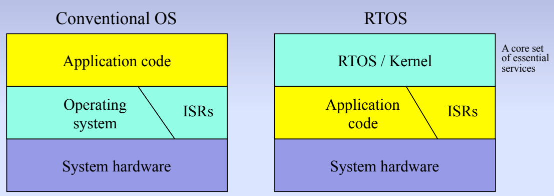
System organization



Key: who writes software closest to hardware?

"RTOS" vs. "kernel"

- For some,
  RTOS = kernel = real-time kernel (RTK)
- For others,
  RTOS is more than a 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, used interchangeably

RTOS application code: example

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

```
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();
}
```


Example: Kernel services in μC/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()**

Comparison

Windows/Unix	RTOS
• Application code and OS are separate entities.	• Application code and RTOS are compiled and linked together.
• The OS runs first and makes the application run.	• The application runs first and calls the RTOS.
• OS runs regularly as separate entity in separate mode.	• RTOS runs only when called by application code.
• Multiple processes, general purpose.	• Dedicated to a single embedded application.
• File system, I/O systems, user interface.	• No file system, doesn't handle I/O, no user interface.
• Occasionally system hangs.	• Should run forever without crashing.

Commercial RTOS Choices

Debugged, with nice tools

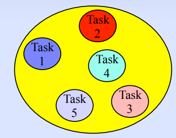
VxWorks	VRTX	pSOS
Nucleus	C Executive	LynxOS
QNX	MultiTask!	AMX

So... why *write* an RTOS?

- To gain insight and experience:
  - Insight into essential operations in computer systems
  - Insight into challenges of parallel programming: threads, multi-cores
  - Experience with design, coding, and debugging
- Bottom line
  - Good preparation for a variety of careers in computing

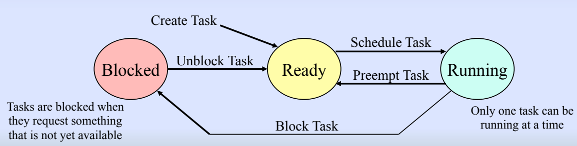
RTOS essentials: tasks

- The main building blocks of application software written for an RTOS
- Tasks run independently; 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*



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

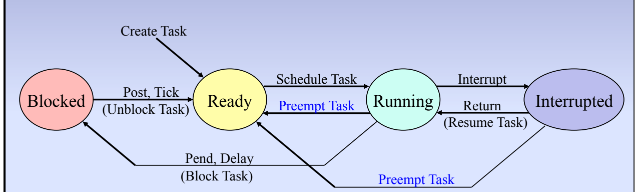


Task States and Transitions

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 it receives a message that it is waiting for
- A non-preemptive RTOS will stop a task only when that task blocks or delays itself (e.g., by calling a *pend* or *delay* RTOS function)
- The kernel we implement this semester is preemptive

Task states, transitions revisited



What is required for kernel to manage this?

Internal task representation

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

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!

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

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
  - From here (lab 4a) on out, must work in teams of two
  - Exceptions must be approved in advance
- In class, we'll discuss many details, challenges and options
  - Important to understand issues thoroughly before writing code

Lab 4

- Design your YAK kernel, implement a core subset
  - Enough to run application code provided
  - Divided into 4 parts, each requiring new functionality
- Some coding 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* as quickly as possible

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 and partner submit (via Learning Suite)
  - 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?

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: TAs cannot catch all potential problems in your submission
  - Ternary feedback: (1) on-target, (2) a few loose ends, (3) major problems
  - You benefit from 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

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

Lab 4b application code

(must be run without modification)

```
#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

```
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++);
   }
}
```


Lab 4b output

```
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 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

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

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

   printString("Task started.\n");
   while (1)
   {
      printString("Delaying task...\n");

      YKDelayTask(2);

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

      printString("Task running after ");
      printUInt(numCtxSwitches);
      printString(" context switches! YKIdleCount is ");
      printUInt(idleCount);
      printString(".\n");
   }
}
```

Lab 4c output

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

TICK 1

TICK 2
Task running after 3 context switches! YKIdleCount is 2330.
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 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 AStk[ASTACKSIZE];       /* Space for each stack  */
int BStk[BSTACKSIZE];
int CStk[CSTACKSIZE];
int DStk[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();
   printString("Creating tasks...\n");
   YKNewTask(ATask, (void *) &AStk[ASTACKSIZE], 3);
   YKNewTask(BTask, (void *) &BStk[BSTACKSIZE], 5);
   YKNewTask(CTask, (void *) &CStk[CSTACKSIZE], 7);
   YKNewTask(DTask, (void *) &DStk[DSTACKSIZE], 8);
   printString("Starting kernel...\n");
   YKRun();
}
```

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

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

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

void DTask(void)  {
   printString("Task D started.\n");
   while (1)   {
      printString("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.
Task B, delaying 3.

