Sharing the Processor: A Survey of Approaches to Supporting Concurrency

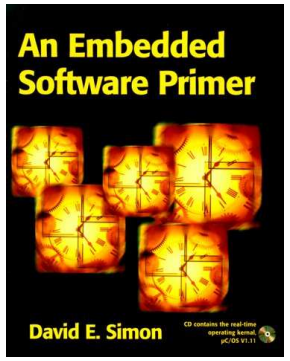
Today

Topic - How do we make the processor do things at the right times?

- For more details see Chapter 5 of D.E. Simon, **An Embedded Software Primer**, Addison-Wesley 1999

There are various methods; the best fit depends on…

- system requirements – response time
- software complexity – number of threads of execution
- resources – RAM, interrupts, energy available



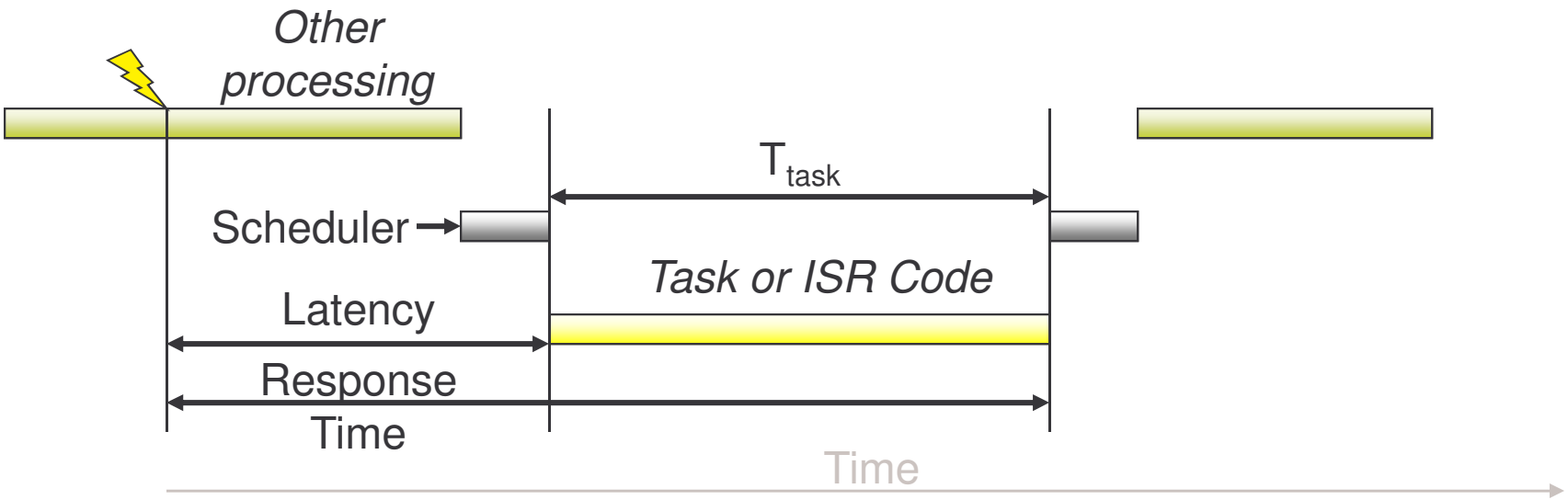
RTOS Cult De-Programming

- How do we schedule the tasks on the CPU?
- An infinite loop in main
- Real-time operating system
- Is there anything else available?

Real-Time Operating System


```
while (1) {
   ...
};
```

Definitions



- $T_{Release}(i)$ = Time at which task i is becomes ready to run
- $T_{response}(i)$ = Delay between request for service and completion of service for task i
- $T_{task}(i)$ = Time needed to perform computations for task i
- $T_{ISR}(i)$ = Time needed to perform interrupt service routine i

Round-Robin/Super-Loop

Extremely simple

- No interrupts
- No shared data problems

Poll each device (`if (device_A_ready())`)

Service it with task code when needed

```
void main(void) {
  while (TRUE) {
    if (device_A_ready()) {
      service_device_A();
    }
    if (device_B_ready()) {
      service_device_B();
    }
    if (device_C_ready()) {
      service_device_C();
    }

  }
}
```

Example Round-Robin Application

```
void DMM_Main(void) {
  enum {OHMS_1, ... VOLTS_100} SwitchPos;
   while (TRUE) {
        switch (SwitchPos) {
        case OHMS_1:
               ConfigureADC(OHMS_1);
               EnableOhmsIndicator();
               x = Convert();
               s = FormatOhms(x);
               break;
        ...
        case VOLTS_100:
               ConfigureADC(VOLTS_100);
               EnableVoltageIndicator();
               x = Convert();
               s = FormatVolts(x);
               break;
        }
        DisplayResult(s);
        Delay(50);
   }
}
```

Sample Application - Network Videophone

- Video
  - 30 frames/s
  - 360 x 240 images
  - Compress/ Decompress with MPEG-2
- Audio
  - 8 kHz sampling
  - Compress with GSM 06.10
- Processor
  - 3000 MIPS
- Tasks have deadlines

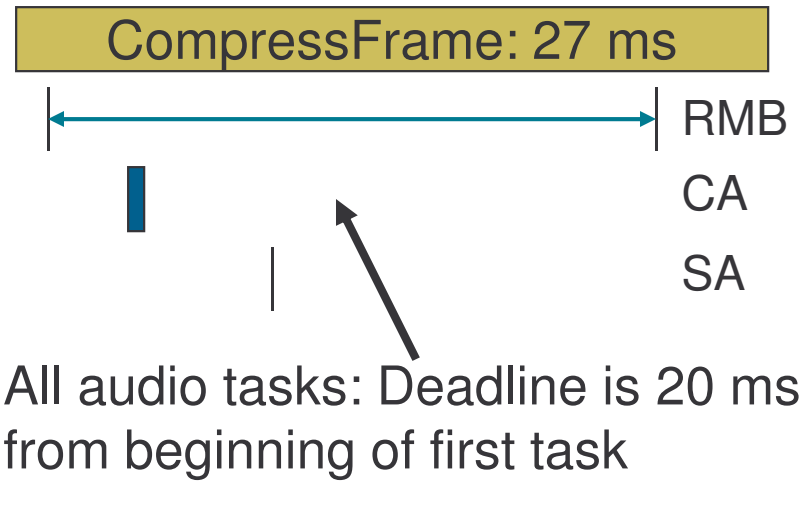
Service	Direction	Function	WCET	Deadline
Video	Send	SampleFrame	1 ms	33.3 ms
		CompressFrame	27 ms	33.3 ms
		SendFrame	0.1 ms	33.3 ms
	Receive	ReceiveFrame	0.1 ms	33.3 ms
		DecompressFrame	2.7 ms	33.3 ms
		DisplayFrame	1 ms	33.3 ms
Audio	Send	ReadMicBuffer	0.001 ms	20 ms
		CompressAudio	0.160 ms	20 ms
		SendAudio	0.001 ms	20 ms
	Receive	ReceiveAudio	0.001 ms	20 ms
		DecompressAudio	0.160 ms	20 ms
		LoadAudioBuffer	0.001 ms	20 ms

Scheduling NV with Round-Robin

Round robin works for either video or audio, but not both

Need to split up video

`CompressFrame()`




```
void main() {
  while(TRUE) {
    if (TimeToSample) {
      SampleFrame();
      CompressFrame();
      SendFrame();
    }
    if (FrameWaiting) {
      ReceiveFrame();
      DecompressFrame();
      DisplayFrame();
    }
  }
}
```

Limitations of Round-Robin

Architecture supports multi-rate systems very poorly

- Voice Recorder: sample microphone at 20 kHz, sample switches at 15 Hz, update display at 4 Hz. How do we do this?

Polling frequency limited by time to execute main loop

- Can get more performance by testing more often (A/Z/B/Z/C/Z/...)
- This makes program more complex and increases response time for other tasks

Potentially Long Response Time

- In worst case, need to wait for all devices to be serviced
- $$\max\left(T_{response}(j)\right) = \sum_{\forall t} T_{task}(t)$$

Fragile Architecture

- Adding a new device will affect timing of all other devices
- Changing rates is tedious and inhumane

Event-Triggered using Interrupts

Very basic architecture, useful for simple low-power devices, very little code or time overhead

Leverages built-in task dispatching of interrupt system

- Can trigger ISRs with input changes, timer expiration, UART data reception, analog input level crossing comparator threshold

Function types

- Main function configures system and then goes to sleep
  - If interrupted, it goes right back to sleep
- Only interrupts are used for normal program operation

Example: bike computer

- Int1: wheel rotation
- Int2: mode key
- Int3: clock
- Output: Liquid Crystal Display

Bike Computer Functions

Reset

```
Configure timer,
inputs and
outputs

cur_time = 0;
rotations = 0;
tenth_miles = 0;

while (1) {
  sleep;
}
```

ISR 1: Wheel rotation

```
rotations++;
if (rotations>
  R_PER_MILE/10) {
  tenth_miles++;
  rotations = 0;
}
speed =
 circumference/
 (cur_time – prev_time);
compute avg_speed;
prev_time = cur_time;
return from interrupt
```

ISR 2: Mode Key

```
mode++;
mode = mode %
   NUM_MODES;
return from interrupt;
```

ISR 3: Time of Day Timer

```
cur_time ++;
lcd_refresh--;
if (lcd_refresh==0) {
 convert tenth_miles
    and display
 convert speed
    and display
  if (mode == 0)
    convert cur_time
      and display
  else
    convert avg_speed
      and display
  lcd_refresh =
     LCD_REF_PERIOD
}
```

Limitations of Event-Triggered using Interrupts

All computing must be triggered by an event of some type

- Periodic events are triggered by a timer

Limited number of timers on MCUs, so may need to introduce a scheduler of some sort which

- determines the next periodic event to execute,
- computes the delay until it needs to run
- initializes a timer to expire at that time
- goes to sleep (or idle loop)

Everything (after initialization) is an ISR

- All code is in ISRs, making them long
- Response time depends on longest ISR. Could be too slow, unless interrupts are re-enabled in ISR
- Priorities are directly tied to MCU's interrupt priority scheme

