Kernels II

ENCE361 Embedded Systems 1

Course Coordinator: Ciaran Moore (ciaran.moore@Canterbury.ac.nz)
Lecturer: Le Yang (le.yang@canterbury.ac.nz)
Department of Electrical and Computer Engineering

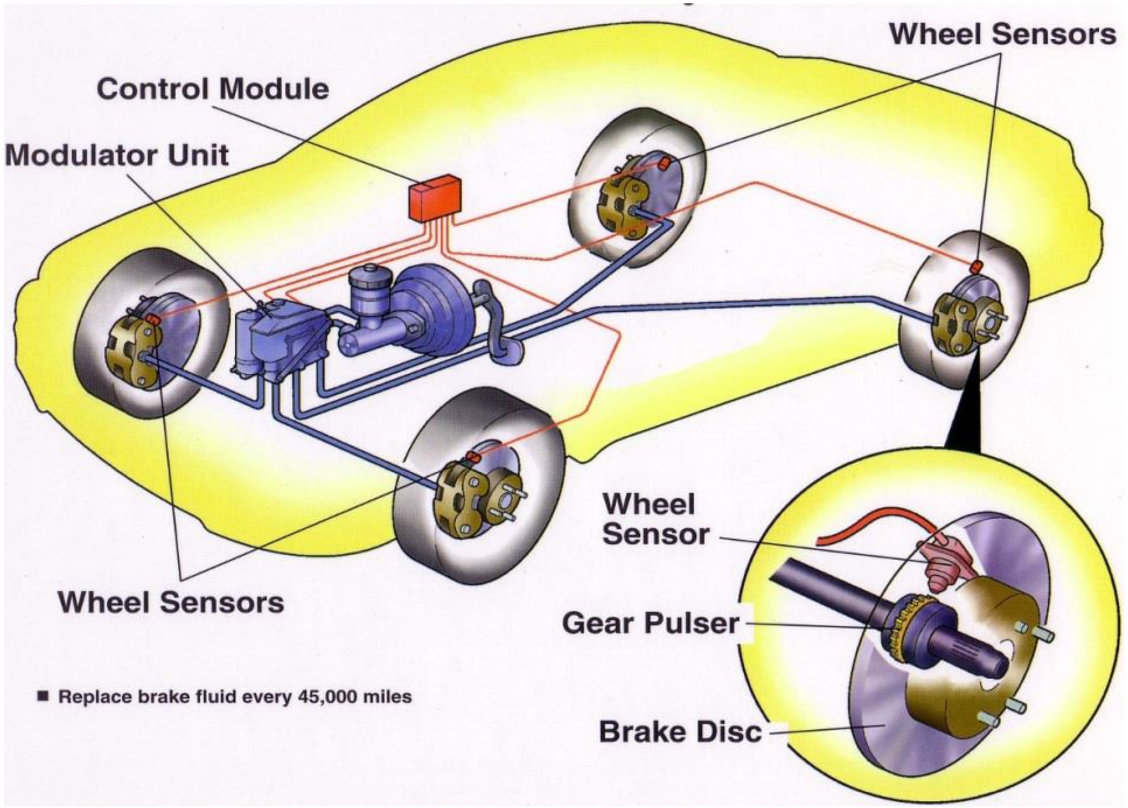
Where we’re going today

- **ABS example revisited**

- Refine kernel design

ABS Example Revisited (1)

- Accurately monitor speeds of four wheels to detect potential skid
- Interrupt-triggered scheduler
  - Interrupts for real-time clock and wheel pulses
  - Background task scheduler undecided
- Pulse rate up to 1320 pulses/s (200 km/h)
  - 1320*4 = 5280 pulses/s
- Wheel speed evaluated every 5 pulses
  - 264 speed estimates/s for each wheel



Also see Lecture 21

ABS Example Revisited (2)