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

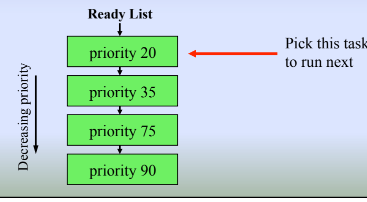
- The scheduler is the kernel routine that decides what task to run next
- Each task has:
  - A unique priority
  - A state (e.g., running, ready, blocked, interrupted, ...)
- The scheduler always selects the highest priority ready task
- Once the next task has been selected, the scheduler calls the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

A simple scheduler

- Schedulers in an RTOS are simple-minded
  - In YAK, the number of ready tasks is always ≥ 1
  - It is 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 RTOS!) to make sure all tasks get the CPU time they need

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



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 registers, 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

Task dispatch

- How does dispatcher actually cause a task to run?
- What 8086 instruction should 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?

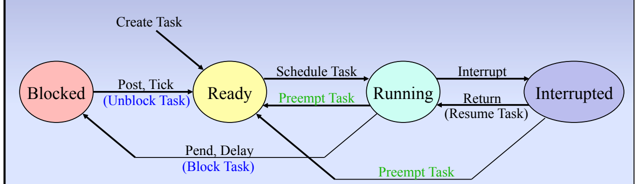
Task dispatch, cont.

- Interrupt status is crucial
  - Interrupts almost certainly *off* in scheduler and dispatcher
    - Critical section: bad place to get an interrupt, do possible context switch
  - Need to be turned back on *simultaneously* with transfer of control to task
    - Thought experiment: what can go wrong if they do not happen at same time?
- Dispatcher is tricky, but not lengthy:
  - My dispatcher consists of just 19 instructions
  - It conditionally calls a subroutine to save context (21 instructions long, also called by ISRs)

Calling the scheduler

- Key idea: scheduler must be called in *all* kernel code which could change the state of *any* task, before returning to task code
  - This provides preemption: the highest priority ready task will always be the next task to run
- In YAK, the scheduler must be called:
  1. In YKRun, to run the first task
  2. At end of every kernel function that can cause a task to block (Examples: YKDelayTask, YKSemPend, YKQPend)
  3. At end of every kernel function that can cause a task to become ready (Examples: YKNewTask, YKSemPost, YKQPost)
  4. In YKExitISR, called near end of each ISR (Handler may have called a kernel function that unblocked a task)

Task states revisited



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)

RTOS essentials: Possible TCB entries

- Task name or ID
- Task priority
- Task state (Running, Ready, Blocked, Delayed, etc.)
- Delay count
- Stack pointer (top of stack) for this task
- Program counter (address of next task instruction to run)
- Space to store register context of task
- Pointers to link TCBs (to form lists)

Context

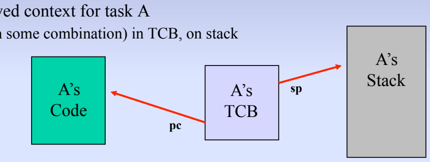
- Each task has its own private context
  - Register values
  - Stack (including all stack-based variables)
  - Program counter
- Register context must be saved whenever a task stops running, and it must be restored by dispatcher when it runs again
- The tricky part: context must be saved and restored *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? Interrupt handler?)
- How/where might you save context?
  - Options: in TCB or on task's stack
  - Observation: the code to save context must be written in assembly

Saving private context

- Hardest part of lab 4: devising a way to save context consistently and reliably
- Critical questions to address in your design:
  - When (in call sequence) will task context be saved?
    1. If suspended by ISR?
    2. If suspended by action of the task?
  - How do you obtain a return address and *where* is that address saved?
  - What stack pointer value do you save in the TCB, and *when* is it saved?
- Suggestions:
  - Think this 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!

Challenges

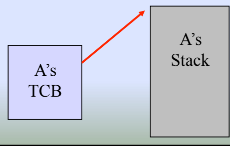
- Context is a snapshot of all values associated with a task at some instant in time
- Consider saved context for task A
  - Stored (in some combination) in TCB, on stack



  - Ideally, state will be consistent regardless of *why* A last stopped running, making it easier to run A again the next time
    - Why is this challenging?
    - Let's consider three cases

Saving context: case 1

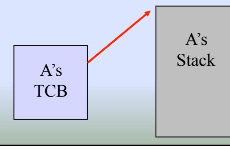
- What happens when a task runs for the first time?
  - Is there a context to restore?
- Consider two options:
  1. Treat as special case (e.g. use flag in TCB) with special dispatcher
  2. Treat same as other cases by storing initial context before task runs the first time (key values: SP, IP, flags)



©J Archibald 425 F19 4:43

Saving context: case 2

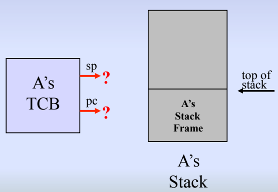
- What happens when a task is interrupted?
  - Suppose A is running, tick interrupt makes higher priority task B ready
    - Scenario: B called YKDelayTask when it last ran
  - How will A's context be saved?
    - Key question to consider: what did ISR already do?