Round-Robin with Interrupts

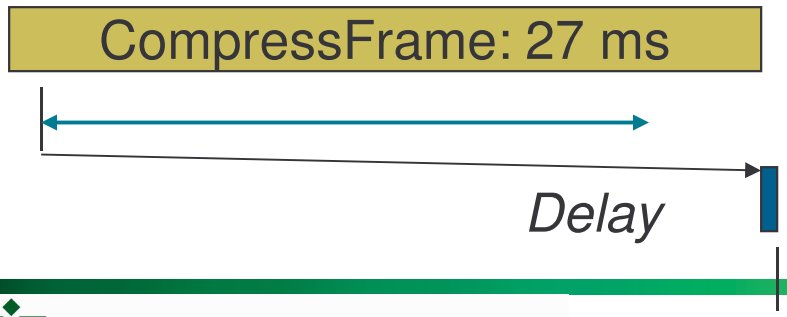
- Also called
  - foreground/background
- Interrupt routines
  - Handle most urgent work
  - Set flags to request processing by main loop
- More than one priority level
  - Interrupts – multiple interrupt priorities possible
  - **`main`** code

```
BOOL DeviceARequest, DeviceBRequest,
DeviceCRequest;
void interrupt HandleDeviceA(){
  /* do A's urgent work */
  ...
  DeviceARequest = TRUE;
}
void main(void) {
  while (TRUE) {
    if (DeviceARequest) {
      FinishDeviceA();
    }
    if (DeviceBRequest) {
      FinishDeviceB();
    }
    if (DeviceCRequest) {
      FinishDeviceC();
    }
  }
}
```

Scheduling NV with Round Robin + Interrupts

```
BOOL ReadMicBuffer_Req = FALSE,
  SampleFrame_Req = FALSE;

interrupt void HandleMicBuffer()
{
  copy contents of mic buffer
  ReadMicBuffer_Done = TRUE;
}
interrupt void
HandleSampleFrame() {
  Sample a frame of video
  SampleFrame_Done = TRUE;
}
```




```
void main(void) {
  while (TRUE) {
    if (ReadMicBuffer_Done) {
       CompressAudio();
       SendAudio();
       ReadMicBuffer_Done=FALSE;
    }
    if (SampleFrame_Done) {
       CompressFrame();
       SendFrame();
       SampleFrame_Done = FALSE;
    }
    etc.
  }
}
```

Limitations of Round-Robin with Interrupts

All task code has same priority

- What if device A must be handled quickly, but **`FinishDeviceC`** (slow) is running?
- $$\max\left(T_{response}(j)\right) = \sum_{\forall t} T_{task}(t) + \sum_{\forall i} T_{ISR}(i)$$
- Difficult to improve A's response time
  - Only by moving more code into ISR

Shared data can be corrupted easily if interrupts occur during critical sections

- Flags (**`DeviceARequest`**, etc.), data buffers
- Must use special program constructs
  - Disable interrupts during critical sections
  - Semaphore, critical region, monitor
- New problems arise – Deadlock, starvation

What is Scheduling?

We have seen a “reactive” system – activities are processed based on interrupts.

When scheduled activities are needed, you can set up timer interrupts, or you can have the operating system **schedule** these periodic tasks (perhaps triggered by interrupts…).

Scheduling is choosing which task to run and then running it

The rules:

- Define certain functions to be **tasks**
- If there is a task ready to run, then run it
- Finish a task before you start another one
- If there is more than one task to start, run the highest priority task first (0 is highest priority)

A Simple Example

- We will keep track of how long until the task will run ("time") and if it is scheduled now ("run")

	Priority		Length		Frequency
Task 1	2		1		20
Task 2	1		2		10
Task 3	3		1		5

Elapsed time	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
Task executed						**T3**					**T2**		**T3**			**T3**					**T2**		**T1**	**T3**		**T3**
time T1	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	20	19	18	17	16	15
time T2	10	9	8	7	6	5	4	3	2	1	10	9	8	7	6	5	4	3	2	1	10	9	8	7	6	5
time T3	5	4	3	2	1	5	4	3	2	1	5	4	3	2	1	5	4	3	2	1	5	4	3	2	1	5
run T1																					**1**	1	1			
run T2											**1**										**1**					
run T3						**1**					**1**	1	1			**1**					**1**	1	1	1		**1**

A More Complex Example

- Note at the end, things "stack up" (one T3 missed)

	Priority		Length		Frequency	
Task 1	2		1		20	
Task 2	1		2		10	
Task 3	3		1		5	
Task 4	0		1		3	

Elapsed time	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
Task executed				T4		T3	T4			T4	T2		T4	T3		T4	T3		T4		T2		T4	T1	T4	T3
time T1	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	20	19	18	17	16	15
time T2	10	9	8	7	6	5	4	3	2	1	10	9	8	7	6	5	4	3	2	1	10	9	8	7	6	5
time T3	5	4	3	2	1	5	4	3	2	1	5	4	3	2	1	5	4	3	2	1	5	4	3	2	1	5
time T4	3	2	1	3	2	1	3	2	1	3	2	1	3	2	1	3	2	1	3	2	1	3	2	1	3	2
run T1																					1	1	1	1		
run T2											1										1					
run T3						1					1	1	1	1		1	1				1	1	1	1	1	1
run T4				1			1			1			1			1			1			1	1		1	

Over-extended Embedded System

This is an “overextended” system because some tasks are missed – several times. There is not enough processor time to complete all of the work. This is covered in more detail in a future lecture.

	Priority	Length	Frequency
Task 1	2	1	20
Task 2	1	2	10
Task 3	3	1	5
Task 4	0	**2**	3

Elapsed ti	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
Task executed				T4		T3	T4			T4		T2		T4		T4		T3	T4		T2		T4		T1	T4		T4		T3	T4
time T1	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	20	19	18	17	16	15	14	13	12	11	10
time T2	10	9	8	7	6	5	4	3	2	1	10	9	8	7	6	5	4	3	2	1	10	9	8	7	6	5	4	3	2	1	10
time T3	5	4	3	2	1	5	4	3	2	1	5	4	3	2	1	5	4	3	2	1	5	4	3	2	1	5	4	3	2	1	5
time T4	3	2	1	3	2	1	3	2	1	3	2	1	3	2	1	3	2	1	3	2	1	3	2	1	3	2	1	3	2	1	3
run T1																					**1**	1	1	1	1						
run T2											**1**	1									**1**										**1**
run T3						**1**					**1**	1	1	1	1	**1**	1	1			**1**	1	1	1	1	**1**	1	1	1	1	**1**
run T4				**1**			**1**			**1**			**1**	1		**1**			**1**			**1**	1		**1**	1		**1**			**1**

Run-To-Completion Scheduler

- Use a ***scheduler*** function to run task functions at the right rates
  - Table stores information per task
    - Period: How many ticks between each task release
    - Release Time: how long until task is ready to run
    - ReadyToRun: task is ready to run immediately
  - "round-robin" scheduler runs forever, examining schedule table which indicates tasks which are ready to run (have been "released")
  - A periodic timer interrupt triggers an ISR, which updates the schedule table
    - Decrements "time until next release"
    - If this time reaches 0, set that task's Run flag and reload its time with the period
- Follows a "run-to-completion" model
  - A task's execution is ***not interleaved*** with any other task
  - Only ISRs can interrupt task
  - After ISR completes, the previously-running task resumes
- Priority is determined by position in table. Hard to change dynamically

RTC Scheduler App Programmer's Interface

API enables control of tasks at more efficient level

- Add Task(task, time period, priority)
  - task: address of task (function name without parentheses)
  - time period: period at which task will be run (in ticks)
  - priority: lower number is higher priority. Also is task number.
  - automatically enables task
- Remove Task(task)
  - removes task from scheduler.
- Run Task(task number)
  - Signals the scheduler that task should run when possible and enables it
- Run RTC Scheduler()
  - Run the scheduler!
  - Never returns
  - There must be at least one task scheduled to run before calling this function.
- Enable_Task(task_number) and Disable_Task(task_number)
  - Set or clear enabled flag, controlling whether task can run or not
- Reschedule_Task(task_number, new_period)
  - Changes the period at which the task runs. Also resets timer to that value.

Limitations of Run-To-Completion Scheduler

- Tasks run to completion – problem with long tasks
  - Maximum response time for a task is the duration of the longest task
  - Long tasks complicate programming
    - No elegant way to start an operation (e.g. flash programming) and yield processor for 10 ms
    - Can improvise
      - Trigger another task
      - Use a state machine within this task
- Prioritization implies unfair processor allocation – starvation possible

Function-Queue Scheduling

- Interrupt routine enqueues a function to be called by **`main`**
- Queue provides scheduling flexibility
  - Functions can be enqueued with any order desired
  - Use priority of device to determine position in queue

```
void interrupt HandleDeviceA(){
  /* do urgent work for A */
  ...
  Enqueue(Queue,FinishDeviceA);
}
...
void FinishDeviceA(void) {
  /* do remainder of A's work */
}

void main(void) {
  while (TRUE) {
    while (NotEmpty(Queue)) {
      f = Dequeue(Queue);
      f();
    }
  }
}
```

Limitations of Function-Queue Scheduling

What if a long lower-priority function (**`FinishDeviceC`**) is executing and we need to run **`FinishDeviceA`**?

- Must wait until **`FinishDeviceC`** completes
- $\max\left(T_{response}(j)\right) = \max\left(T_{task}(t)\right)\forall t + \sum_{\forall i} T_{ISR}(i)$
- Cooperative multitasking, no pre-emption

What if the lowest-priority functions never get to run?

- Heavily loaded system

Real-Time OS (*RTOS, Kernel, ...*)

As with previous methods

- ISRs handle most urgent operations
- Other code finishes remaining work

Differences:

- The RTOS can ***preempt*** (suspend) a task to run something else.
- Signaling between ISRs and task code (service functions) handled by RTOS.
- We don't write a loop to choose the next task to run. RTOS chooses based upon priority.

Why These Differences Matter

- Signaling handled by RTOS
  - Shared variables not needed, so programming is easier
- RTOS chooses next task to run
  - Programming is easier
- RTOS can preempt tasks, and therefore schedule freely
  - System can control *task code response time* (in addition to interrupt routine response time)
  - Worst-case wait for highest-priority task doesn't depend on duration of other tasks.
  - System's response (time delay) becomes more stable
    - A task's response time depends only on higher-priority tasks (*usually* – more later)