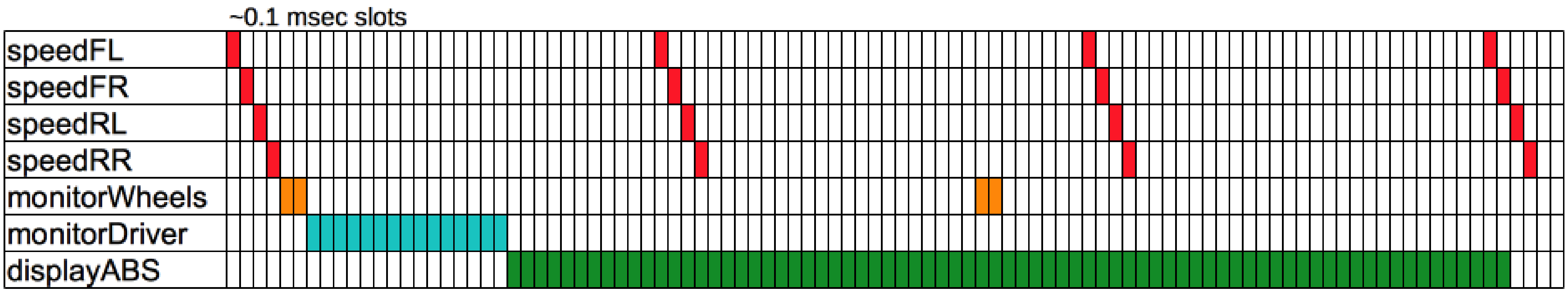
	Task	Function	Required frequency	Execution time
Foreground Tasks	Real-time clock (RTC)	ISR_SysTick ()	10^5 / s	2.5 µs
	Wheel pulses	ISR_Wheels ()	5280/ s	20 µs
Background Tasks	Task A: FLwheel speed	speedFL ()	264 / s	30 µs
	Task B: FRwheel speed	speedFR ()	264 / s	30 µs
	Task C: RLwheel speed	speedRL ()	264 / s	30 µs
	Task D: RRwheel speed	speedRR ()	264 / s	30 µs
	Task E: Monitor relative speeds	monitorWheels ()	200 / s	250 µs
	Task F: Monitor driver actions	monitorDriver ()	50 / s	1.5 ms
	Task G: Activate ABS	activateABS ()	Infrequent	5 ms
	Task H: Display status	displayABS ()	20 / s	7.5 ms

ABS Example Revisited (3)

```
void main (void)
{
// Initialise
. . . .
// Start time-triggered kernel
   while (1)
   {
       if (flagFL) { speedFL (); flagFL = 0; }                    // 30 µs,  264/s = True frequency
       if (flagFR) { speedFR (); flagFR = 0; }
       if (flagRL) { speedRL (); flagRL = 0; }
       if (flagRR) { speedRR (); flagRR = 0; }
       if (monitorWheels ())                                      // 250 µs,  200/s = True frequency
           activateABS ();
       if (monitorDriverFlag) { monitorDriver (); monitorDriverFlag = 0;}     // 1.5 ms,   50/s = True frequency
       if (displayABSFlag)   {  displayABS ();   displayABSFlag = 0;}         // 7.5 ms,   20/s = True frequency
   }
}
```

Will this actually work?
If yes, why?
If no, why not?

Timing Analysis



- speedFL (), speedFR (), speedRL () and speedRR () need to be executed at a frequency of 264 Hz (i.e., once every 3.78 ms)

- Possibly within a single kernel loop
  - Speed estimation for 4 wheels takes 4∗30 us = 0.12 ms
  - Relative speed monitoring using monitorWheels () takes 250 us = 0.25 ms
  - Driver action monitoring using monitorDriver () takes 1.5ms
  - BUT status display using displayABS () needs 7.5 ms > 3.78 ms

