Kernel Design Issues

A Possible Kernel Implementation is Used to Draw Out and Expose Issues

Winter 2014

(Example) Kernel Services

- process_create create a new process

- sema4_create create a semaphore
- sema4_wait wait on a semaphore
- sema4_signal signal a sema4

- install_ISR bind an interrupt to an ISR

(Example) Kernel Services (con't)

- sema4_wait_timed
  - wait on a semaphore with a maximum specified time limit
- driver_create
  - create an (application) driver
- driver_sleep
  - place the driver in the asleep state – may only be called by the driver to be put to sleep (i.e. self-inflicted sleep only; the driver yields the processor)

(Example) Kernel ISR Services

- driver_awake
  - awake an application driver – note: if the driver is not currently asleep, then the call has no effect

- Also need Timer ISR (internal to kernel) to support sema4_wait_timed

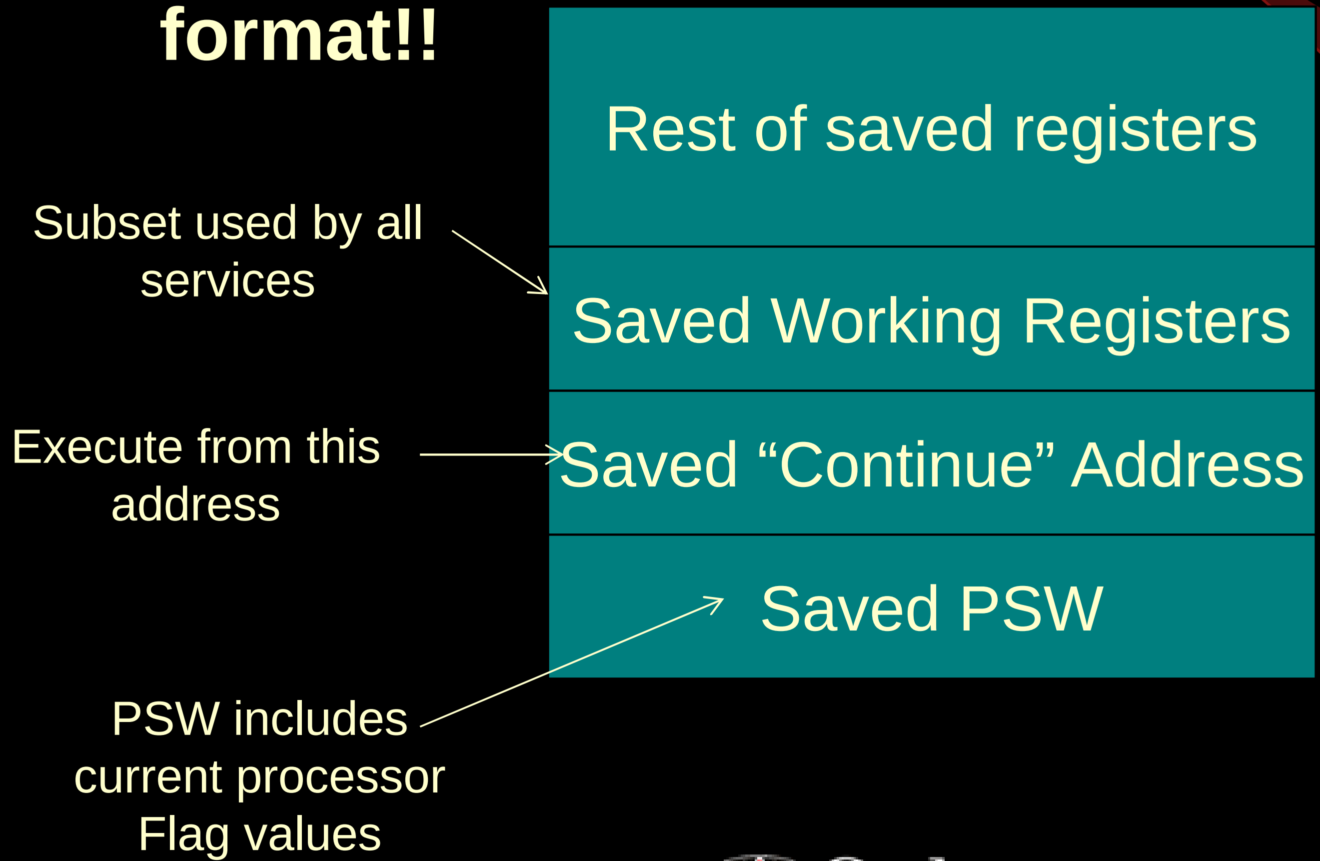
Design Issues to Expose

- Stack → Save state for Context Switch !
- S/W Interrupt to invoke Service
- ISR use of Kernel:
  - sema4_wait_timed service
  - driver_awake
- Interrupt → results in kernel activity and possibly a context switch!
- Kernel protection → kernel_busy flag
  - Protect from ISR interference

Carleton UNIVERSITY

Context Switch Design

- **What does the stack look like for a process that is not running? ← must be only one format!!**



Carleton UNIVERSITY

Service Design

- Service entry via s/w interrupt (SWI)
  - SWI pushes PSW and Return Address
  - No “function” return values in signatures
    - If a return value is needed, pass in a pointer to a variable where value is to be written
  - If process gets switched off processor, save registers on top of return address and PSW
    - will resume at instruction after SWI into service!
    - i.e. won’t resume in the kernel

→ **think about stack**!

Carleton UNIVERSITY

Invoking Service

Process Code:

…

SWI(service, params)

Next_instruction

…

service

On arrival in service:

SPreg

return address

PSW

Stack

If context switch, will eventually continue at Next_instruction

Relate to previous slide!

Carleton UNIVERSITY

Service Design (con't)

- all services end with common exit procedure which includes a context switch if needed
  - "**goto** do_exit" (goto? no return address! ☺)
- All services are protected by "kernel_busy" flag
  - ISR's can't come into kernel if kernel is busy
  - ISR's leave a message in memory if kernel is busy