More RTOS Issues

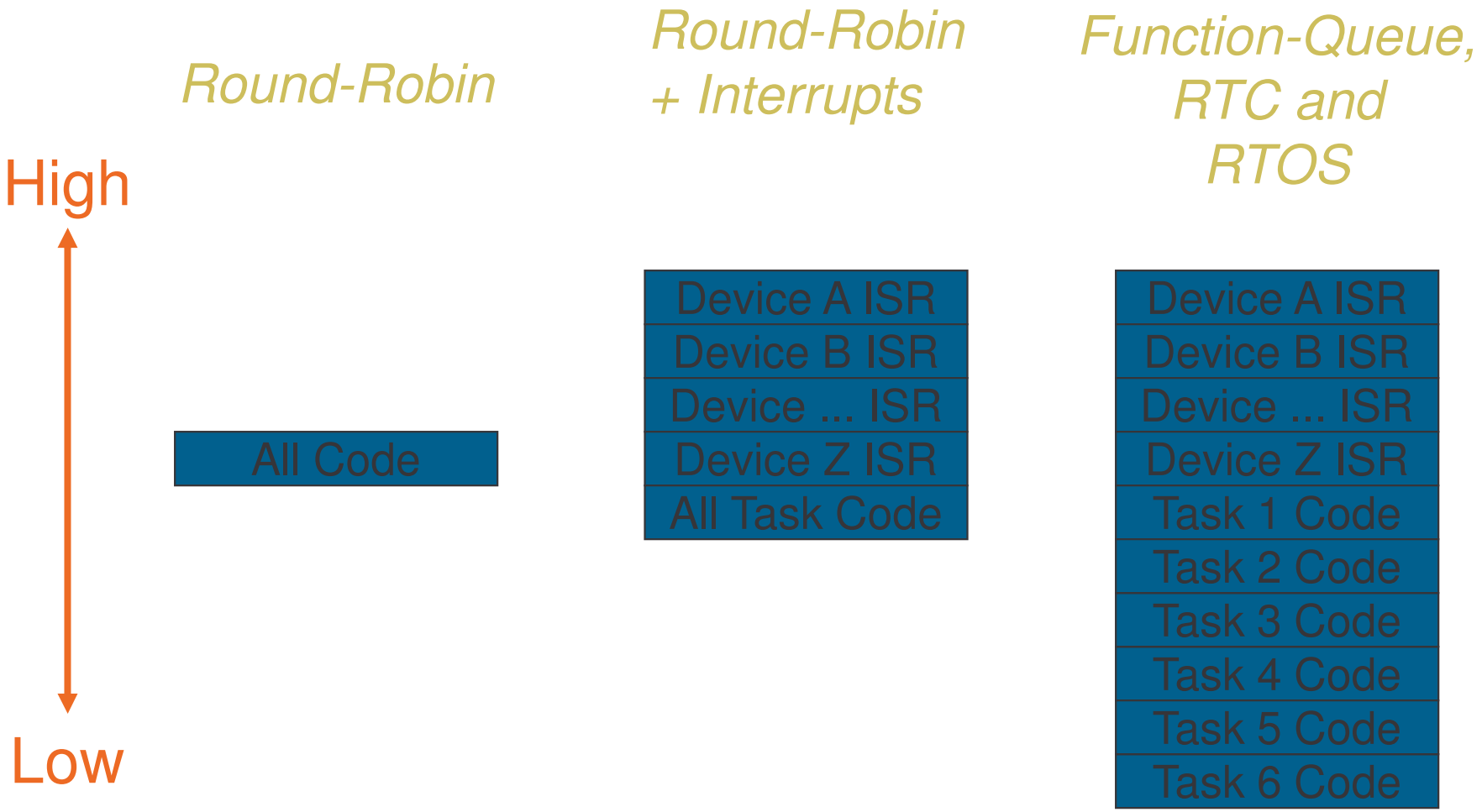
Many RTOS's on the market

- Already built and debugged
- Debug tools typically included
- Full documentation (and source code) available

Main disadvantage: RTOS costs resources (e.g. uC/OSII compiled for 80186. YMMV)

- Compute Cycles: 4% of CPU
- Money: ???
- Code memory: 8.3 KBytes
- Data memory: 5.7 KBytes

Comparison of Priority Levels Available



Software Architecture Characteristics

	Priorities Available	**Worst Case $T_{Response}$ for Highest Priority Task Code**	**Stability of $T_{Response}$ when Code Changes**	**Simplicity**
Round-robin	None	ΣT_{Task}	Poor	Very simple
Round-robin with interrupts	Prioritized interrupt routines, then task code at same priority	$\Sigma T_{Task} + \Sigma T_{Interrupt}$	Good for interrupts, poor for task code	Must deal with shared data (interrupts/tasks)
RTC and Function-queue scheduling	Prioritized interrupt routines, then prioritized task code	$\max(T_{Task}) + \Sigma T_{Interrupt}$	Relatively good	Must deal with shared data and must write/get scheduler code
Real-time operating system	Prioritized interrupt routines, then prioritized task code	$\Sigma T_{Interrupt} + T_{OS}$	Very good	Most complex (much is handled by RTOS)

Review of Scheduler Information

Scheduler provided in these slides

Details

- Scheduler uses a software timer per task
- All software timers are decremented using a timer tick based on the Timer B0 hardware overflow interrupt
- Each task runs to completion before yielding control of MCU back to Scheduler (*non-preemptive)*

Round Robin Scheduler API

Init_RR_Scheduler(void)
- Initialize tick timer B0 and task timers

Add Task(task, time period, priority)
- task: address of task (function name without parentheses)
- time period: period at which task will be run (in ticks)
- priority: lower number is higher priority. Also is task number.
- automatically enables task
- return value: 1 – loaded successfully, 0 – unable to load

Remove Task(task)
- removes task from scheduler.

Run Task(task number)
- Signals the scheduler that task should run when possible and enables it

Run RR Scheduler()
- Run the scheduler!
- Never returns
- There must be at least one task scheduled to run before calling this function.

Enable_Task(task_number) and Disable_Task(task_number)
- Set or clear enabled flag, controlling whether task can run or not

Reschedule_Task(task_number, new_period)
- Changes the period at which the task runs. Also resets timer to that value.

Set up Timer B0 in Init_RR_Scheduler

Set up B0 timer to generate an interrupt every 1 millisecond

```
// default tb0 will be = 65536 (timer tick = 5.4613 ms)
// if you load tb0 = 12000, timer tick will = 1.0000ms

init_Task_Timers();          // Initialize all tasks
tb0 = 12000;                 // 1 ms timer tick
DISABLE_INTS
tb0ic = 1;                   // Timer B0 overflow
ENABLE_INTS
tb0st = 1;                   // start timer B0
```

Task List Structure

```
#define USE_ROUND_ROBIN_SCH      1
  // Set to 1 if using Round Robin Task Scheduler
#define MAX_TASKS  5
  // Set maximum number of tasks to be used in system
   // Will affect performance.

typedef struct {
  int initialTimerValue;          // “frequency” of task
  int timer;                      // time to next “run”
  int run;                        // binary – 1 = “run now”
  int enabled;
  void (* task)(void);            // address of function
} task_t;

task_t GBL_task_list[MAX_TASKS];
int    GBL_run_scheduler=0;
```

Running the Scheduler

```
void Run_RR_Scheduler(void) {  // Always running
  int i;
  GBL_run_scheduler = 1;
  while (1) {                    // Loop forever & Check each task
    for (i=0 ; i<MAX_TASKS ; i++) {
      // If this is a scheduled task
      if (GBL_task_list[i].task != NULL) {
        if (GBL_task_list[i].enabled == 1) {
          if (GBL_task_list[i].run == 1) {
            GBL_task_list[i].run=0;  // Reset task timer
            GBL_task_list[i].task(); // Run the task
            break;
          }
        }
      }
    }
  }
}
```

	Priority	Length	Frequency
Task 1	2	1	20
Task 2	1	2	10
Task 3	3	1	5

Elapsed time	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
Task executed						T3					T2		T3			T3					T2		T1	T3		T3
time T1	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	20	19	18	17	16	15
time T2	10	9	8	7	6	5	4	3	2	1	10	9	8	7	6	5	4	3	2	1	10	9	8	7	6	5
time T3	5	4	3	2	1	5	4	3	2	1	5	4	3	2	1	5	4	3	2	1	5	4	3	2	1	5
run T1																					1	1	1			
run T2											1										1					
run T3						1					1	1	1			1					1	1	1	1		1

Task List Initialization

```
void init_Task_Timers(void) { // Initialize all tasks
  int i;
  for (i=0 ; i<MAX_TASKS ; i++) {
    GBL_task_list[i].initialTimerValue = 0;
    GBL_task_list[i].run = 0;
    GBL_task_list[i].timer = enabled = 0;
    GBL_task_list[i].task = NULL;
  }
}
```

	Priority	Length	Frequency
Task 1	2	1	20
Task 2	1	2	10
Task 3	3	1	5

Elapsed time	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
Task executed						T3					T2		T3			T3					T2			T1	T3		T3
time T1	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	20	19	18	17	16	15
time T2	10	9	8	7	6	5	4	3	2	1	10	9	8	7	6	5	4	3	2	1	10	9	8	7	6	5
time T3	5	4	3	2	1	5	4	3	2	1	5	4	3	2	1	5	4	3	2	1	5	4	3	2	1	5
run T1																					1	1	1			
run T2											1										1					
run T3						1					1	1	1			1					1	1	1	1		1

Adding a Task

```
int addTask(void (*task)(void), int time, int priority)
{
  unsigned int t_time;
  /* Check for valid priority */
  if (priority >= MAX_TASKS || priority < 0) return 0;
  /* Check to see if we are overwriting an already scheduled
   task */
  if (GBL_task_list[priority].task != NULL) return 0;
  /* Schedule the task */
   GBL_task_list[priority].task = task;
   GBL_task_list[priority].run  = 0;
   GBL_task_list[priority].timer = time;
   GBL_task_list[priority].enabled = 1;
   GBL_task_list[priority].initialTimerValue = time;
  return 1;
}
```

Task Selection

```
// Make sure to load the vector table with this ISR addr
#pragma INTERRUPT tick_timer_intr
void tick_timer_intr(void) {
static char i;
for (i=0 ; i<MAX_TASKS ; i++) { // If scheduled task
   if (GBL_task_list[i].task != NULL) {
     if (GBL_task_list[i].enabled == 1) {
       if (GBL_task_list[i].timer) {
         if (--GBL_task_list[i].timer == 0){
                GBL_task_list[i].run = 1;
                GBL_task_list[i].timer =
                      GBL_task_list[i].initialTimerValue;
         }
       }
     }
   }
}
```

	Priority	Length	Frequency
Task 1	2	1	20
Task 2	1	2	10
Task 3	3	1	5

Elapsed time	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
Task executed						T3					T2		T3			T3					T2		T1	T3		T3
time T1	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	20	19	18	17	16	15
time T2	10	9	8	7	6	5	4	3	2	1	10	9	8	7	6	5	4	3	2	1	10	9	8	7	6	5
time T3	5	4	3	2	1	5	4	3	2	1	5	4	3	2	1	5	4	3	2	1	5	4	3	2	1	5
run T1																					1	1	1			
run T2											1										1					
run T3						1					1	1	1			1					1	1	1	1		1

Removing a Task

```
void removeTask(void (* task)(void))
{
  int i;

  for (i=0 ; i<MAX_TASKS ; i++) {
    if (GBL_task_list[i].task == task) {
      GBL_task_list[i].task = NULL;
       GBL_task_list[i].timer = 0;
      GBL_task_list[i].initialTimerValue = 0;
       GBL_task_list[i].run = enabled = 0;
      return;
    }
  }
}
```

Enabling or Disabling a Task

```
void Enable_Task(int task_number)
{
  GBL_task_list[task_number].enabled = 1;
}

void Disable_Task(int task_number)
{
  GBL_task_list[task_number].enabled = 0;
}
```