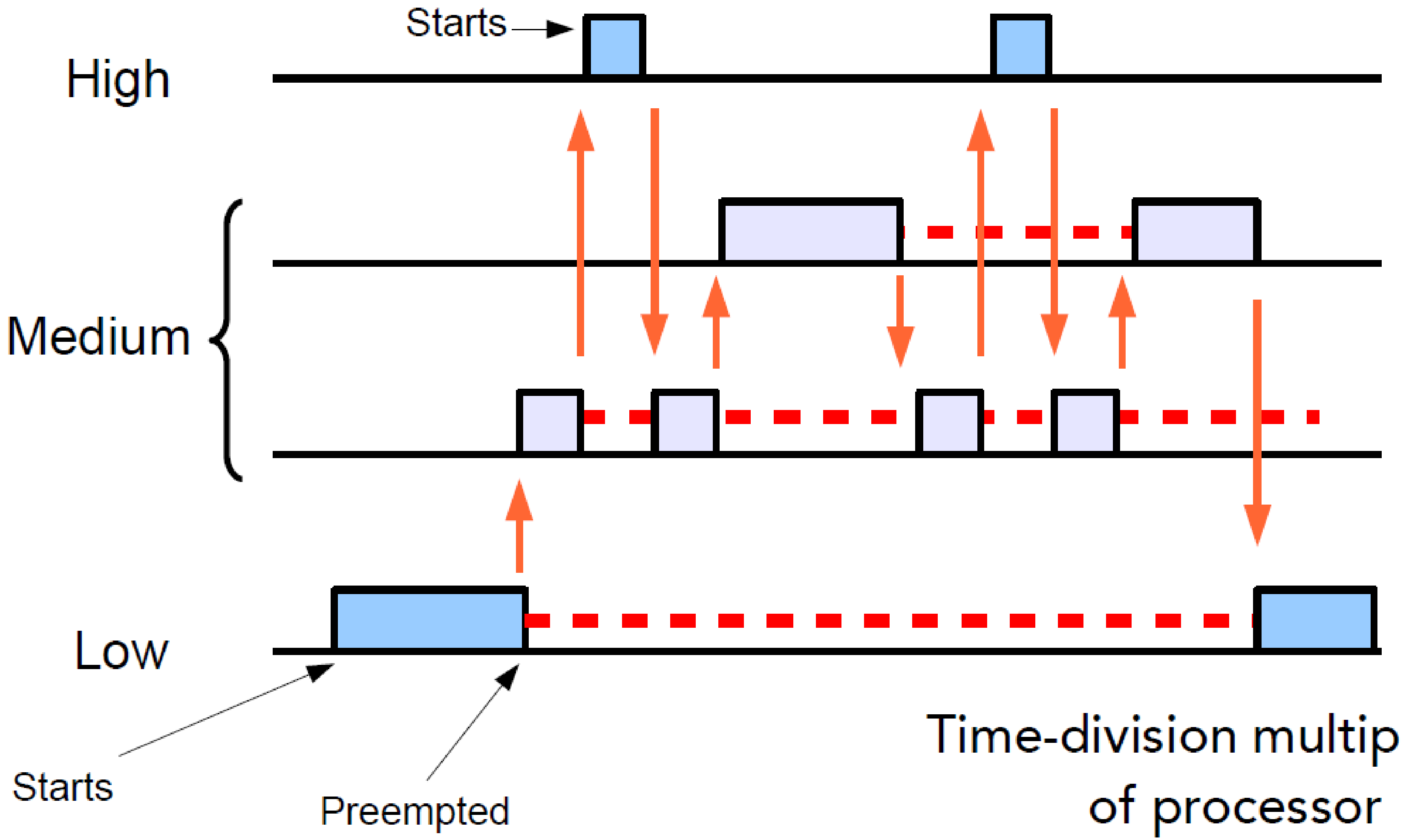
Tasks that can be completed between two speed estimations

Cooperative Multitasking

```
void TaskA() {
    static TaskAState_t state = START;
    switch (state)
    {
        case START:
        // Do some stuff
        state = STATE_1;
        break;

        case STATE_1:
        // Do some more stuff
        state = STATE_2;
        break;
        ...
    }
    return;
}
```

Split the task displayABS ()
into smaller chunks

Preemptive Multitasking



Time-division multiplexing
of processor

Cooperative vs. Preemptive Multitasking

- Cooperative multitasking
  - Harder to control timing
  - Simpler to implement and with fewer concurrency bugs
  - Tends to be ‘homebrewed’
    - Although see TinyOS, Quantum …
- Preemptive multitasking
  - Good control of timing (for foreground tasks)
  - More complex to implement with possibly more bugs (shared data problem)
  - Usually use off-the-shelf systems (FreeRTOS, VxWorks …)