©J Archibald 425 F19 4:44

Case 2: piece by piece

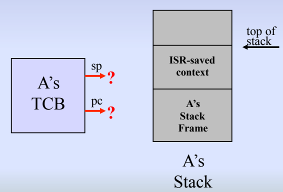
- A is running



©J Archibald 425 F19 4:45

Case 2: piece by piece

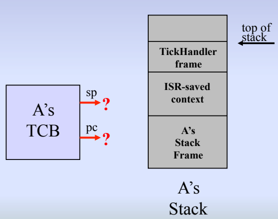
- A is running
- A is interrupted by tick ISR



©J Archibald 425 F19 4:46

Case 2: piece by piece

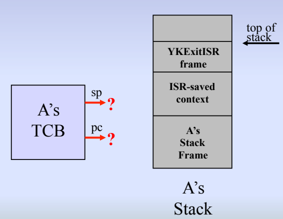
- A is running
- A is interrupted by tick ISR
- Tick ISR calls YKTickHandler, which makes B "ready" and returns



©J Archibald 425 F19 4:47

Case 2: piece by piece

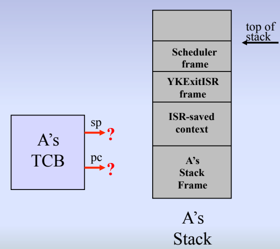
- A is running
- A is interrupted by tick ISR
- Tick ISR calls YKTickHandler, which makes B "ready" and returns
- Tick ISR calls YKExitISR



©J Archibald 425 F19 4:48

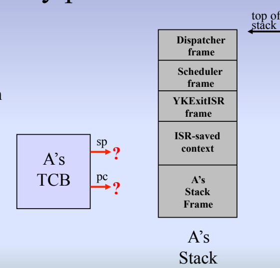
Case 2: piece by piece

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



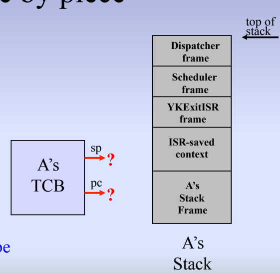
Case 2: piece by piece

- A is running
- A is interrupted by tick ISR
- Tick ISR calls YKTickHandler, which makes B “ready” and returns
- 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, which makes B “ready” and returns
- Tick ISR calls YKExitISR
- YKExitISR calls Scheduler
- Scheduler calls Dispatcher
- Dispatcher causes B to run
- Where is the context of A that will be loaded when A runs next?
  - When are SP and PC updated (in TCB)?

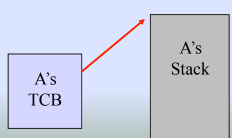


Case 2: discussion

- Observation: 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
    - At what precise point in the execution sequence?
  - Should SP in some TCB be updated by every ISR?
    - What about nested interrupts?
- Consider (likely) case of same task resuming after ISR
  - Better to resume with scheduler/dispatcher or simply return?

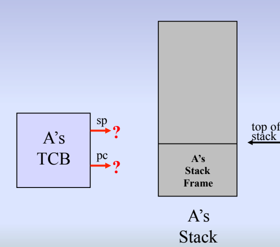
Saving context: case 3

- What happens when a task delays itself?
  - Suppose A is running, and it calls YKDelayTask
  - How will A’s context be saved?
  - Consider: calls to other kernel functions can also cause task to block



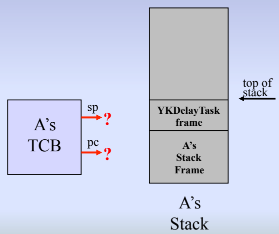
Case 3: piece by piece

- A is running



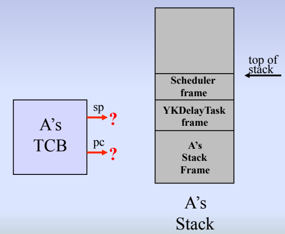
Case 3: piece by piece

- A is running
- A calls YKDelayTask



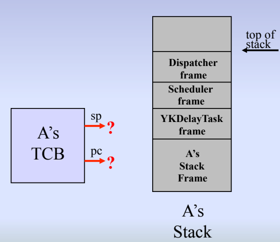
Case 3: piece by piece

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



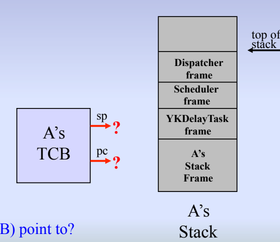
Case 3: piece by piece

- 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 (in TCB) point to?