Rescheduling a Task

Changes period of task and resets counter

```
void Reschedule_Task(int task_number, int new_timer_val)
{
  GBL_task_list[task_number].initialTimerValue =
   new_timer_val;
  GBL_task_list[task_number].timer = new_timer_val;
}
```

Start Round Robin System

To run RR scheduler, first add the function (task):

```
addTask(flash_redLED, 25, 3);
addTask(sample_ADC, 500, 4);
```

Then, the last thing you do in the main program is:

```
Run_RR_Scheduler();
```

Big Picture

Methods learned so far

- We've been using a *foreground/background* system
  - Interrupt service routines run in foreground
  - Task code runs in background
- Limitations
  - Must structure task functions to run to completion, regardless of "natural program structure" – can only yield processor at end of task
  - Response time of task code is not easily controlled, in worst case depends on how long each other task takes to run

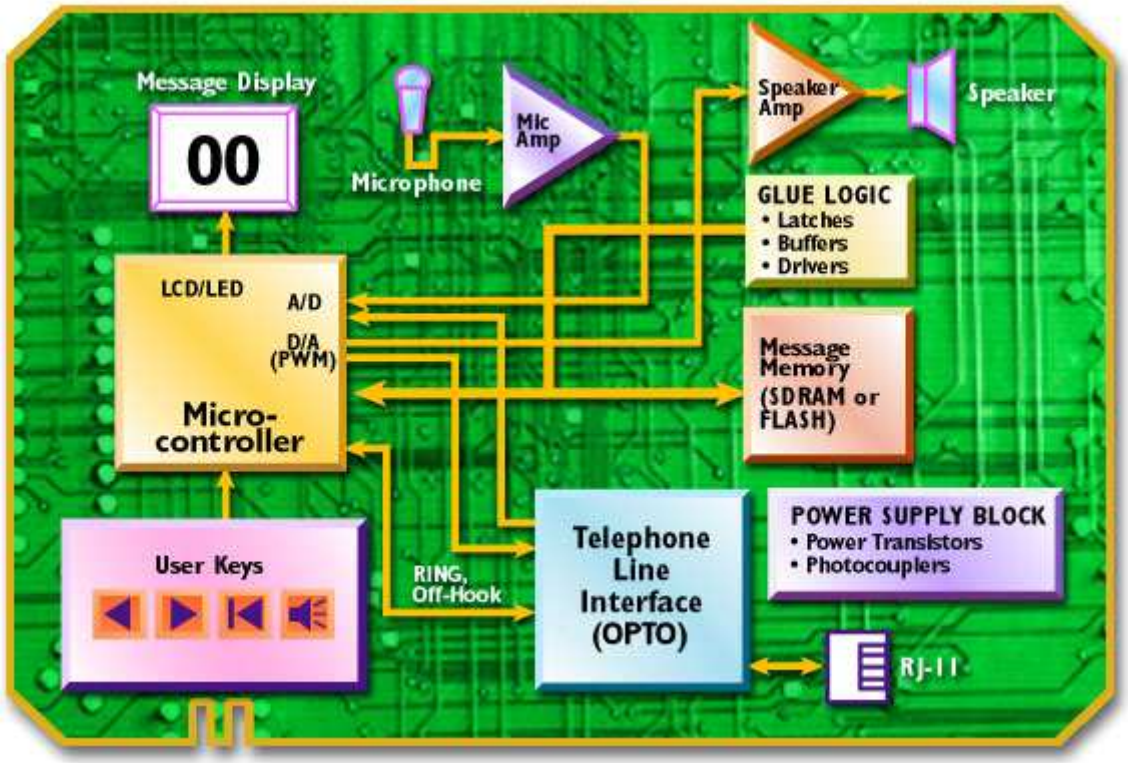
What we will learn next

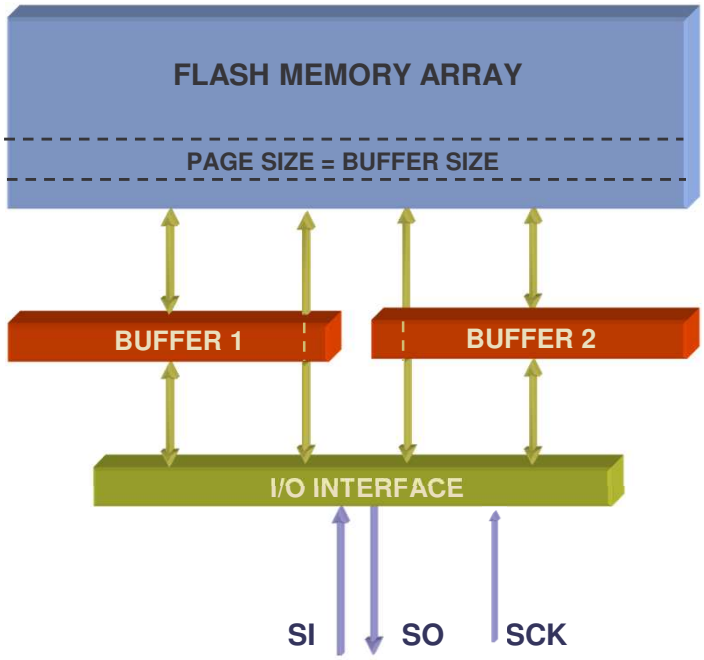
- How to share processor flexibly among multiple tasks, while not requiring restructuring of code

Goal: share MCU efficiently

- Embedded Systems: To simplify our program design by allowing us to partition design into multiple independent components
- PCs/Workstations/Servers: To allow multiple users to share a computer system

Example: Secure Answering Machine (SAM)





Testing the limits of our cooperative round-robin scheduler

Secure Answering Machine

- Stores encrypted voice messages in serial Flash memory
- Want to delete messages fully, not just remove entry from directory (as with file systems for PCs)
- Also have a user interface: LCD, switches

SAM Delete Function and Timing

```
void Delete_Message(unsigned mes_num) {
…
LCD("Are you sure?");                    // 10 ms
get_debounced_switch(&k, 5);             // 400 ms min, 5 s max
if (k == CANCEL_KEY) {
   LCD("Cancelled");                     // 10 ms
} else if (k == TIMEOUT) {
   LCD("Timed Out");                     // 10 ms
} else {
  LCD("Erasing");                        // 10 ms
   Flash_to_Buffer(DIR_PAGE);            // 250 us
   Read_Buffer(dir);                     // 100 us
   …                                     // find offsets
   …                                     // erase dir. entry
   Write_to_Buffer(dir);                 // 6 us
   Buffer_to_Flash(DIR_PAGE);            // 20 ms
   Flash_to_Buffer(data_page);
   …                                     // overwrite msg: 50 us
   Buffer_to_Flash(data_page);           // 20 ms
   LCD("Done");
}
```

Cooperative RR Scheduler?

Since task must **Run To Completion…**

The delete function could take up to five seconds to run, halting all other tasks (but interrupts run)

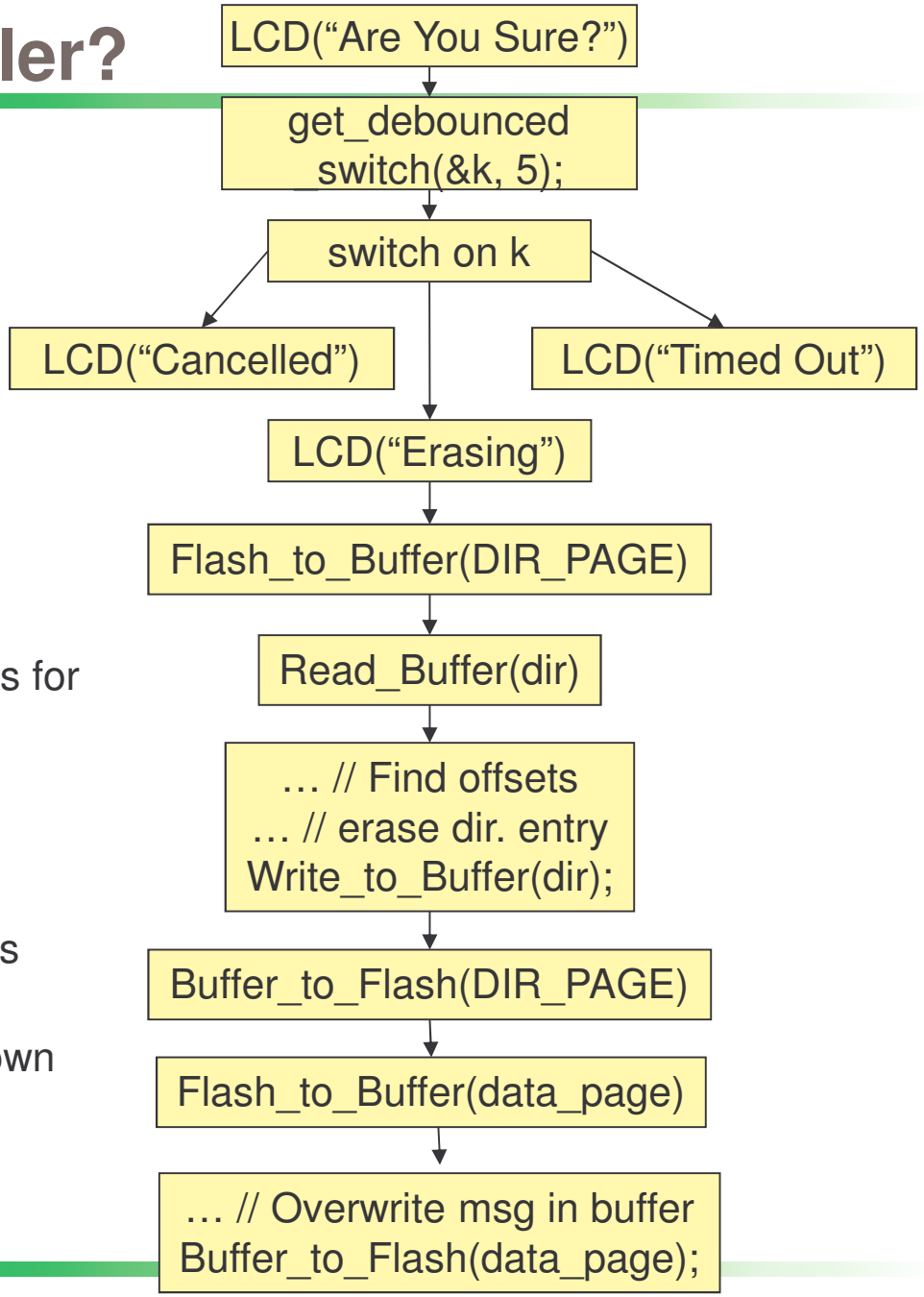
Other software needs to keep running, so break this into pieces. Run one piece at a time.

How to split?