Where we’re going today

- ABS example revisited

- **Refine kernel design**

Refine Kernel Design

- Application-specific code tied to kernels
- Better kernel design
  - Clarify concepts and abstractions
  - Separate concerns
  - Simplify scheduler modifications

```
while (1)
 {
      if (flagFL)
        {
          speedFL ();
          flagFL = 0;
        }
      if (flagFR)
        {
          speedFR ();
          flagFR = 0;
        }
       .
       .
       .
 }
```

protoKernel By Prof. Bones (1)

```
// pKinit: Initialise protoKernel for up to 'maxTasks' tasks.
//             Sets the period for SysTick interrupt in ns.
//             SysTick is used to time the kernel services.
void pK_init (uint8_t maxTasks, uint32_t tickPeriod);

// pK_register_task: Register a task with the protoKernel.
//             A pointer to the function which executes the task and the 'priority', used to
//             order tasks in the round-robin scheduling (0 is highest priority), are passed.
// Tasks are by default in the ready state. Returns a unique ID.
taskId_t pK_register_task (void (*taskEnter) (void), uint8_t priority);

// pK_start: Starts the round-robin scheduling of the set of registered and 'ready' tasks.
void pK_start (void);

// pK_unregister_task: Removes the nominated task from protoKernel. The task can be
subsequently re-registered.
void pK_unregister_task (taskId_t taskId);
```

protoKernel By Prof. Bones (2)

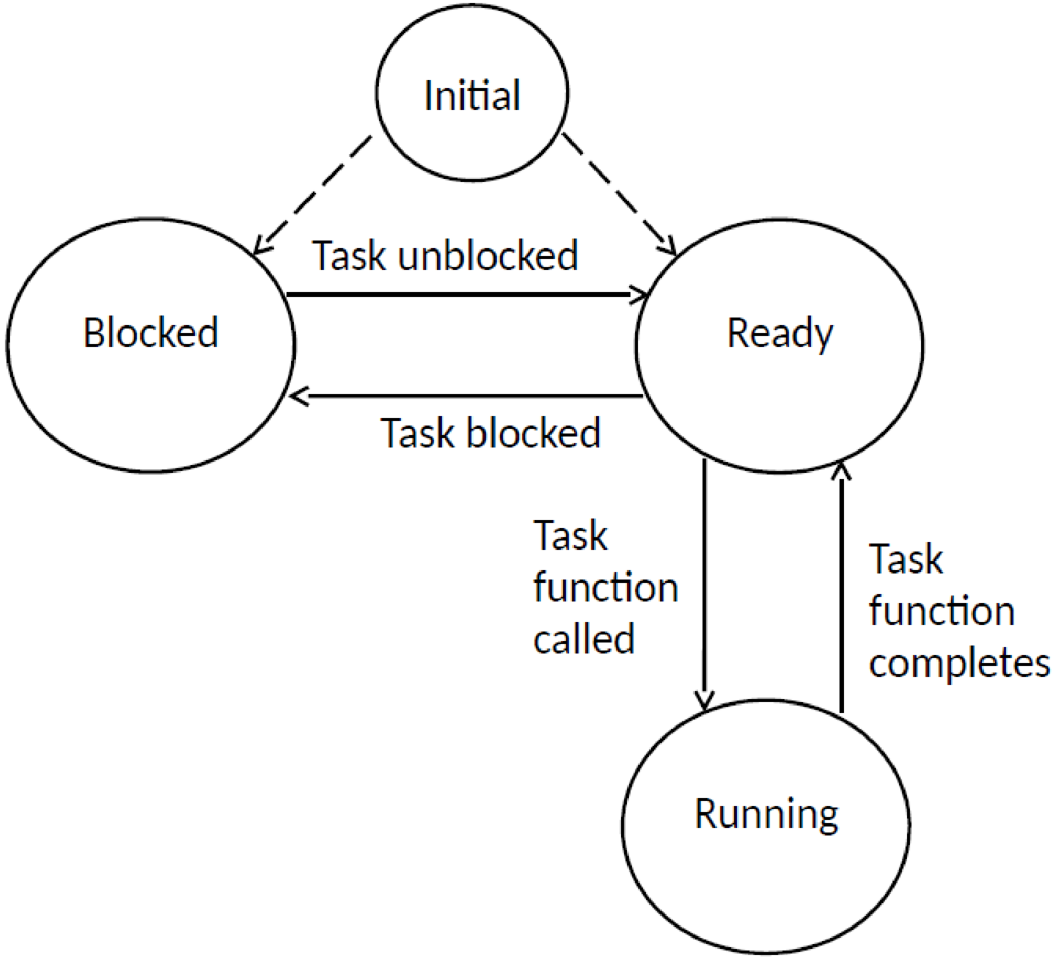
```
// pK_notify_task: Switches the task with ID 'taskId' to 'ready' so that
//                 it will be executed within the round robin.
void pK_notify_task (taskId_t taskId);

// pK_block_task: Switches the task with ID 'taskId' to 'blocked' so that
//                it will not be executed within the round robin.
void pK_block_task (taskId_t taskId);
```