  - Checking for messages is part of the (common) exit procedure for all service calls

Carleton UNIVERSITY

Globals

- Assume **kernel_busy** Boolean: true indicates that the kernel is busy executing a service call that should not be interrupted. Initial value = false.
- Assume **RTRQ** pointer to the head of the ready to run queue. Initial value = null.
  - RTRQ is priority-ordered.
- Assume **currentP** … BUT … note next slide!
- Assume **TimedQ** pointer to head of queue of processes blocked on timed_wait's.

  Initial value = null
- More to come when needed!

Note about RTRQ

- Leave process's PCB in RTRQ while running
- While a process is running (no kernel activity)

  RTRQ == currentP (point to same PCB)
- During kernel activity, a process might be inserted or removed from the RTRQ, but currentP is not modified
- After kernel activity if currentP == RTRQ

  then: the activity did not result in a need for a context switch

  otherwise: a context switch is needed! (and the RTRQ has already been adjusted)

PCB Contents

• priority	
• status	// (running, blocked, asleep)
• RTRQptr	// for linkage in the RTRQ
• SP	// stack pointer value
• sema4ptr	// for linkage in a sema4Q
• sema4ID	// sema4 that process is blocked on
• time_count	// for timeout management
• timedQprt	// for linkage in the TimedQ
• timedRtnPtr	// for returning sleep exit status

Carleton UNIVERSITY

process_create (PCB, stackptr, start_address, priority, processIDptr)

Return value!

```
{   save working registers on stack   // standard entry code!
    tempreg := SPreg          // save current SP value in a register
    // set up stack for launch
    SPreg := stackptr         // could use alternate approach and just write to memory ...
    push default PSW value (includes interrupts enabled!)
    push start_address                     // What does the stack look like for
    push default_register_values           // a process that is not running?
    *PCB.SP := SPreg          // save SP for launch
    SPreg := tempreg          // restore SP
    *PCB.status := ready
    *PCB.priority := priority
    *PCB.timedRtnPtr := null          // default = not a sleeper
    *processIDptr := PCB              // give ID of process to creator  <=
    kernel_busy := true       // up until now, nothing needed to be protected
    priorityInsertIntoRTRQ ( PCB )    // code not provided
    goto   do_exit   // non-traditional control flow! stack state?
}
```