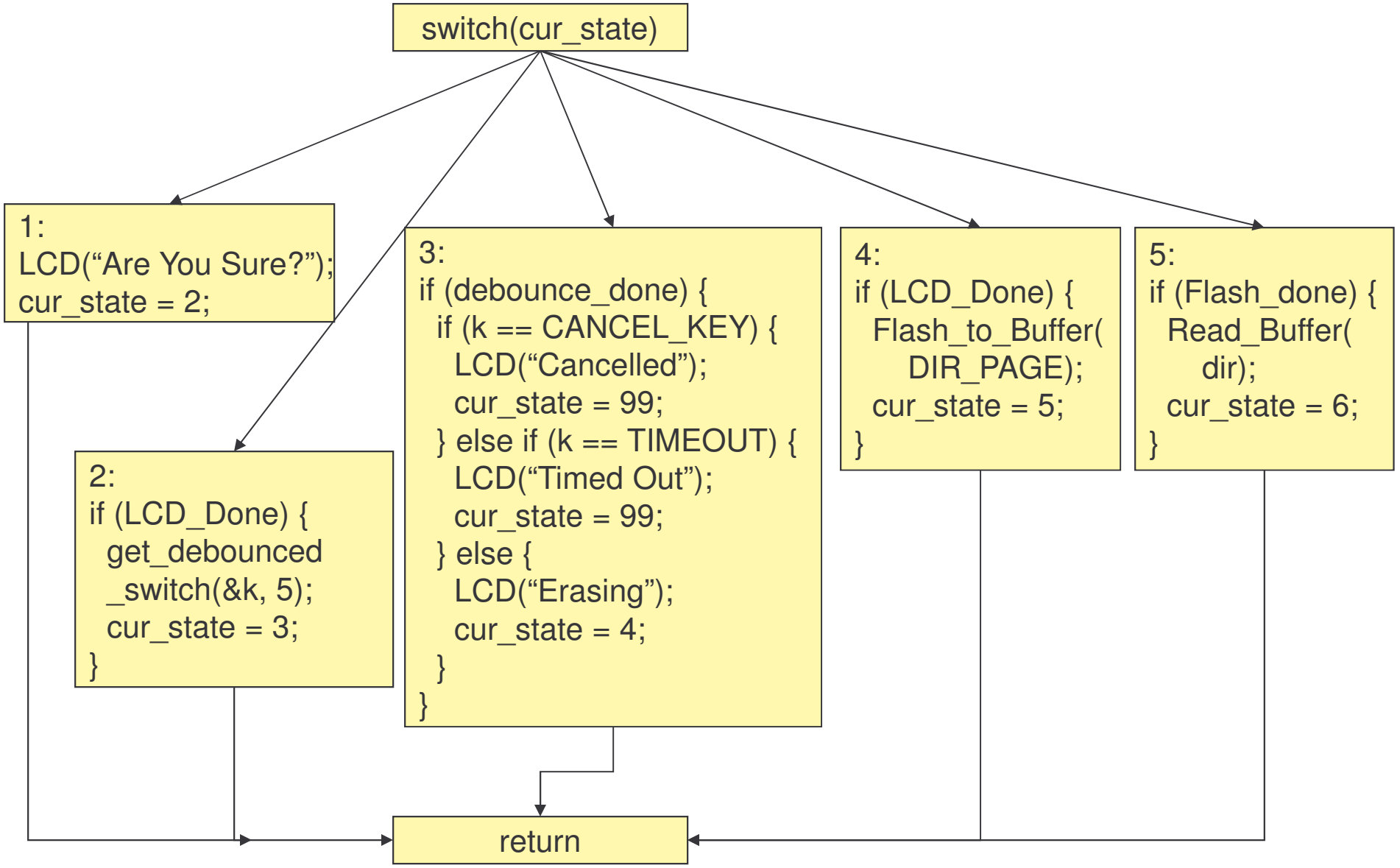
- Each piece ends where processor waits for user (e.g. debounced switch) or other devices (Flash, LCD).

How to control execution of pieces?

1. Use a task per piece, use calls to Reschedule_Task and Disable_Task as needed
   - Need 13 different tasks (12 shown here)
2. Use a state machine within one task



State Machine in One Task



Daydreaming

Some functions are causing trouble for us – they use slow devices which make the processor wait

- LCD: controller chip on LCD is slow
- DataFlash: it takes time to program Flash EEPROM
- Switch debouncing: physical characteristics of switch, time-outs

Wouldn't it be great if we could …

- Make those slow functions *yield the processor* to other tasks?
- Not have the processor start running that code again *until the device is ready*?
  - Maybe even have the processor interrupt less-important tasks?
- *Avoid breaking up one task* into many tasks, or a state machine?
- Open ourselves up to a whole new species of bugs, bugs which are very hard to duplicate and track down?

Preemptive Scheduling Kernel

What we need is a ***kernel***

- Shares the processor among multiple concurrently running tasks/threads/processes
- Can forcibly switch the processor from thread A to B and resume B later (preemption)
- Can resume threads when their data is ready
- Can simplify inter-thread communication by providing mechanisms
- The heart of any operating system

Terminology: “Kernel Mode”

- PCs and workstations don’t expose all of the machine to the user’s program
- Only code in *kernel* or *supervisor* mode have full access
- Some high-end embedded processors have a restricted mode (e.g. ARM, MIPS)

Operating Systems (for PCs and Workstations)

Two perspectives

- Extended Machine – top-down view (using abstractions)
  - File System: make a magnetic platter, read/write head, spindle motor and head servo look like a hierarchical collection of directories containing files and other directories
  - Virtual Memory: make a disk and 512 MB of RAM look like 4 GB of RAM
- Resource Manager – bottom-up view
  - Share access to resources
  - Keep them from interfering

Common PC/Workstation operating system features

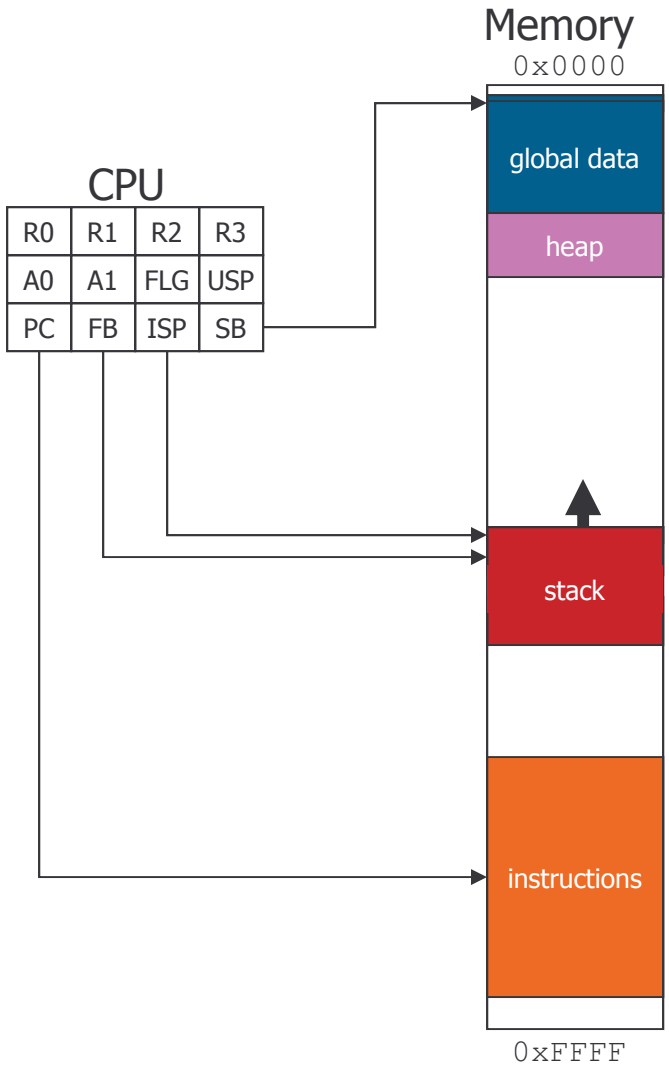
- Process management – share the processor
- Process synchronization and communication
- Memory management
- File management
- Protection
- Time management
- I/O device access

*For embedded systems, we care mostly about preemptive thread management – **sharing the processor***

What Execution State Information Exists?

A program, process or thread in execution which has *state information*…

- Current instruction – identified with program counter
- Call stack – identified with stack pointer
  - Arguments, local variables, return addresses, dynamic links
- Other CPU state
  - Register values (anything which will be shared and could be affected by the other processes) – general purpose registers, stack pointer, etc.
  - Status flags (zero, carry, interrupts enabled, carry bit, etc.)
- Other information as well
  - Open files, memory management info, process number, scheduling information
  - Ignore for now



Processes vs. Threads

Process – No information is visible to other processes (nothing is shared)

Thread – Shares address space and code with other threads (also called *lightweight process)*

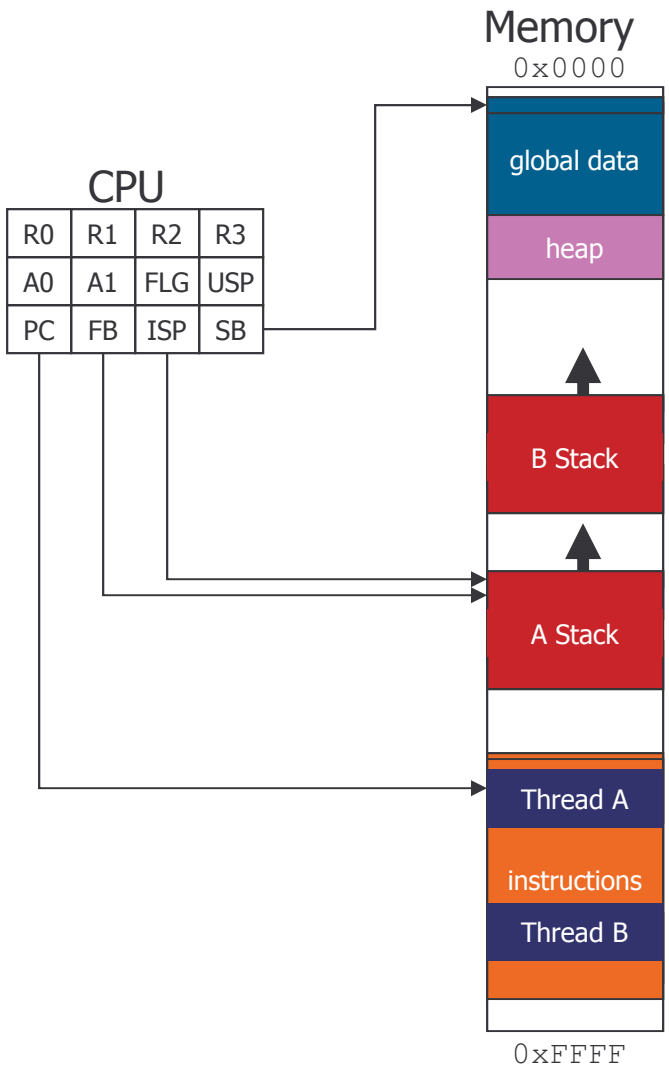
One big side effect: context switching time varies

