Preemptive Scheduling

Lecture 18

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

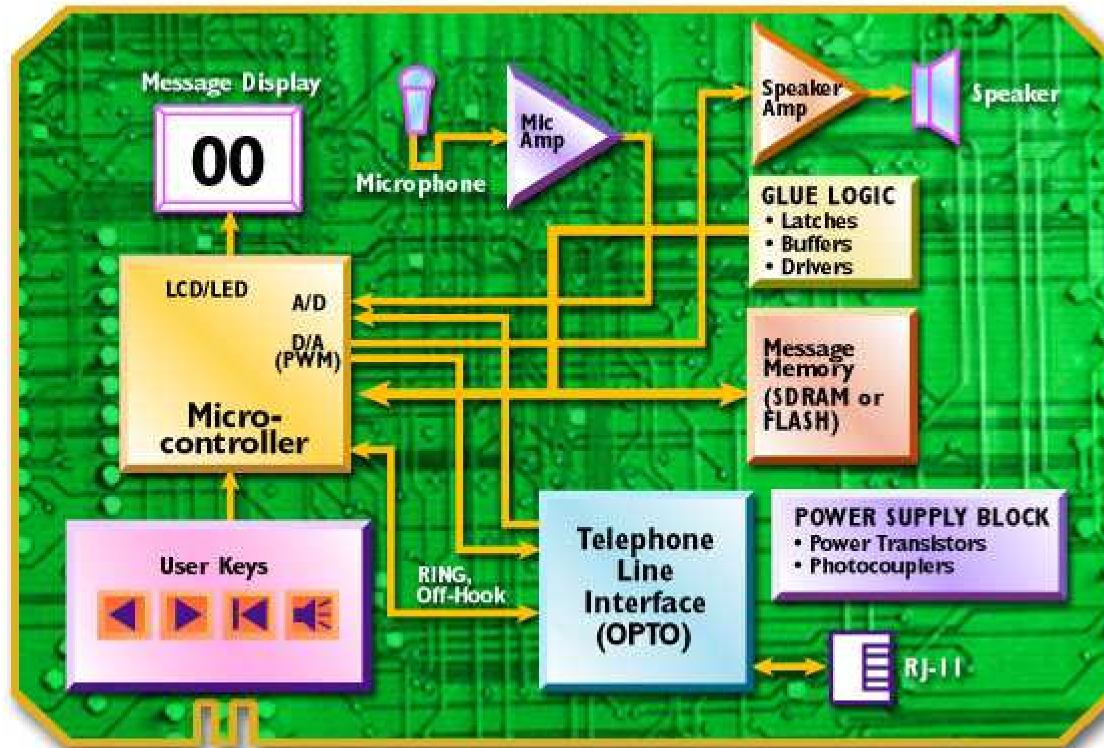
What we will learn next

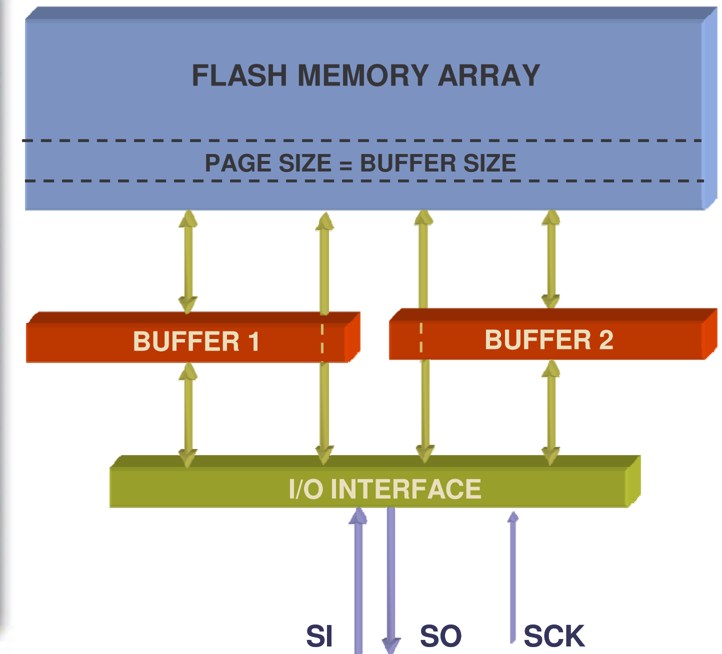
- How to share processor flexibly among multiple tasks, while not requiring restructuring of code

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)

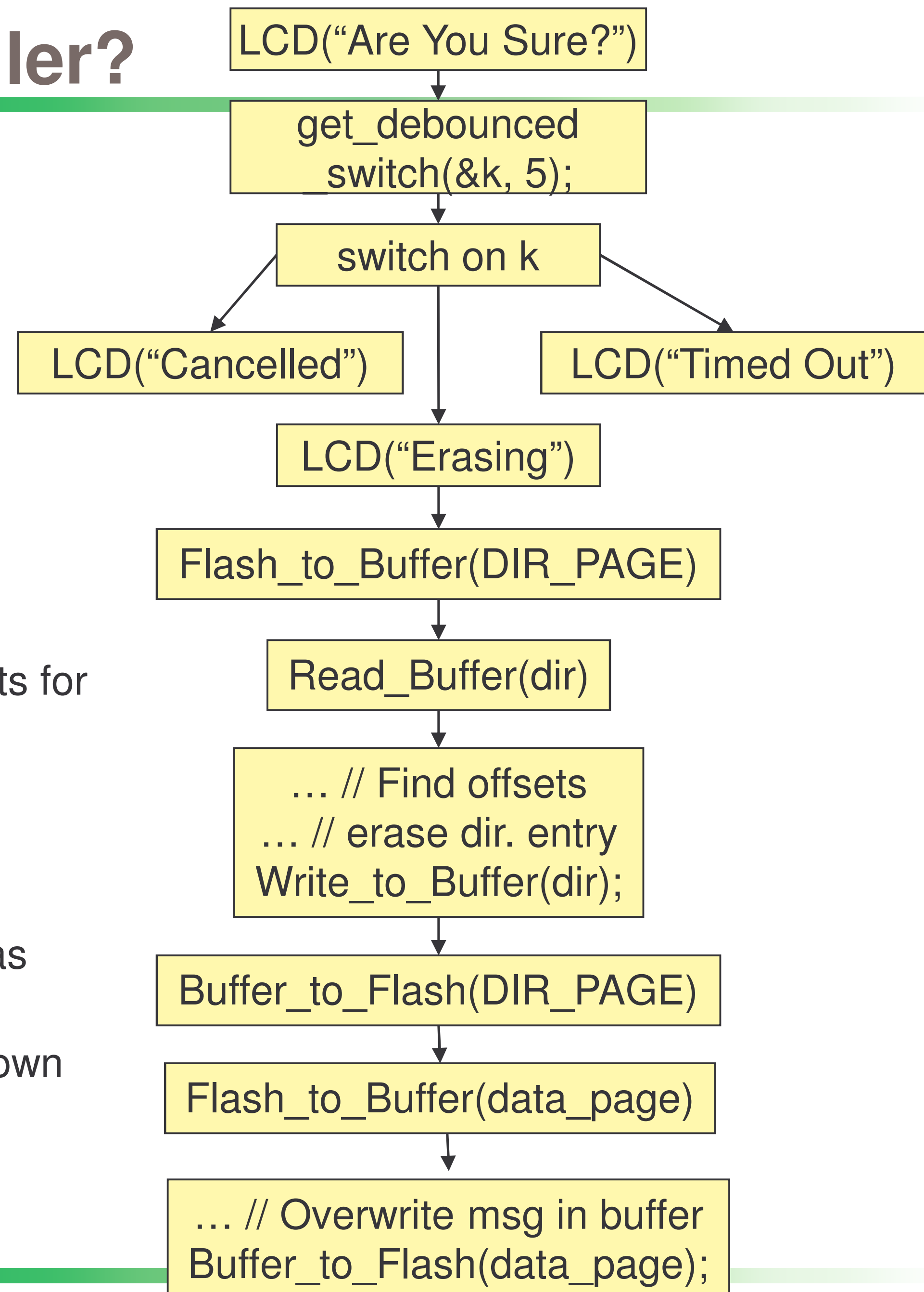
Other software needs to keep running, so break this into pieces. Run one piece at a time.