Sema4

```
SCB:    sema4Q          // pointer to head of the sema4Q
        count           // sema4 count value

sema4_create ( SCB, count, sema4IDptr)
{   save working registers
    *SCB.sema4Q := null
    *PCB.count := count
    *sema4IDptr := SCB  // give ID of sema4 to creator
    // if there are no further linkages to internal structures ... then done!
    // no real internal "work" has been done … just leave
    restore working registers
    RTI    ← Return from interrupt:  pops return address and PSW!
}
```

sema4_wait (sema4ID)

Note: sema4 count can go below 0!

```
{   save working registers on stack   // standard entry procedure!
    kernel_busy := true
    if ( --(*sema4ID.count) < 0) {      // block the process!
            *currentP.status := blocked
            *currentP.sema4 := sema4ID
            PutInSema4Q (sema4ID, currentP)  // code not provided
            RTRQ := *currentP.RTRQptr   // remove process from RTRQ
      // note: at this point, currentP != RTRQ
    }
    goto   do_exit
}
```

sema4_wait_timed (sema4ID, **time_count**, **rtnptr**)

```
{   save working registers on stack   // standard entry procedure!
    kernel_busy := true
    if (--(*sema4ID.count) < 0) {        // block the process!
          *currentP.status := blocked
          *currentP.sema4 := sema4ID
          *currentP.timedRtnPtr := rtnPtr     // save for later!
          *currentP.time_count := time_count
          PutInSema4Q (sema4ID, currentP)  // code not provided
          PutInTimedQ (sema4ID, currentP)  // code not provided
          RTRQ := *currentP.RTRQptr   // remove process from RTRQ
    } else {  *rtnptr := OK  }
    goto   do_exit
}
```

sema4_signal (sema4ID)

```
{   save working registers on stack   // standard entry procedure!
    kernel_busy := true
    if ( (*sema4ID.count)++ < 0 ) {    // unblock a process!
            procID := DequeueFromSema4Q(sema4ID)   // code not provided
            *procID.status := ready
            if (*procID.timedRtnPtr != null) { // timed waiter!
                    *(*procID.timedRtnPtr) := OK // no timeout!
                    *procID.timedRtnPtr := null  // reset to default
                    RemoveFromTimedQ( procID ) // code not provided
            }
            priorityInsertIntoRTRQ ( procID )     // code not provided
    }
goto   do_exit
}
```

Carleton UNIVERSITY

driver_sleep

```
{   save working registers on stack
    *currentP.status := asleep
    kernel_busy := true
    RTRQ := *currentP.RTRQptr   // remove from RTRQ
    goto   do_exit
}
```

H/W ISR Design

- H/W ISRs:
  1. start in the kernel – increment counter of nested interrupts in progress (leave ints disabled)
  2. If App int: Perform SWI to App ISR (ints disabled)
     - App ISR executes RTI → returns to kernel, ints disabled
     - If Timer int: do timer processing
  3. Perform ISR exit procedure: decrement nested counter
     - Do kernel behaviour if needed

Carleton UNIVERSITY

Major Issue: Protect Kernel

- ISR can't invoke kernel activity if kernel is busy
- Solution: ISR leaves "request for work" and then finishes … must run to completion
  - Kernel services check for requests before leaving (part of do_exit)
  - If finds requests, do associated processing that would have been done by ISR if kernel was not busy
- Multiple concurrent interrupts?
  - Last one to finish does requested work

Managing Driver_Awake Calls

- Request a driver to be awoken by putting driver id in AwakeTable
  - Order in table is irrelevant
- AwakeTable: array of Driver (Process) IDs
  - Assume max size = 8 (assuming 8 int sources)
- AwakeTableIndex: index of next <u>free</u> entry in table
  - Initially: 0

Draw on board?