Case 3: possible solutions

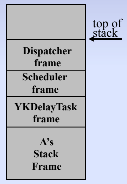
- Would any of these approaches work?
  - At beginning of YKDelayTask, call assembly function to save context
  - At beginning of YKDelayTask, use inline assembly to save context
  - Write YKDelayTask entirely in assembly; save context at start
  - Make YKDelayTask an assembly wrapper function that saves context, calls C code
  - In Dispatcher, use back trail of saved bp values: determine sp and pc for return to YKDelayTask, save those values and corresponding context
  - Save context in Dispatcher: use return address to Scheduler, which will return to YKDelayTask, then task code

Solutions: discussion

- Things to consider:
  - If I call a function or use inline assembly, won't that change some registers?
  - I may obtain return address and stack pointer for some previous point of execution, but how would I get the corresponding register values?
  - Best to avoid assembly code whenever possible
  - Many kernel functions that *can* cause a task to block will not *always* do so
    - Example: task calling YKSemPend may be blocked, or call may return immediately
  - Dangerous to reach into previous stack frames for values
  - For every frame allocated on stack (regular stack frame created by function, or frame that stores context), there must be corresponding code somewhere to remove it
  - If execution resumes in Scheduler (after call to Dispatcher), what will that code do?

Thinking it through

- Scenario:
  - Dispatcher conditionally saves context, uses return address to the Scheduler
  - What happens when task A resumes?



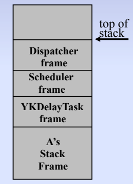

```
void YKDelayTask(int ticks)
{
    if (ticks <= 0) return;
    YKEnterMutex();
    // move TCB to blocked list
    YKScheduler(CTXNOTSAVED);
    YKExitMutex();
}
```

```
void YKScheduler(int i)
{
    if (YKRdyList != YKCurrTask)
    {
        YKCtxSwCount++;
        YKDispatcher(i);
    }
}
```

Sample YAK code

Thinking it through

- How will all register values be saved in this scenario?
  - Suppose critical value is in register in YKDelayTask
  - Register used is either **caller-save** or **callee-save**



Case 1: **caller-save**

```
void YKDelayTask(int ticks)
{
    ...
    // register is saved
    YKScheduler(CTXNOTSAVED);
    // register is restored
    ...
}
```

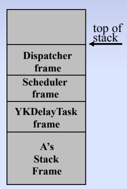
```
void YKScheduler(int i)
{
    ...
    // register is modified
    ...
    YKDispatcher(i);
}
```

Execution starts here when task runs again

Sample YAK code

Thinking it through

- How will all register values be saved in this scenario?
  - Suppose critical value is in register in YKDelayTask
  - Register used is either **caller-save** or **callee-save**



Case 2: **callee-save**

```
void YKDelayTask(int ticks)
{
    ...
    YKScheduler(CTXNOTSAVED);
    ...
}
```

```
void YKScheduler(int i)
{
    ...
    // register is saved
    // register is modified
    YKDispatcher(i);
    // register is restored
}
```

Execution starts here when task runs again

Sample YAK code

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 both Scheduler and Dispatcher fairly simple
- Option 2: treat each case separately
  - Cases: (1) first time running, (2) interrupted, (3) stopped by self
  - Scheduler must detect (using info in TCB) and call correct Dispatcher
  - Scheduler more complicated, multiple versions of Dispatcher required
- Extremely important issue – think this through carefully!
  - **Remember:** SP, PC and registers must be consistent snapshot in time

Kernel questions

- Some we've addressed:
  - How are tasks represented? What data structures are used?
  - When and where are task contexts saved?
    - Where in execution sequence? 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 task delay mechanism work?
- Some we've not yet addressed:
  - How do you allocate TCBs?
  - How big do stacks need to be?
  - How do you organize your files?

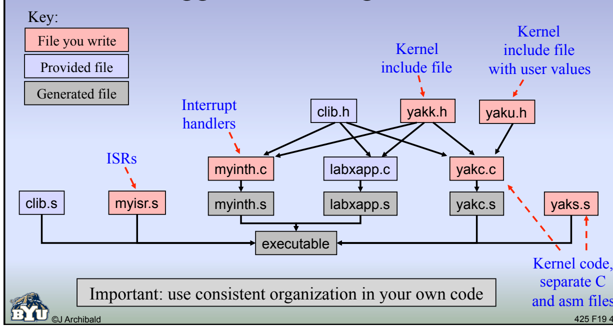
How are TCBs allocated?

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

How big must stacks be?

- When are stack frames added?
  - When any function is called
  - When an ISR runs
- 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
- Good news: stack size is responsibility of application code, not kernel
  - Sole exception: idle task

Suggested file organization



Important: use consistent organization in your own code

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 help you debug your code
  - 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 useful!

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