How to split?

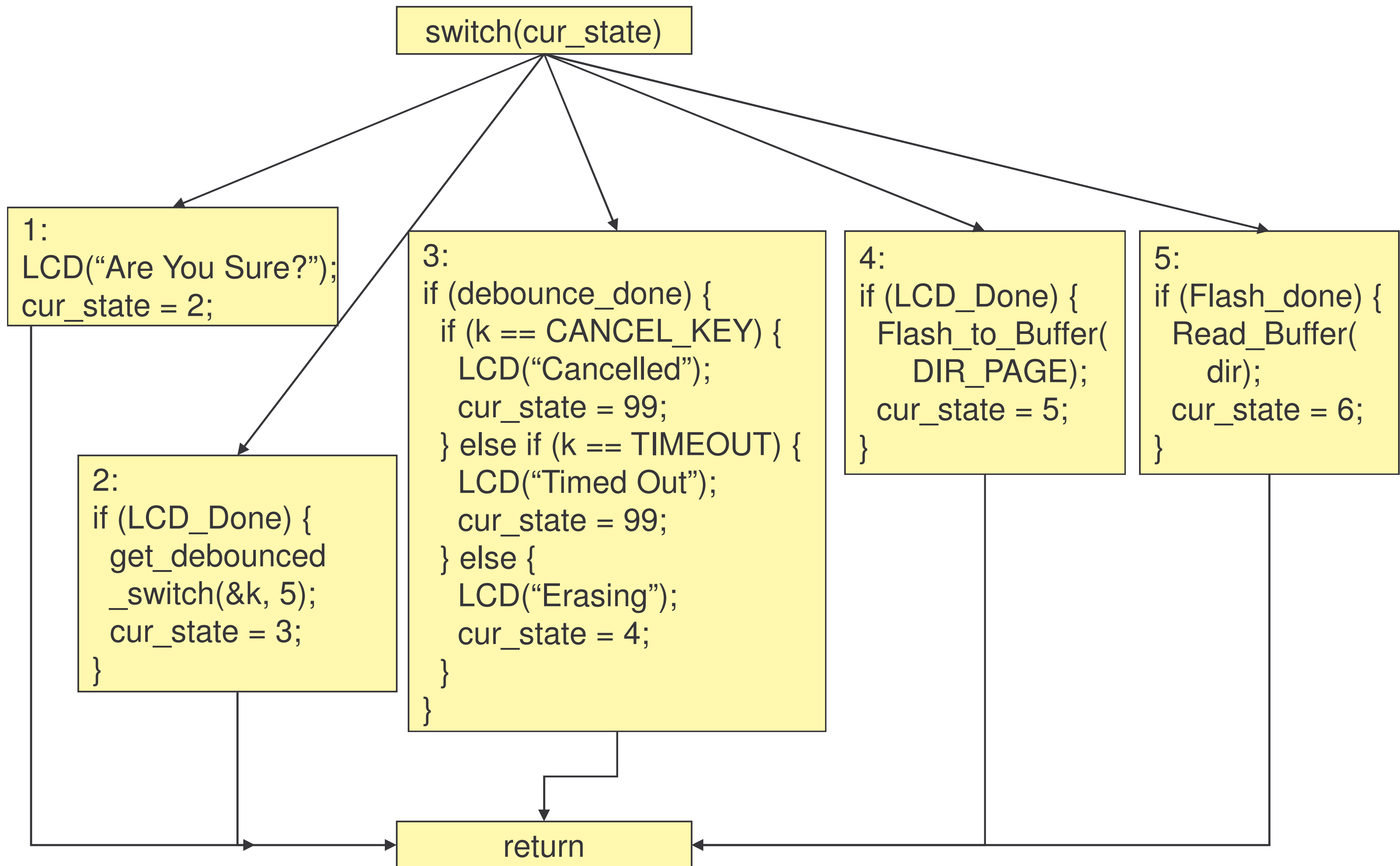
- Each piece ends where processor waits for user (e.g. debounced switch) or other devices (Flash, LCD).

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

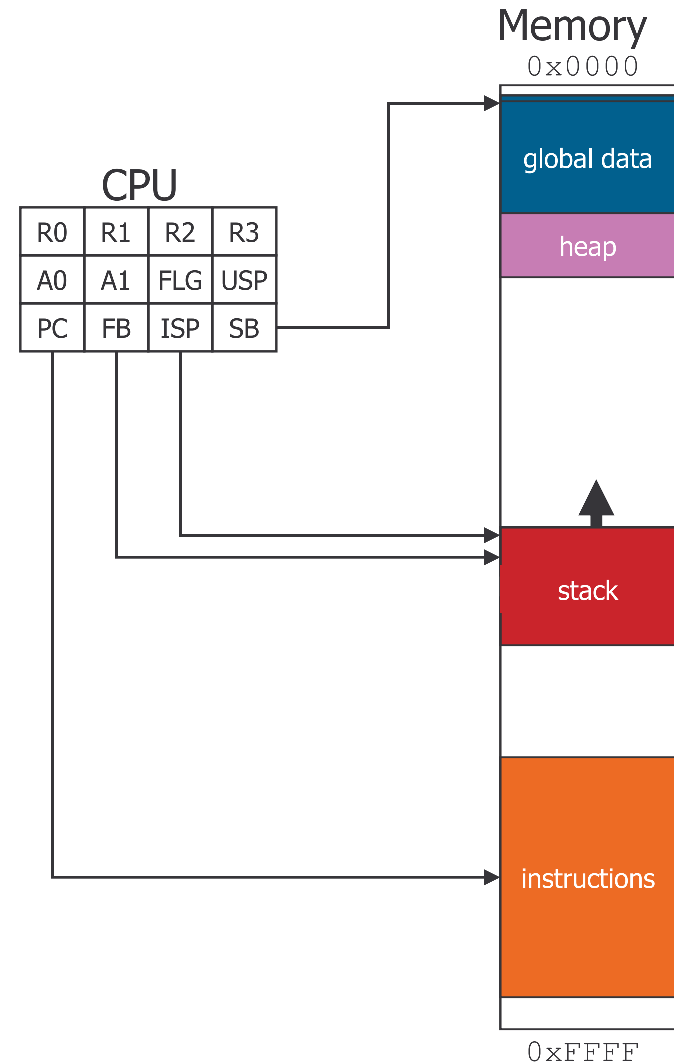
- Process management – share the processor
- Process synchronization and communication
- Memory management
- File management
- Protection
- Time management
- I/O device access