Carleton UNIVERSITY

driver_awake(driver process id)

```
[called by ISR!]
{  // just log the request here
   // ... process later in ISR exit code!!
   disable    //protect!
   AwakeTable[ AwakeTableIndex++ ]
        := driver process id
   RTI      // restores interrupt state & con't
}
```

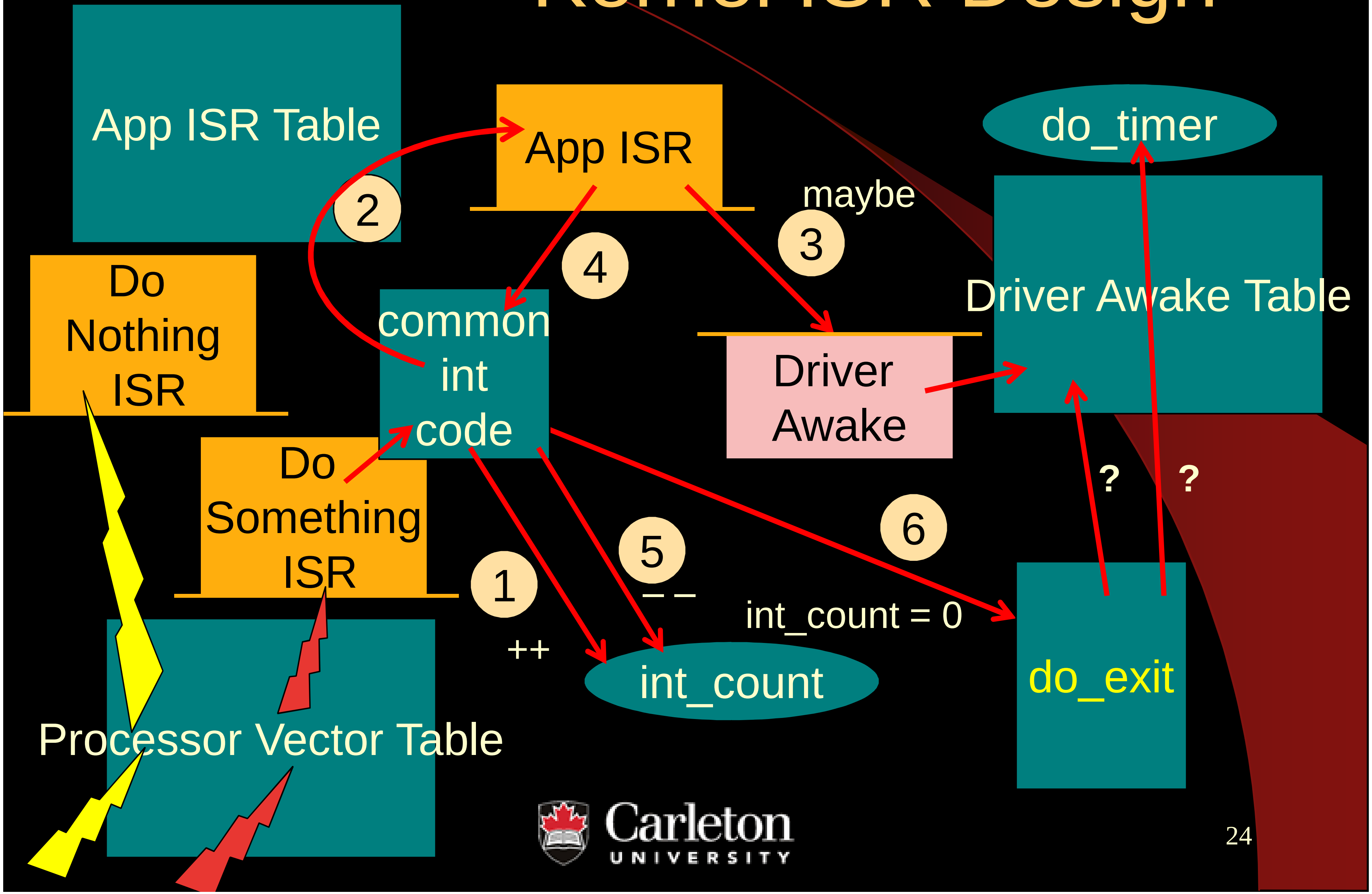
Expand on board?
Include interrupt state/PSW

Carleton UNIVERSITY

ISR Details

- **ProcessorVectorTable**: hardware vectors ISR though this table
- **Do Nothing ISR** … int not used in application
- **App_ISR_Table**: saves application ISR addresses
- **Do Something ISR** … unique ISR for each interrupt level used by App (and timer) … installed in ProcessVectorTable … will (eventually) redirect through App_ISR_Table
- **int_count**: counter of currently active (nested) interrupts
  - Initially = 0