- Switching among processes requires swapping large amounts of information
- Switching among threads requires swapping much less information (PC, stack pointer and other registers, CPU state) and is much faster

For this discussion, concepts apply equally to threads and processes

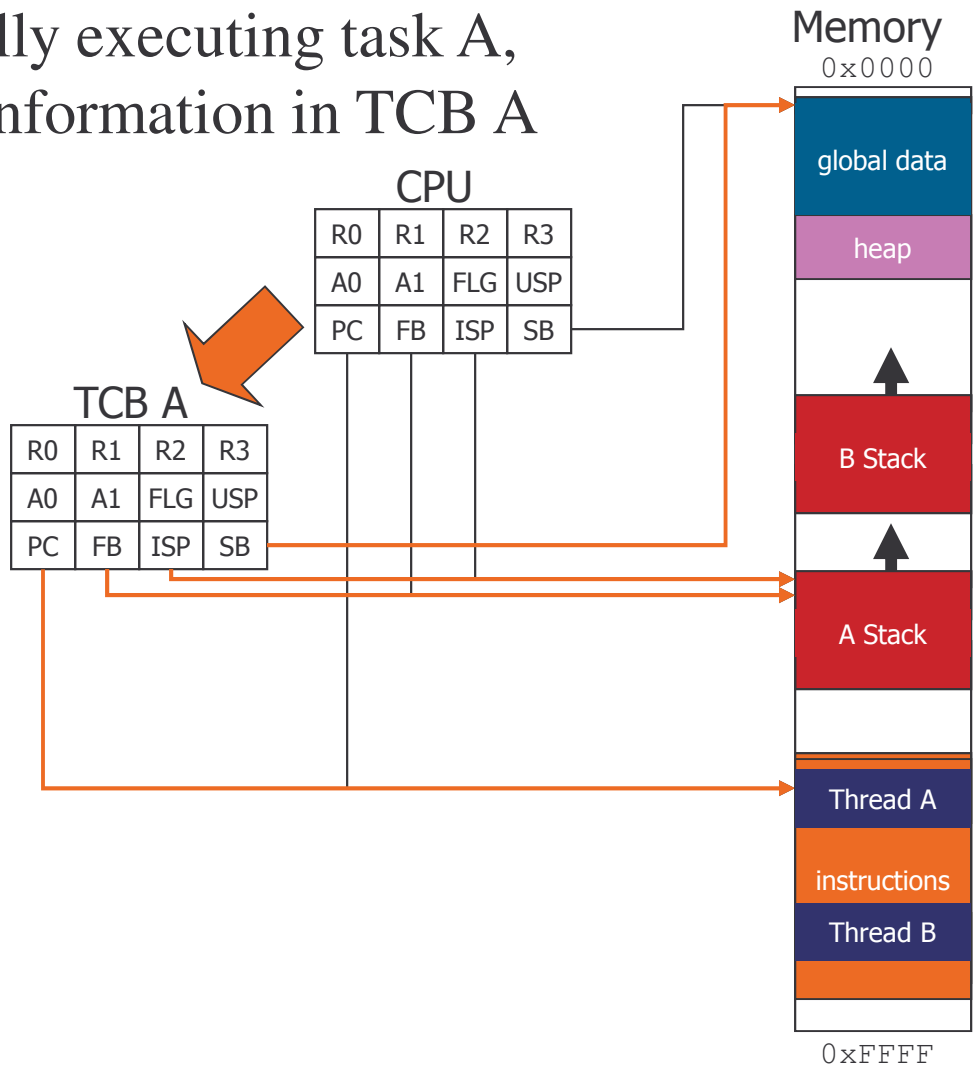
Maintaining State for Multiple Threads

- Store this thread-related information in a task/thread control block (TCB)
  - process control block = PCB
- Shuffling information between CPU and multiple TCBs lets us share processor
- Consider case of switching from thread A to thread B
  - Assume we have a call stack for each thread
  - Assume we can share global variables among the two threads
    - Standard for threads
    - For M16C architecture, SB register is same for both threads



Step 1. Copy CPU State into TCB A

CPU is initially executing task A, so save this information in TCB A

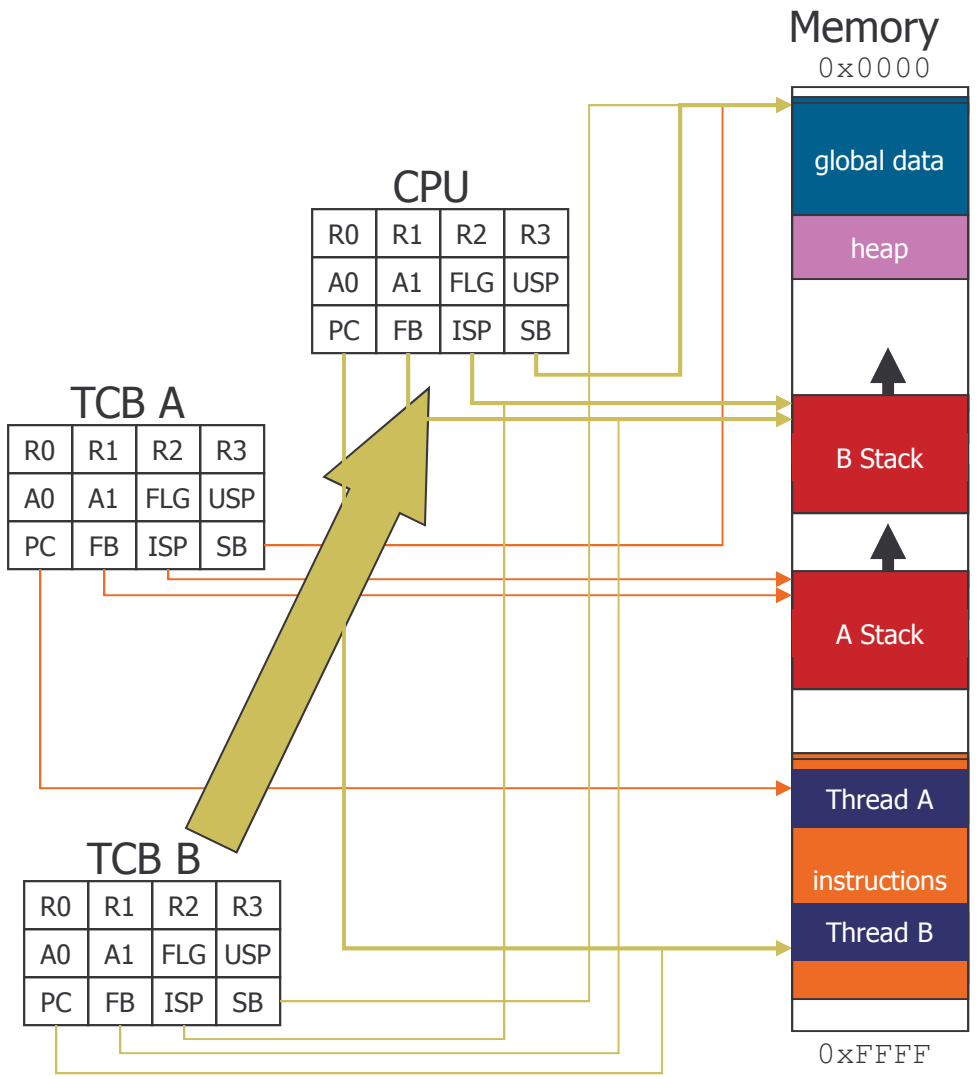


Step 2. Reload Old CPU State from TCB B

Reloading a previously saved state configures the CPU to execute task B from where it left off

This *context switching* is performed by the *dispatcher* code

Dispatcher is typically written in assembly language to gain access to registers not visible to C programmer

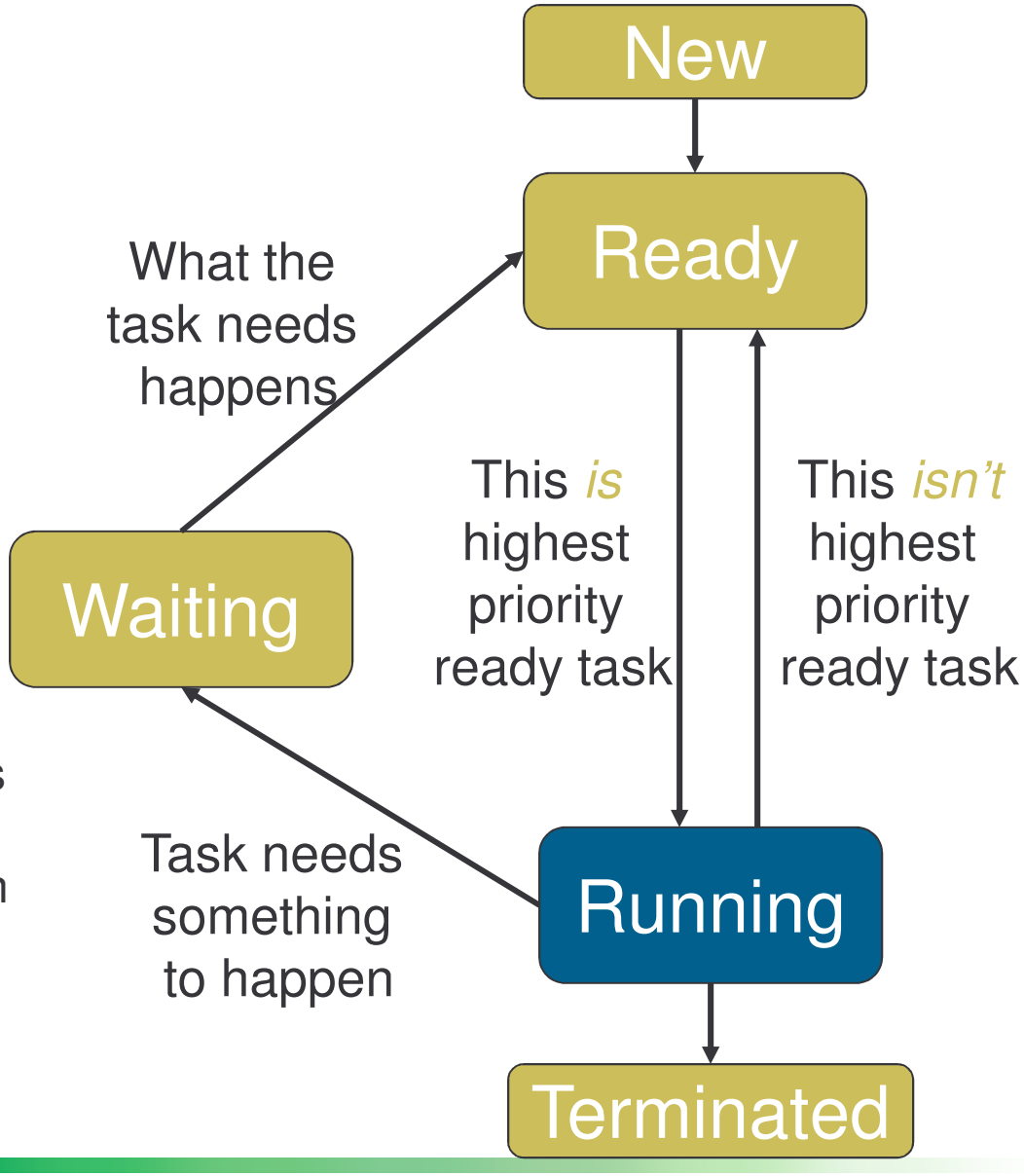


Thread States

Now that we can share the CPU, let's do it!