*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)*

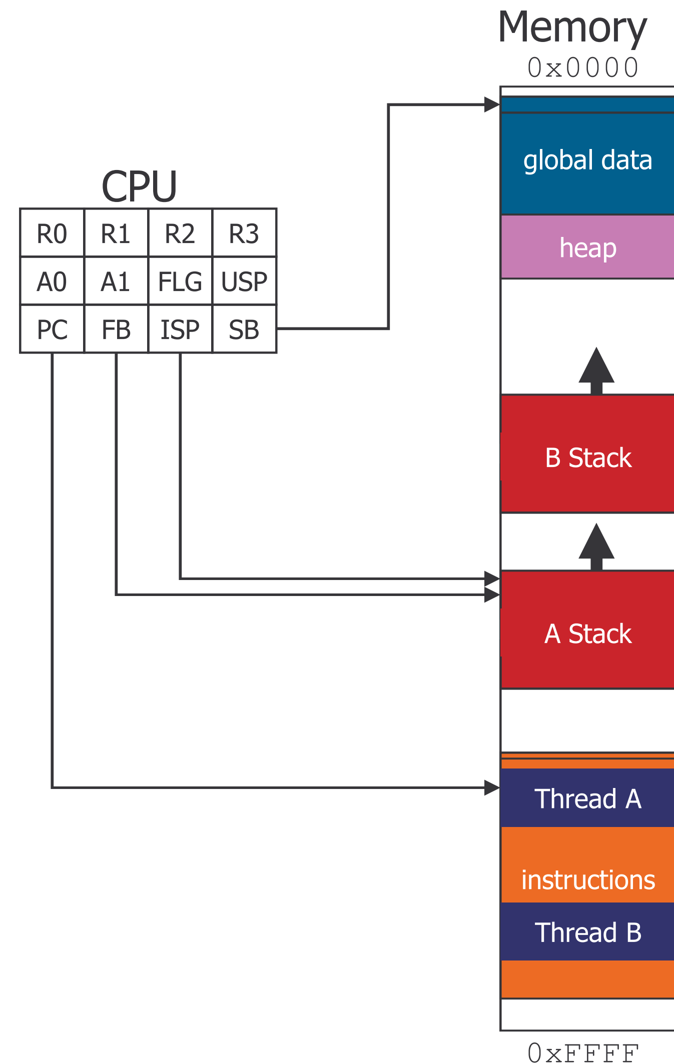
One big side effect: context switching time varies

