused it to stop execution.
 - Keeps scheduler, dispatcher simple.
- Option 2: treat each case separately
 - Scheduler must detect (using flag in TCB?) and call correct dispatcher.
 - (1) First time running, (2) interrupted, (3) stopped by self
 - Scheduler more complicated, multiple dispatchers required.
 - Will future functionality require additional dispatchers?
- **Extremely important issue – think this through carefully!**
 - **Crucial:** SP, PC, registers must be consistent snapshot in time



425 F16 4.62

Kernel questions

- Some we've addressed:
 - How are tasks represented? What data structures are used?
 - When and where do you store task contexts?
 - In which routines? What information exists at that point?
 - What code removes every stack frame that is added?
 - How does the dispatcher transfer control to a task?
 - How are tasks run for the first time?
 - How does the delay mechanism work?
- Some we've not explicitly addressed:
 - How do you allocate TCBs?
 - How big do stacks need to be?
 - How do you organize your files?



425 F16 4.63

How are TCBs allocated?

- Each call to YKNewTask needs a new TCB
- We don't have dynamic memory allocation (e.g., malloc)
 - So **where will each TCB struct come from?**
- Recommended solution: write your own allocation routines
 - Declare array of TCB structs, allocate them when needed
 - Set size of array with **#define** in **#include** kernel file, edited by user
 - Example: **#define MAXTASKS 6**
 - They are never recycled: no YKDeleteTask function.



425 F16 4.64

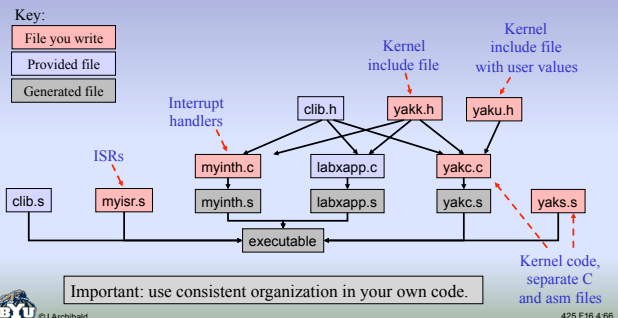
How big must stacks be?

- When are stack frames added?
 - When any function is called
 - When an ISR interrupts task
- What is maximum number of frames that can exist?
 - What is maximum nesting depth of function calls?
 - What is maximum interrupt nesting level?
- What is size of each frame?
 - For functions: local vars, saved registers, arguments to other functions
 - For ISRs: size of context that will be saved
- **Stack size** is responsibility of **application code**, not kernel.



425 F16 4.65

Suggested file organization



425 F16 4.66

Recommendations

- Think things through from multiple perspectives
 - What happens to each *stack frame*?
 - What happens to each *TCB struct*?
 - What happens to each *list*?
 - What happens to the *interrupt flag*?
- Create routines to assist in debugging
 - I wrote a `dumplists()` function, can be called anywhere
 - Sample output: (key: [priority,state,delay])
`YKCurrTask: [2]`
`Rdy: [2,1,0] [4,1,0] [100,1,0]`
`Susp: [3,4,2] [13,4,8]`
 - Easy to confirm state transitions in context switches, etc.
 - Concise output extremely helpful!



425.F16.4.67

Lab 4: implementation questions?

Kernel functions:

YKInitialize	Initializes all required kernel data structures
YKRun	Starts actual execution of user code (tasks)
YKEnterMutex	Disables interrupts
YKExitMutex	Enables interrupts
YKEnterISR	Called on entry to ISR
YKExitISR	Called on exit from ISR
YKScheduler	Determines the highest priority ready task
YKDispatcher	Begins or resumes execution of the next task
YKNewTask	Creates a new task
YKDelayTask	Delays a task for specified number of clock ticks
YKTickHandler	The kernel's timer tick interrupt handler
YKIdleTask	Lowest priority task, never blocks

Kernel variables:

YKCtxSwCount	Number of context switches
YKIdleCount	Incremented in idle task



425.F16.4.68