Realization of Round-Robin Scheduler

```
// Task state and handler entry point
typedef struct {
      taskId_t id;            // Unique task ID
      bool ready;             // Ready/Blocked?
      uint8_t priority;       // Priority level
      void (*run) (void);     // Task pointer
} Task_t;
. . .
void pK_start (void) {
      while (true) {
            for (uint8_t i = 0; i < scheduler_max_tasks; i++) {
                  if (scheduled_task[i].ready) {
                        scheduled_task[i].ready = false;
                        // Call task handler
                        scheduled_task[i].run();
                  }
            }
      }
}
```

Round-Robin Scheduler with Ready/Blocked task status

Finite-State Machine Representation



Realization of Priority-based Scheduler

```
// On each iteration, execute the highest priority task marked as `ready`.
// Assume tasks are entered into scheduled_task[] in priority order.
void pK_start (void)
{
    while (true) {
        uint8_t i = 0;
        bool handled = false;
        while (i < scheduler_max_tasks && !handled) {
            if (scheduled_task[i].ready) {
                scheduled_task[i].ready = false;
                handled = true;          // End of the current iteration.
                // Call task handler
                scheduled_task[i].run();
            }
            i++;
        }
    }
}
```

Possible Use: ABS Example (1)

```
void main (void)
{
.
.
// Initialise the protoKernel with a tick period of 10 us
pK_init (8, PK_TICK_10_USEC);

// Register the tasks
iDspeedFL = pK_register_task (speedFL, 0);
iDspeedFR = pK_register_task (speedFR, 0);
.
.
// ABS task waits for a skid to occur
pK_block_task (iDactivateABS);

// Start the kernel (forever)
pK_start ();
}
```


From Table 2-8 on Page 103 in

Possible Use: ABS Example (2)

```
// Interrupt on rising edge on any pin; use NUM_WHEELS sRTC software timers
// (starting from FL_TIMER) to time the period in ticks for each wheel and
// load into NUM_WHEELS circular buffers.
void WheelsIntHandler (void) {
    static uint32_t ulPortStatus;
    // Read the interrupt status for the pins and clear the interrupt
    ulPortStatus = GPIOPinIntStatus(WHEEL_PORT, 0);
    GPIOPinIntClear (WHEEL_PORT, PIN_FL | PIN_FR | PIN_RL | PIN_RR);

    // Read and restart the software timer for any pin which has changed 0->1
    if (ulPortStatus & (uint32_t)PIN_FL) {
        circBuf_write (&wheelBuf[FL], readSTimer (FL_TIMER + FL));
        startSTimer (FL_TIMER + FL);
        if (++pulseCount[FL] >= WHEEL_PROCESSING_THRESHOLD ) {
            pulseCount[FL] = 0;
            pK_notify_task (iDspeedFL);
        }
    }
    // Same for other wheels
}
```