- 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

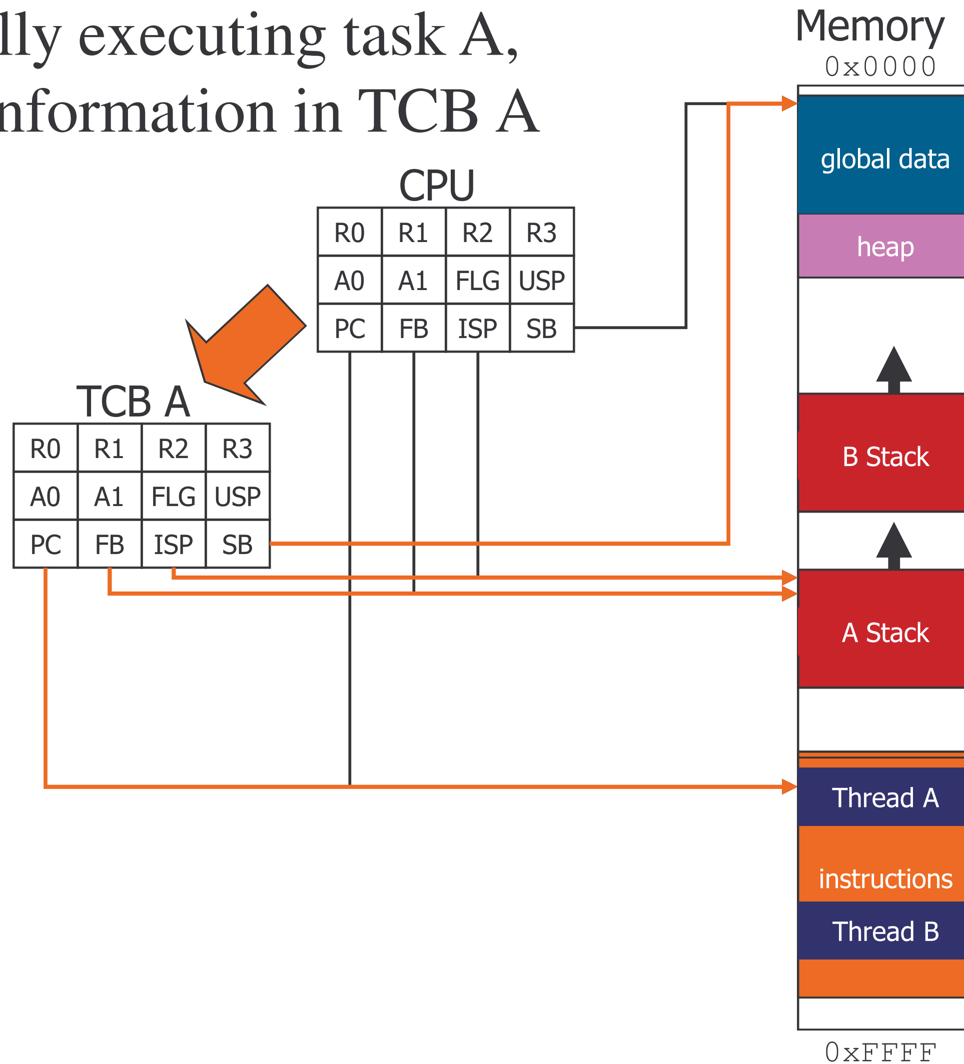
Maintaining State for Multiple Threads

- 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

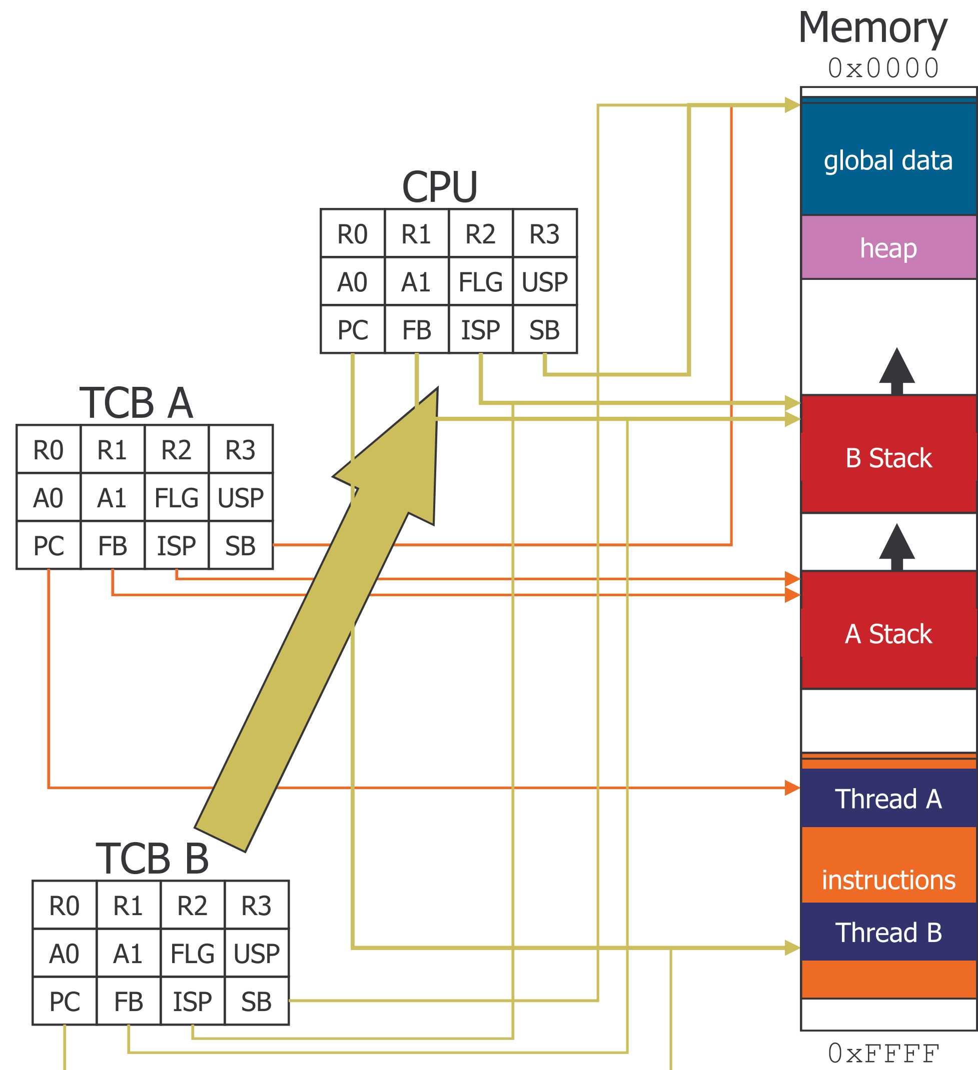


Step 2. Reload Old CPU State from TCB B

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

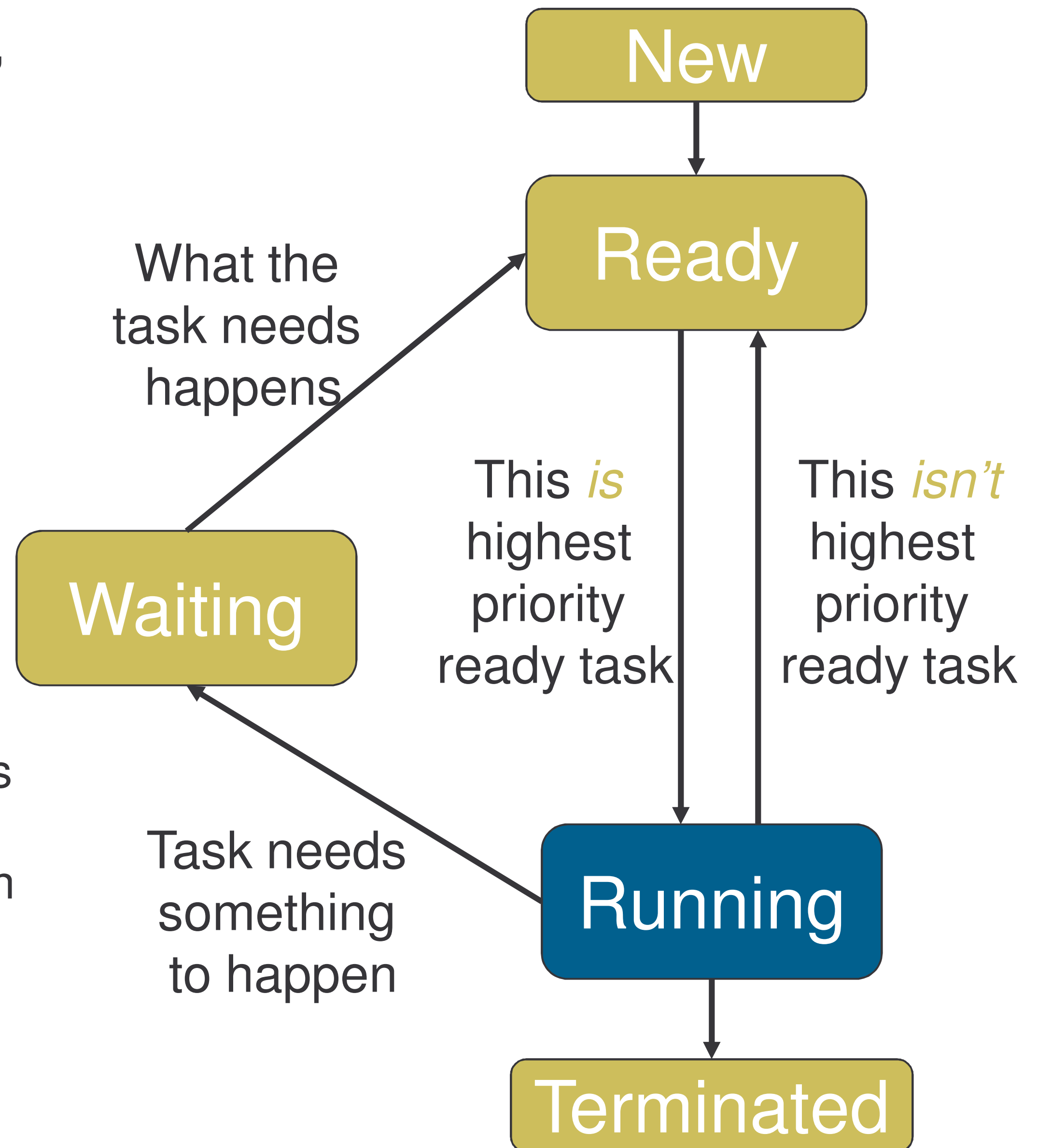


Thread States

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

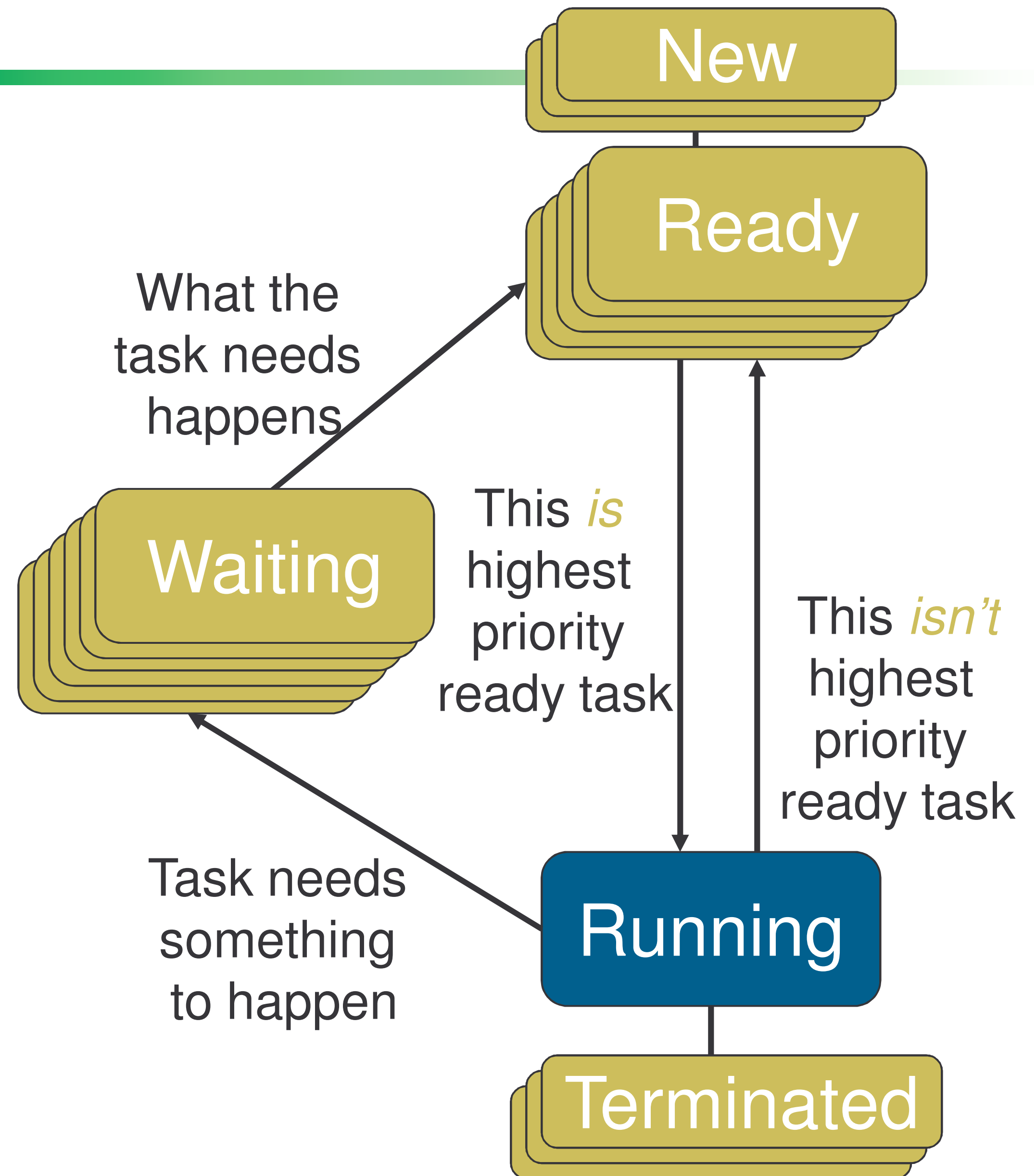


Thread Queues

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



Example Dispatcher Code

- Use interrupt to trigger a context switch
  - Timer tick
  - Break instruction
- Recall the interrupt sequence of activities
  - Clear request bit of the active interrupt
  - Save FLG in temporary register in CPU
  - Clear flags in FLG: I (interrupt enable), D (debug flag), and U (stack pointer select)
  - Push temporary register (holding old FLG) onto stack
  - Save PC (20 bits) on stack

```
typedef struct {
 int sr0, sr1, sr2,
   sr3;
 int sa0, sa1;
 int sfb, ssp;
 int spc_lm;
 char sflg_l;
 char spch_flgh;
} TCB_T;
```

new SP	PC Low	
new SP+1	PC Middle	
new SP+2	FLG Low	
new SP+3	FLG High	PC High
old SP		

Example Dispatcher Code to Save Context

```
push.w A0              ; save A0
mov.w cur_TCB, A0      ; load pointer to cur_TCB