Define five possible states for a thread to be in

- New – just created, but not running yet
- Running – instructions are being executed (only one thread can be running at a time!)
- Waiting/Blocking – thread is waiting for an event to occur
- Ready – process is not waiting but not running yet (is a candidate for running)
- Terminated – process will run no more

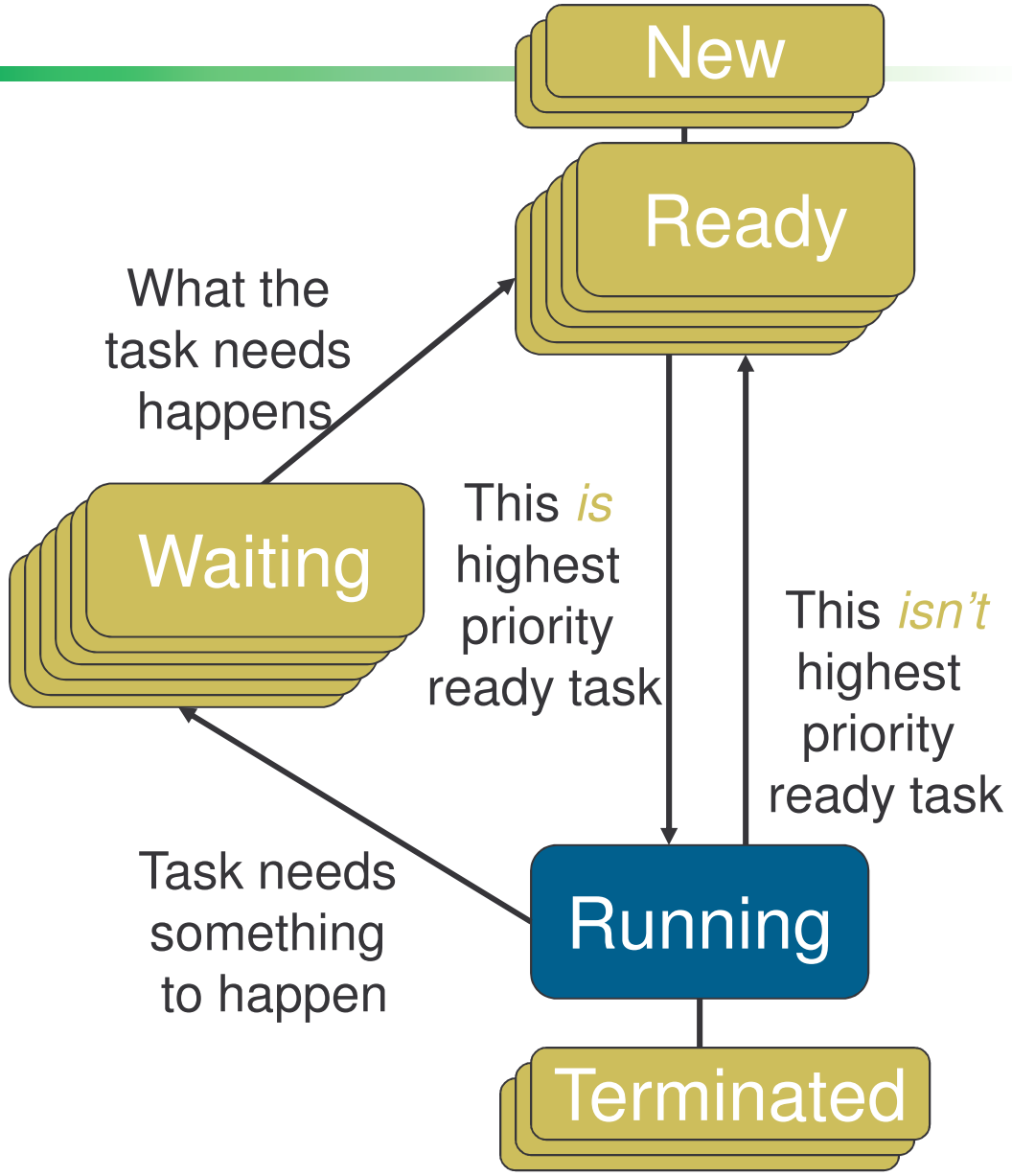


Thread Queues

Create a queue for each state (except running)

Now we can store thread control blocks in the appropriate queues

Kernel moves tasks among queues/processor registers as needed



Thread State Control

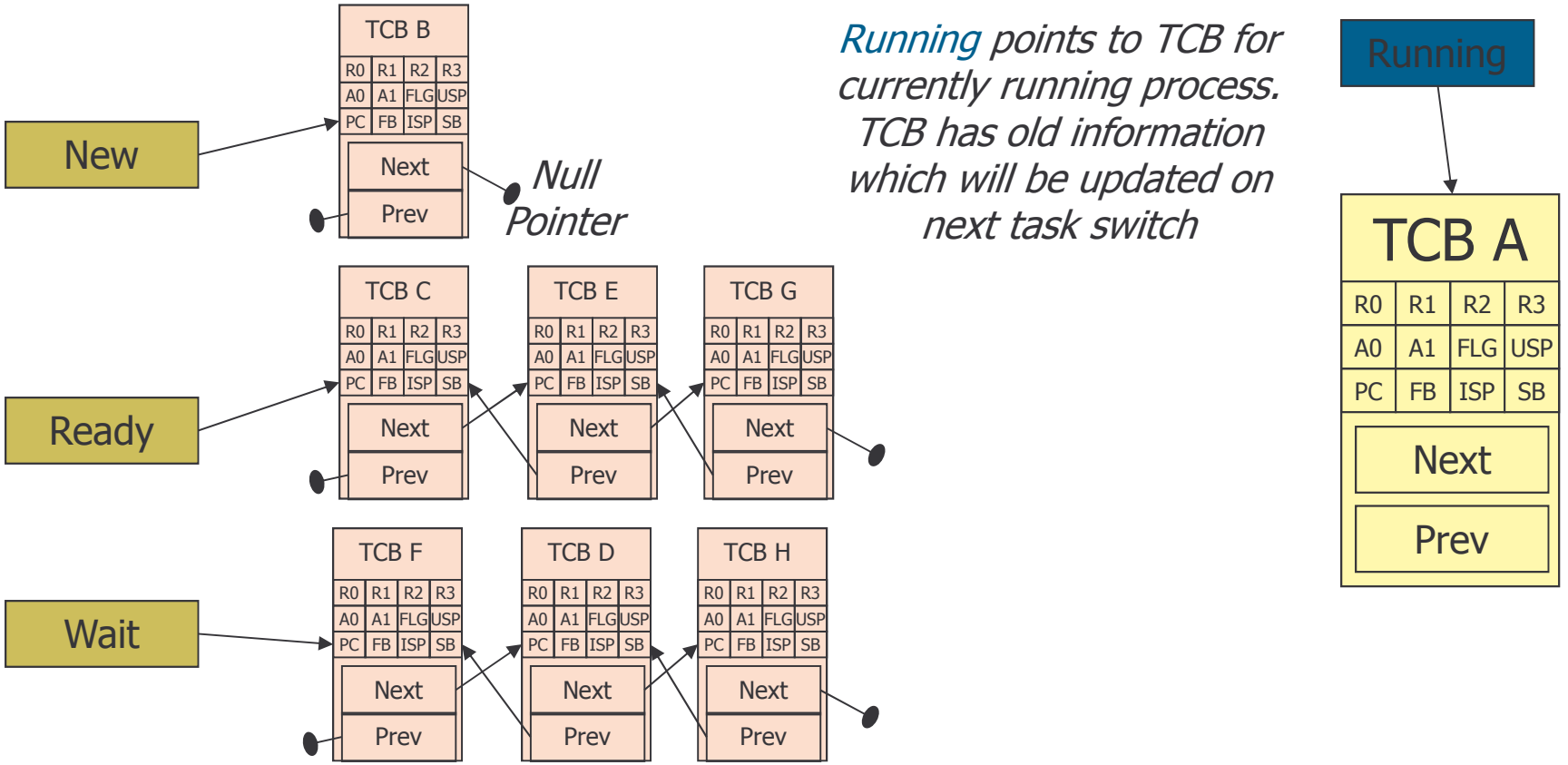
Use OS scheduler to keep track of threads and their states

- For each state, OS keeps a queue of TCBs for all processes in that state
- Moves TCBs between queues as thread state changes
- OS's scheduler chooses among *Ready* threads for execution based on priority
- Scheduling Rules
  - Only the thread itself can decide it should be *waiting (blocked)*
  - A *waiting* thread never gets the CPU. It must be signaled by an ISR or another thread.
  - Only the scheduler moves tasks between *ready* and *running*

What changes the state of a thread?

