CG2271

Real-Time Operating Systems

Revision Lecture

colintan@nus.edu.sg

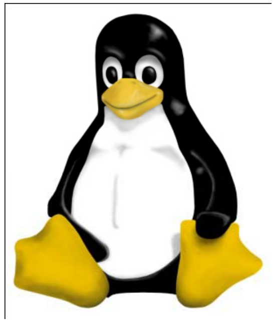
NUS National University of Singapore

School of Computing

Real Time Operating Systems

DIFFERENCES BETWEEN OS AND RTOS

What are Operating Systems?



- **An "operating system" is a suite (i.e. a collection) of specialized "system" software that:**
  - Gives you access to the hardware devices like disk drives, printers, keyboards and monitors.
  - Controls and allocate system resources like memory and processor time.
  - Gives you the tools to customize your and tune your system.
- **Examples include LINUX, MacOS (a variant of LINUX), Windows 7.**

What are Real-Time Operating Systems?

- **Real-time operating systems are OSes that are designed with real-time issues in mind:**
  - Applications are time-critical so schedulers guarantee upper-bounds on execution time.
  - Strong emphasis on reliability.
  - Memory and CPU power are limited, so RTOSs can be made very small and compact – highly customizable.
  - Emphasis on reading sensors and operating actuators rather than interacting with the user.
- **Examples:**
  - RT Linux
  - MicroC/OS-II
  - FreeRTOS

Real Time Operating Systems

HARDWARE PROGRAMMING: TYPES OF HARDWARE I/O

Communicating with External Devices

- **For a processor to do anything useful, it must be able to communicate with the outside world.**
  - Some examples of external devices include:
    - ✓**Input devices:**
      - *–Temperature sensors, light sensors, skid sensors, pitot tubes + static ports, etc etc*
    - ✓**Output devices:**
      - *–piezo-electric alarms, LCD/LED displays, actuators, servos, etc etc.*

NUS National University of Singapore School of Computing

Desktops and Servers Memory Mapped I/O

- **Memory Mapped I/O example:**
  - Read from temperature sensors 0 and 1 and write to memory locations 400 and 401

```
–LI R0, 8192 ; Address of sensor 0
–LW R0, (R0)        ; Read from temp sensor 0
–LI R1, 400         ; Address to write to
–SW R0, (R1) ; Write temperature
–LI R0, 8193 ; Address of sensor 1
–LW R0, (R0) ; Read from temp sensor 1
–ADDI R1, R1, 1     ; Increment R1 to 401
–SW R0, (R1) ; Write temperature to 401
```

Desktops and Servers
Direct Mapped I/O

- **Direct Mapped I/O example:**
  - Read from temperature sensors at port numbers 8192 and 8193
  - Write to memory locations 8192 and 8193

```
» IN R0, 8192      ; Read sensor 1
» LI R1, 8192
» SW R0, (R1)      ; Write to memory
»                  location 8192
» ADDI R1, R1, 1 ; Inc. to location 8193
» IN R0, 8193      ; Read sensor 2
» SW R0, (R1)      ; Write to memory location
»                  ; 8193
```

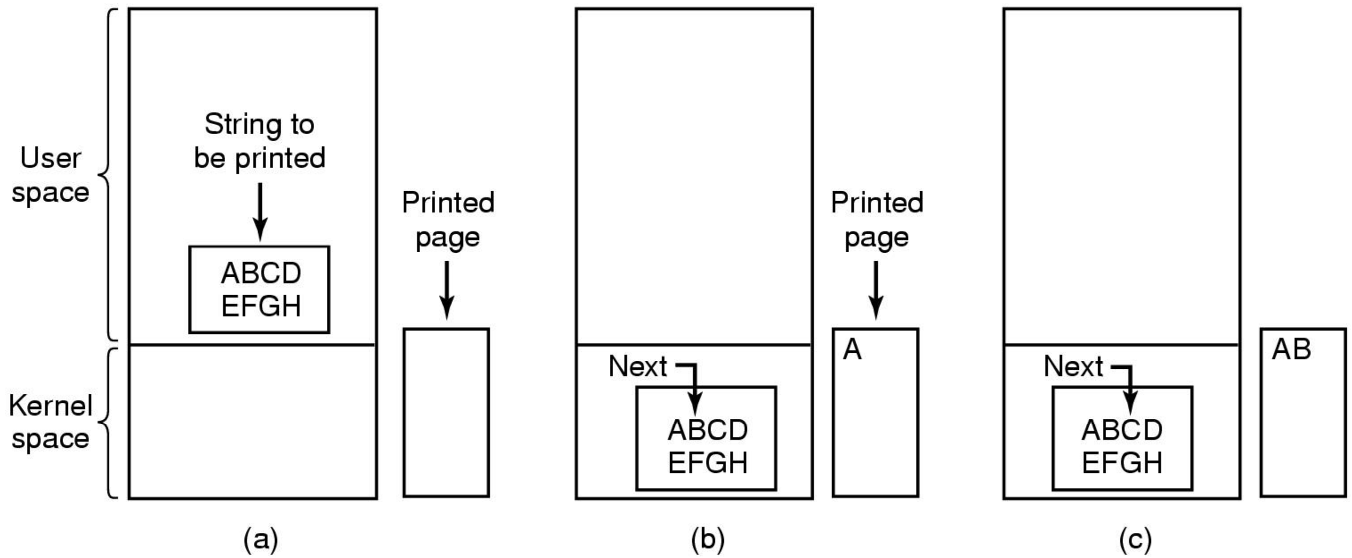
Microcontrollers
Register Mapped I/O

- **Register-mapped I/O is a variation of memory-mapped I/O.**
  - In memory mapped I/O, any location that is not used to store data can be used for I/O.
    - ✓**The corresponding device is activated when the “decoder” circuit matches the address on the address bus with the device’s ID.**
  - In register-mapped I/O, the memory locations that are used for I/O is fixed.
    - ✓**These fixed locations are typically called “registers”, which is different from CPU registers that you are used to.**
    - ✓**Suitable for microcontrollers as the set of peripherals is usually fixed.**