mov.w R0,sr0[A0]       ; save R0
mov.w R1,sr1[A0]       ; save R1
mov.w R2,sr2[A0]       ; save R2
mov.w R3,sr3[A0]       ; save R3
pop.w R0               ; get old value of A0
mov.w R0,sa0[A0]       ; save it
mov.w A1,sa1[A0]       ; save A1
mov.w FB,sfb[A0]       ; save frame base register
pop.w spc_lm[A0]       ; get lower word of old PC from stack
pop.b sflg_l[A0]       ; get lower byte of flag from stack
pop.b spch_flgh[A0]    ; get upper nibbles of old PC and flag
   register
mov.w ISP, ssp[A0]     ; save stack pointer, which now has no extra
   information on it

   ; now scheduler can decide what thread to run next
```

Restore Context

```
mov.w new_TCB, A0              ; load pointer to new_TCB
mov.w sr0[A0], R0              ; restore R0
mov.w sr1[A0], R1    ; R1
mov.w sr2[A0], R2    ; R2
mov.w sr3[A0], R3    ; R3
mov.w sa1[A0], A1    ; A1
mov.w sfb[A0], FB              ; FB
mov.w ssp[A0], ISP   ; SP
push.b spch_flgh[A0]           ; high nibbles of FLG and PC
push.b sflg_l[A0]              ; low byte of FLG
push.w spc_lm[A0]              ; low and middle bytes of PC
mov.w sa0[A0], A0              ; finally restore A0
reit   ; return from interrupt. This will reload PC and FLG
   from the stack
```

Thread State Control

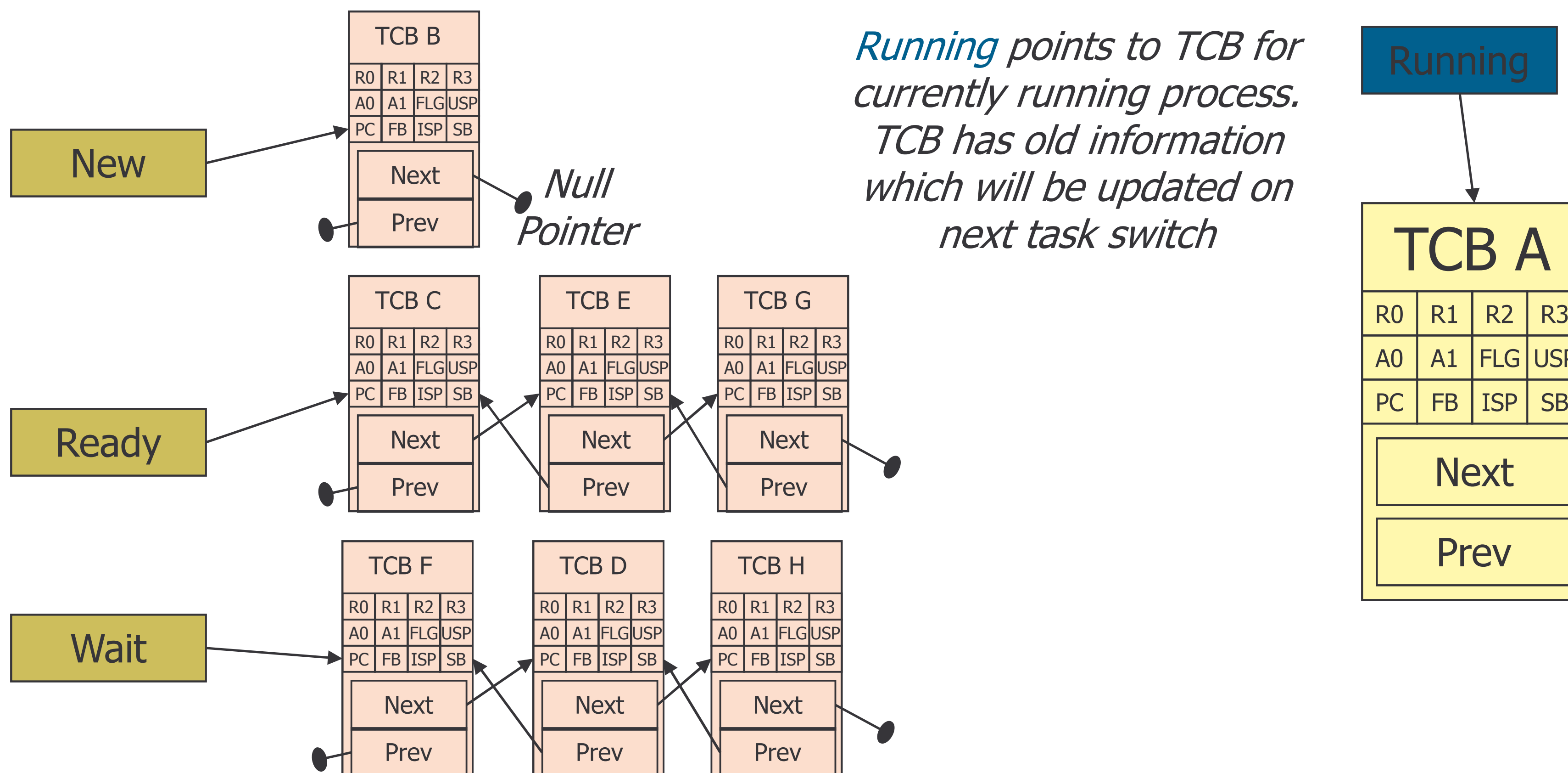
Use OS scheduler to keep track of threads and their states