Kernel ISR Design



Do-Nothing ISR

// at this point: interrupts are disabled

RTI   // pop return address and PSW

// PSW contents will re-enable interrupts

// at the processor

In theory, these interrupts should never occur!!

install_ISR (intNo, ISRaddress) *[kernel service]*

```
{  disable  // be safe!
   App_ISR_table[ intNo ]  := ISRaddress
   ProcessorVectorTable[ intNo ]
        := appropriate "do something" ISR (slide 28)
   RTI  // return from service, restore interrupt state
}
```

(assume timer is "installed" on interrupt 2
install INT2ISR in processor vector table)

Timer (Application) ISR

- Treat it like an application ISR, but code is in kernel ☺
- All it does is:

  do_timer = true // request work

  re-enable interrupts at the controller

  RTI   // back to kernel ISR manager
- Requested work will be done in exit code ☺

Carleton UNIVERSITY

D0-Something ISR

// one of these for each interrupt number in use by app
IntxISR: // for interrupt number X

save 1st working register (call it Reg1)

Reg1 := X // ISR specific! E.g. X = 2 for timer

goto common_ int_entry

// one common entry is shared by all Do-Something ISRs
common_int_entry:
save rest of working registers

int_count++ // log the start of a new ISR

// now do the body of the ISR:

SWI App_ISR_table[Reg1] // launch app ISR

// return from App ISR will return to this point

→ ints were disabled when SWI executed so they will be disabled here too!! (after ISR executes RTI) ☺ follow with *exit code*

Carleton UNIVERSITY

Stack State & Interrupt State?

- Go back to slides 24 and 28 and develop state over time on board
- Show tables and variable

  … sequences …

Kernel ISR *exit code*

[remember: ints are disabled here!]
if ((--int_count != 0) OR kernel_busy){
// easy case ... exit processing will be done later
restore working registers
RTI
// interrupt state will be returned to state at time of the
// interrupt by RTI
}
// at this point, int_count == 0 AND kernel_busy = false
//... do exit processing → involves kernel activity
kernel_busy := true
// do-any-pending-work … (next slide … ints are still disabled)

Carleton UNIVERSITY

do-any-pending-work

do_exit: ← **entry from kernel services too !!!!!!!!** ☺

[interrupts must be disabled here for loop test!!]

disable // if already disabled … won't matter

while ((AwakeTableIndex > 0) OR do_timer) do

{ // may have to iterate several times to finish work

enable

// kernel_busy is set, so interrupts can safely happen

do-specific-requested-work (timer or awake driver(s))

disable // and check loop again

} // ints disabled when exit loop … continue on slide 33

let's look at doing specific requested work first …

Carleton UNIVERSITY

Specific Requested Work: Awaken Sleeping Driver

```
disable
// this is a critical region
//          → AwakenTable shared with ISRs
driverID :=  AwakenTable[ --AwakenTableIndex ]
enable
*driverID.status = ready
PriorityInsertIntoRTRQ( driverID )
                                   // that's all for now ☺
```

Carleton UNIVERSITY

Specific Requested Work: Timer processing (no code?) ☹

- Set do_timer false
- Walk the queue of timed-blocked processes
- For each:
  - Decrement the time_count
  - If the count is not zero, leave process blocked
  - Otherwise: remove from the sema4Q and the TimedQ, return status := timed_out, and put process in the RTRQ with status = ready
    - Must also increment count of sema4!
  - Perform with interrupts enabled (kernel_busy!)

Carleton UNIVERSITY

Continuing after requested work

[ints still disabled!]

```
if (RTRQ != currentP) {  // context switch is needed!
  //   RTRQ already manipulated! … just save context
  save registers outside of working subset
  *currentP.SP := SPreg   // save stack pointer
   currentP := RTRQ
   SPreg := *currentP.SP
   restore registers outside of working subset
}
```

Carleton UNIVERSITY

And Finally …

// release the process:

kernel_busy := false

restore working register subset

RTI // PSW restores interrupt state

☺

Carleton UNIVERSITY