- The OS receives a timer tick which forces it to decide what to run next
- The thread voluntarily yields control
- The thread requests information which isn't ready yet

Overview of Data Structures for Scheduler



Add Next, Prev pointers in each TCB to make it part of a doubly linked list

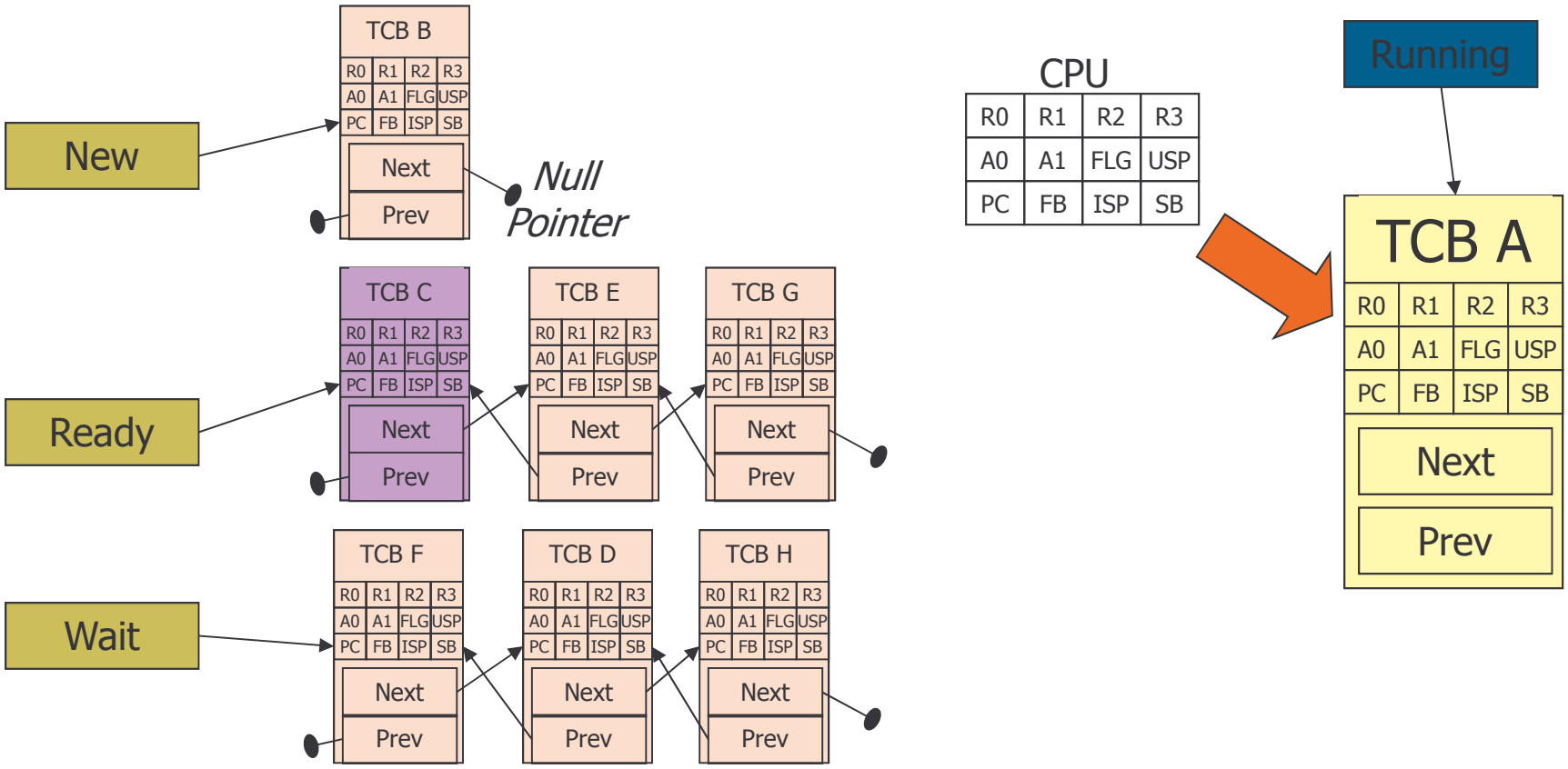
Keep track of all TCBs

- Create a pointer for each queue: Ready, Wait, New
- Create a pointer for the currently running task's TCB

Example: Context Switch

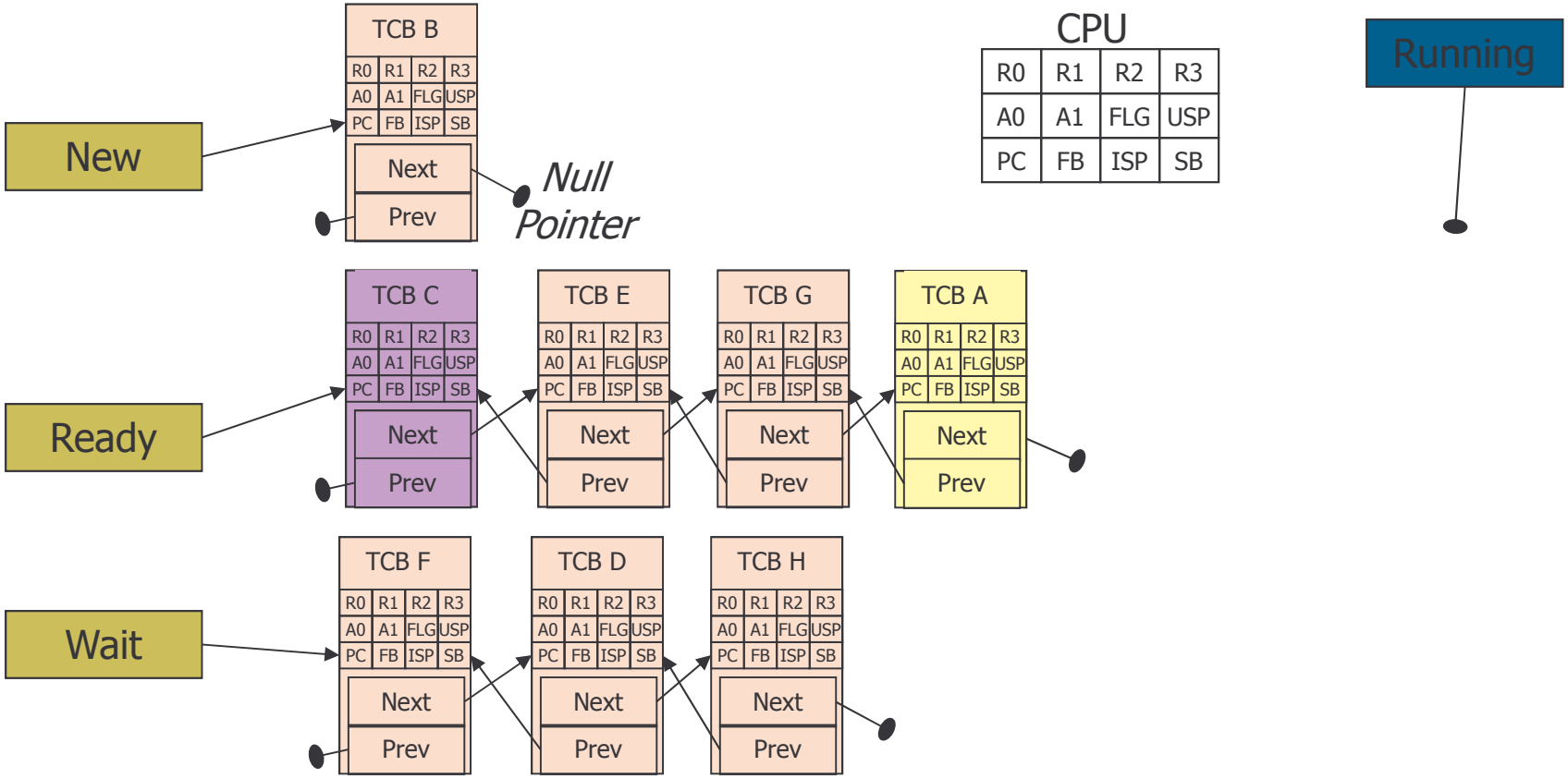
Thread A is running, and scheduler decides to run thread C instead. For example, thread A is still able to run, but has lower priority than thread C.

Start by copying CPU state into TCB A



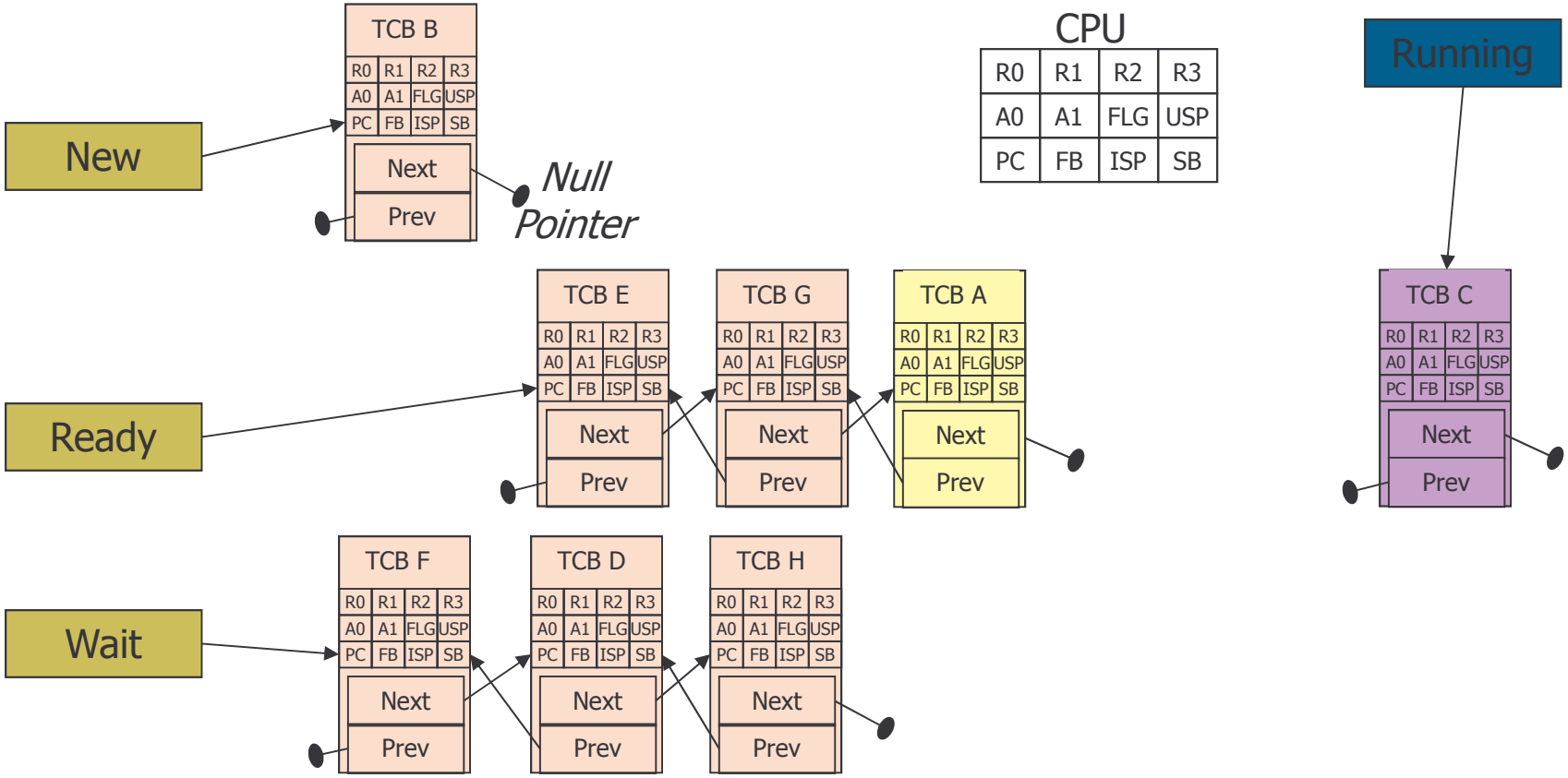
Example: Context Switch

Insert TCB A into ready queue by modifying appropriate pointers



Example: Context Switch

Remove thread C from the ready queue and mark it as the thread to run next



Example: Context Switch

Copy thread C's state information back into the CPU and resume execution