- 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

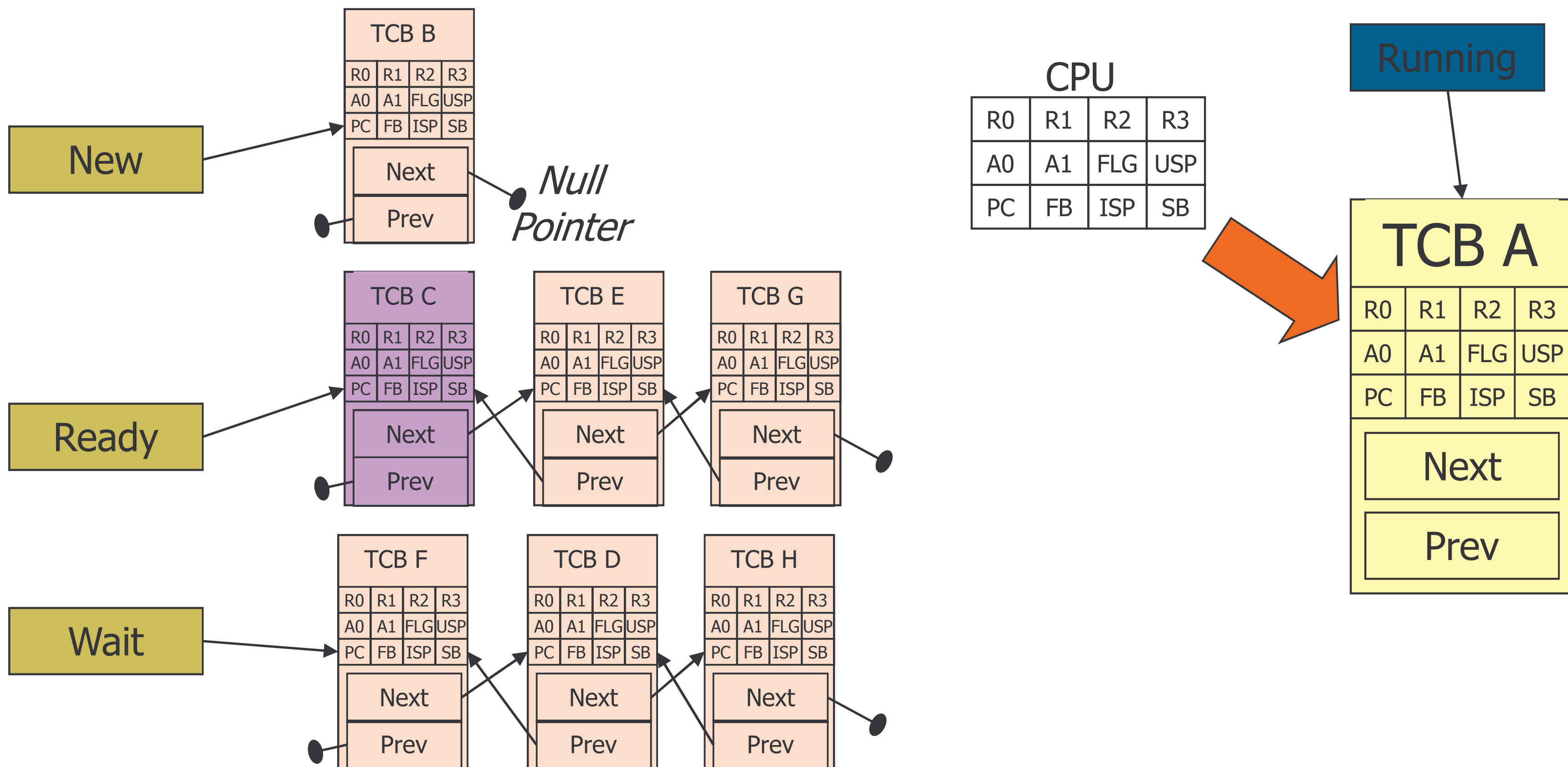
Keep track of all TCBs

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

Example: Context Switch

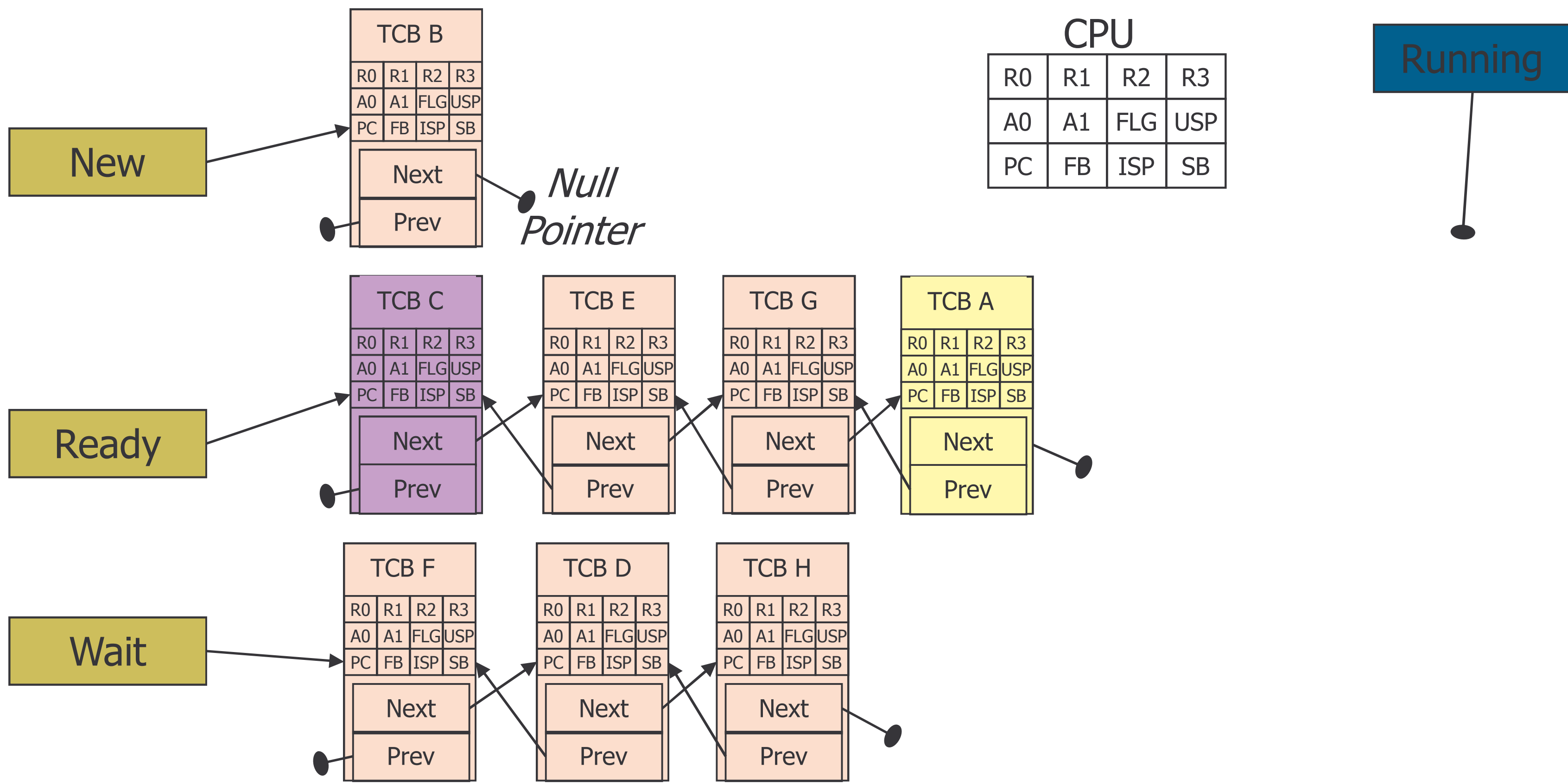
Thread A is running, and scheduler decides to run thread C instead. For example, thread A is still able to run, but has lower priority than thread C.

Start by copying CPU state into TCB A



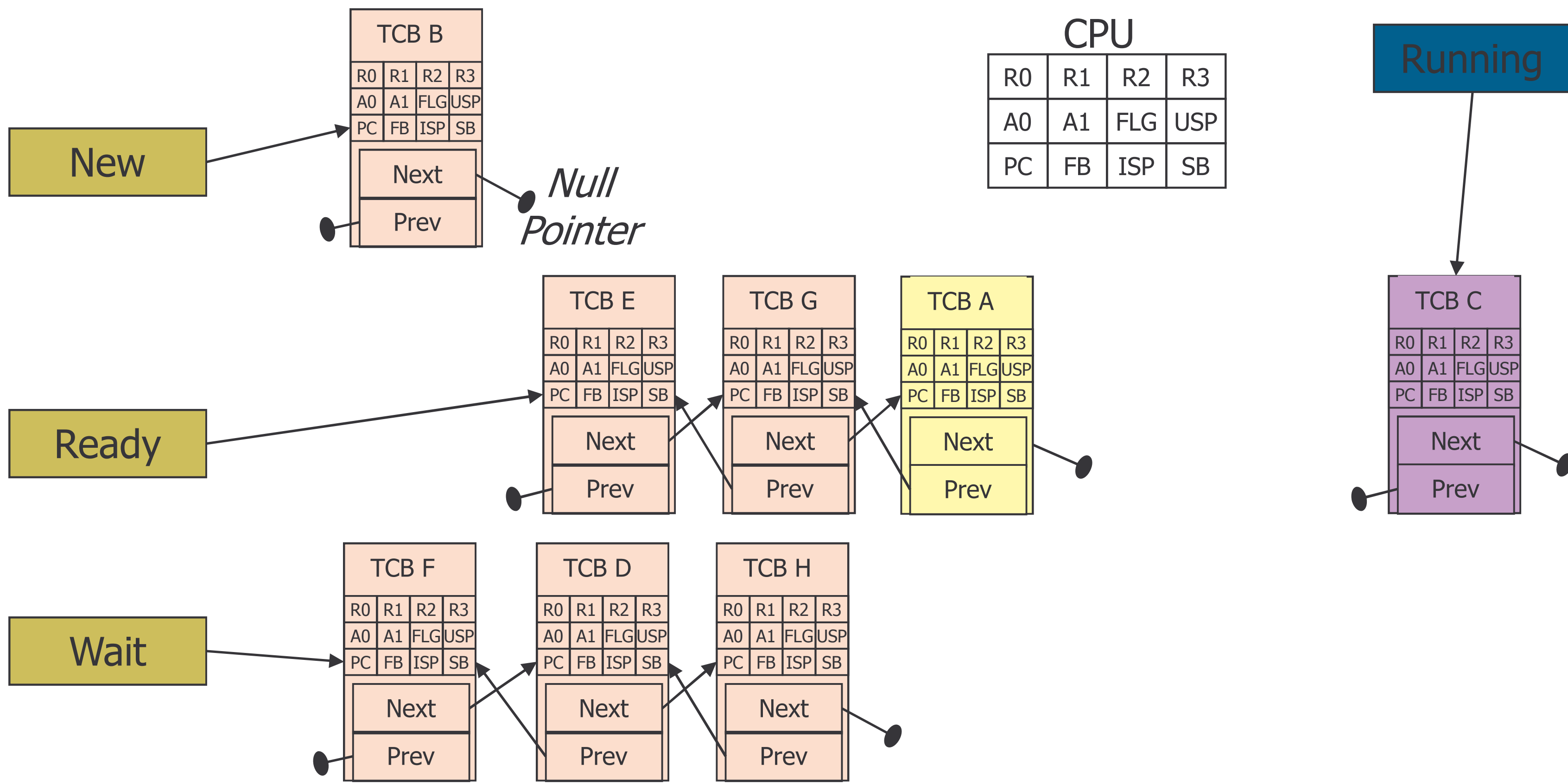
Example: Context Switch

Insert TCB A into ready queue by modifying appropriate pointers



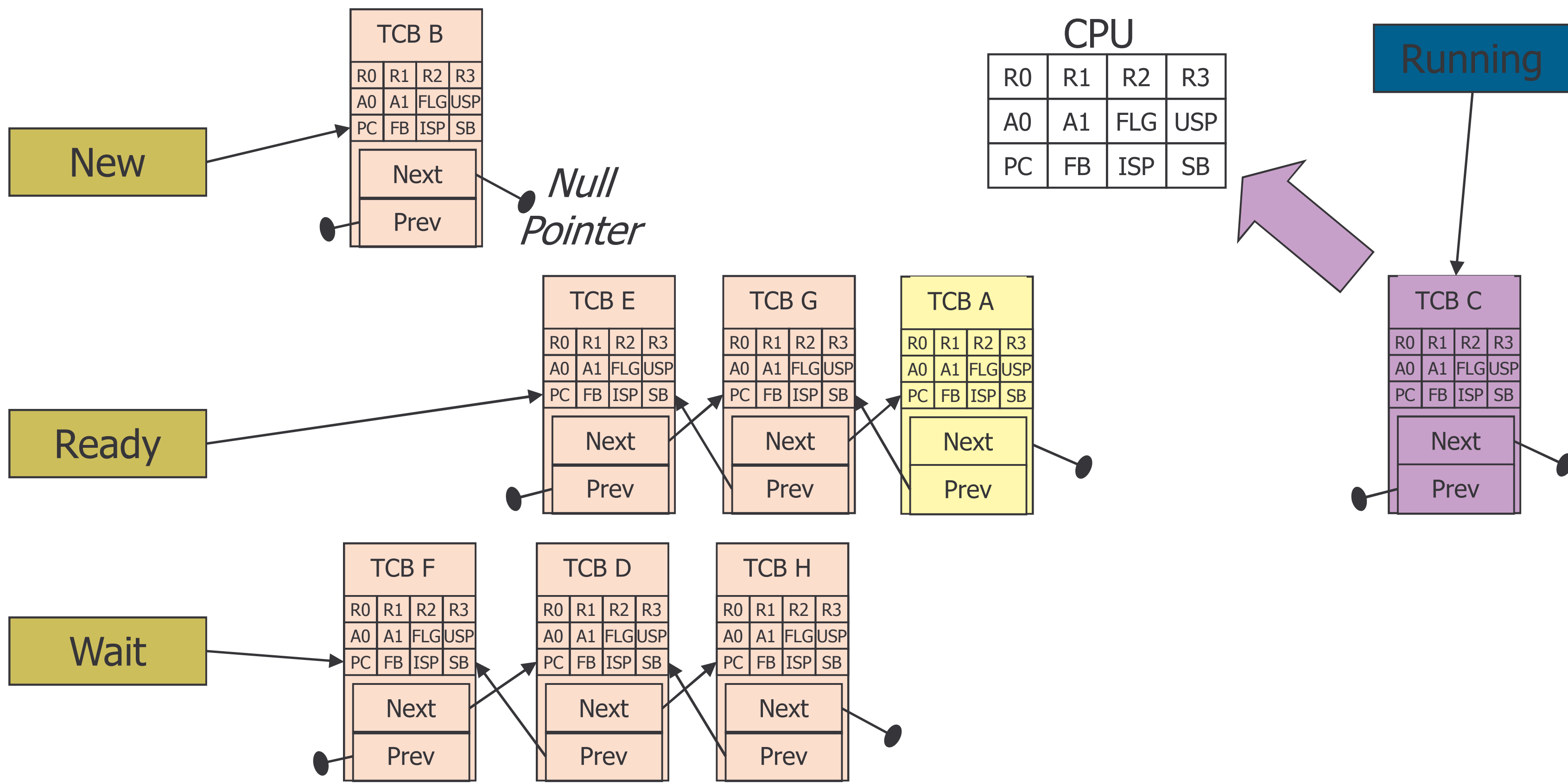
Example: Context Switch

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



uC/OS-II

Real-time kernel

- Portable, scalable, preemptive RTOS
- Ported to over 90 processors

Pronounced “microC OS two”

Written by Jean J. Labrosse of Micrium, http://ucos-ii.com

Implementation is different from material just presented for performance and feature reasons

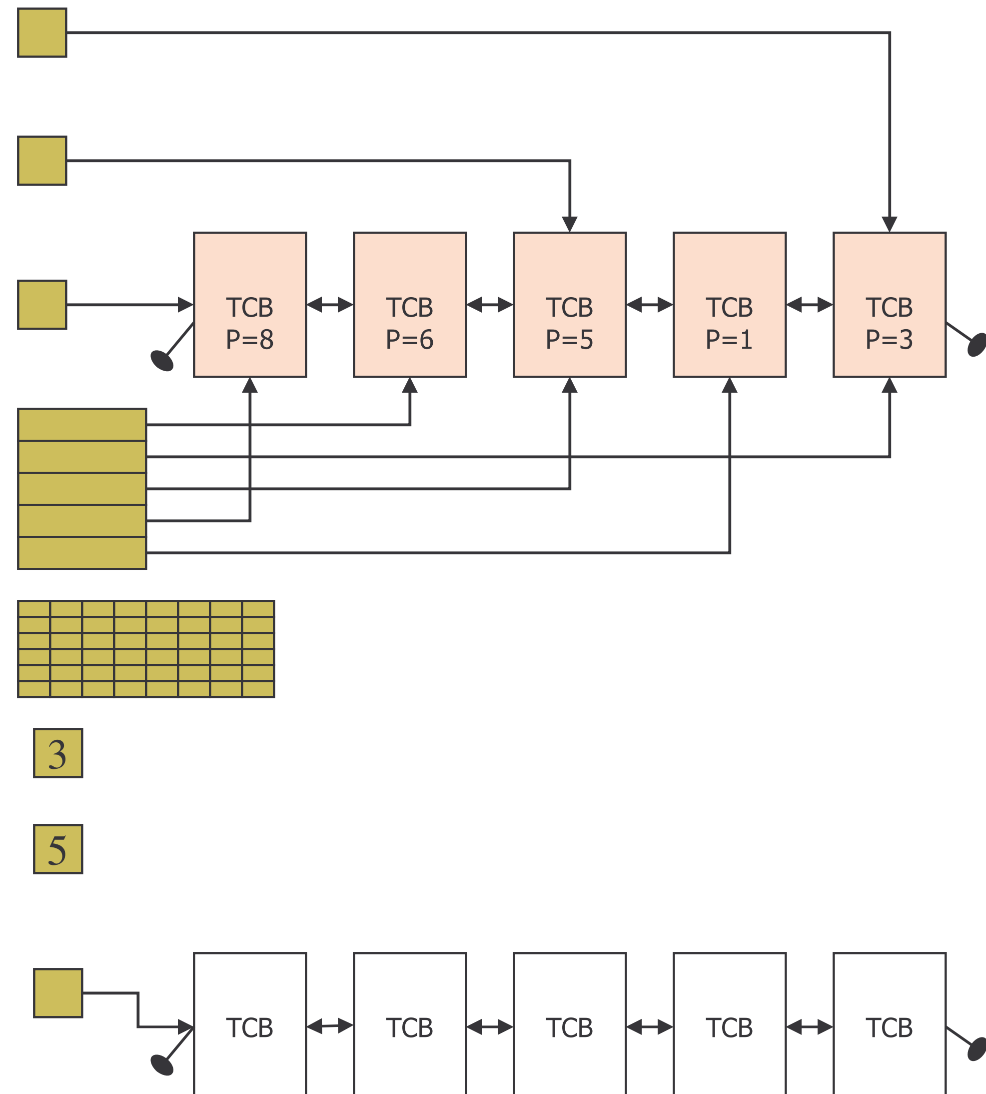
- CPU state is stored on thread’s own stack, not TCB
- TCB keeps track of boundaries of stack space
- TCB also tracks events and messages and time delays

TCB for uC/OS-II

```
typedef struct os_tcb {
   OS_STK *OSTCBStkPtr;        /* Pointer to current top of stack */
   void   *OSTCBExtPtr;        /* Pointer to user definable data for TCB
                                  extension */
   OS_STK *OSTCBStkBottom;     /* Pointer to bottom of stack - last
                                  valid address */
   INT32U OSTCBStkSize;        /* Size of task stack (in bytes) */
   INT16U OSTCBOpt;            /* Task options as passed by
                                  OSTaskCreateExt() */
   INT16U OSTCBId;             /* Task ID (0..65535) */
   struct os_tcb *OSTCBNext;   /* Pointer to next TCB in the TCB list */
   struct os_tcb *OSTCBPrev;   /* Pointer to previous TCB in list */
   OS_EVENT *OSTCBEventPtr;    /* Pointer to event control block */
   void *OSTCBMsg;             /* Message received from OSMboxPost() or
                                  OSQPost() */
   INT16U OSTCBDly;            /* Nbr ticks to delay task or, timeout
                                  waiting for event */
   INT8U OSTCBStat;            /* Task status */
   INT8U OSTCBPrio;            /* Task priority (0 == highest,
                                  63 == lowest) */
   BOOLEAN OSTCBDelReq;        /* Indicates whether a task needs to
                                  delete itself */
} OS_TCB;
```

Data Structures for uC/OS-II

- OSTCBCur - Pointer to TCB of currently running task
- OSTCBHighRdy - Pointer to highest priority TCB ready to run
- OSTCBList - Pointer to doubly linked list of TCBs
- OSTCBPrioTbl[OS_LOWEST_PRIO + 1] - Table of pointers to created TCBs, ordered by priority
- OSReadyTbl - Encoded table of tasks ready to run
- OSPrioCur – Current task priority
- OSPrioHighRdy – Priority of highest ready task
- OSTCBFreeList - List of free OS_TCBs, use for creating new tasks



Dispatcher for uC/OS-II

```
_OSCtxSw:
   PUSHM                R0,R1,R2,R3,A0,A1,SB,FB
   MOV.W                _OSTCBCur, A0
   ;OSTCBCur->OSTCBStkPtr = Stack pointer
   STC                  ISP, [A0]
   ;Call user definable OSTaskSwHook()
   JSR                  _OSTaskSwHook
   ;OSTCBCur = OSTCBHighRdy
   MOV.W                _OSTCBHighRdy, _OSTCBCur
   ;OSPrioCur = OSPrioHighRdy
   MOV.W                _OSPrioHighRdy, _OSPrioCur
   ;Stack Pointer = OSTCBHighRdy->OSTCBStkPtr
   MOV.W                _OSTCBHighRdy, A0
   LDC                  [A0], ISP
   ;Restore all processor registers from the new task's stack
   POPM         R0,R1,R2,R3,A0,A1,SB,FB
   REIT
```

Preemptive vs. Non-Preemptive

Non-preemptive kernel/cooperative multitasking

- Each task must explicitly give up control of CPU
  - E.g. return from task code, call yield function
- Asynchronous events are handled by ISRs
- ISR always returns to interrupted task
- Can use non-reentrant code (covered later)
- Task level response time can be slower as slowest task must complete
- Generally don't need semaphores

Preemptive kernel

- At each scheduling point, the highest priority task ready to run is given CPU control
- If a higher priority task becomes ready, the currently running task is suspended and moved to the ready queue
- Maximum response time is less than in non-preemptive system
- Non-reentrant code should not be used
- Shared data typically needs semaphores