Real Time Operating Systems

HARDWARE PROGRAMMING: HOW TO PROGRAM I/O

I/O Programming
Programmed I/O




```
copy_from_user(buffer, p, count);                  /* p is the kernel bufer */
for (i = 0; i < count; i++) {                      /* loop on every character */
      while (*printer_status_reg != READY) ;       /* loop until ready */
      *printer_data_register = p[i];               /* output one character */
}
return_to_user( );
```

I/O Programming
Interrupt I/O

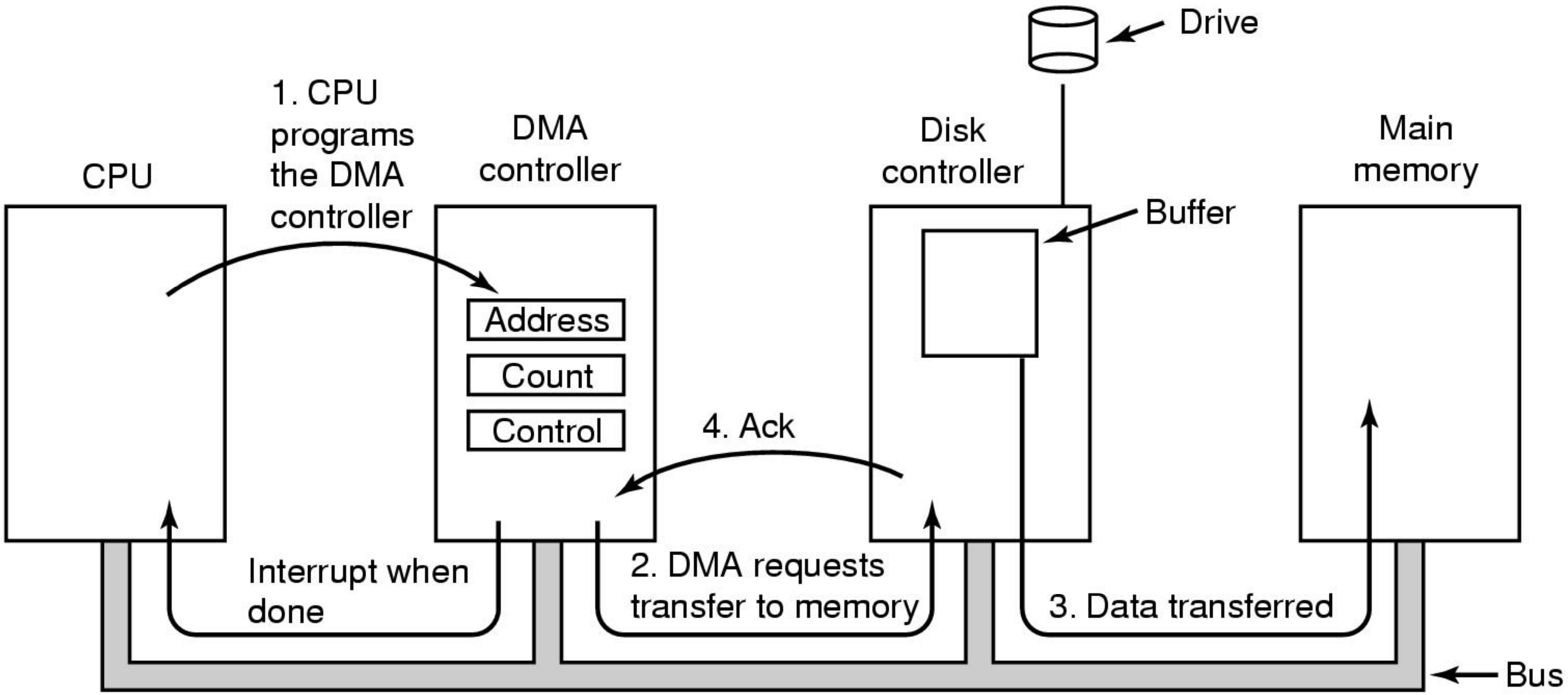
```
copy_from_user(buffer, p, count);
enable_interrupts( );
while (*printer_status_reg != READY) ;
*printer_data_register = p[0];
scheduler( );
```

(a)

```
if (count == 0) {
      unblock_user( );
} else {
      *printer_data_register = p[i];
      count = count – 1;
       i = i + 1;
}
acknowledge_interrupt( );
return_from_interrupt( );
```

(b)

I/O Programming
Direct Memory Access.



I/O Programming
Direct Memory Access.

- **In our printer example:**
  - The user process makes an OS call with a pointer to the text buffer, and the number of characters to print.
  - The OS copies the text into its own buffers, then sets up the DMA transfer and blocks the calling process.
  - The OS initiates a DMA transfer, then hands control to another process.
  - When the DMAC is done, it interrupts the OS. The OS moves the calling process into the READY state.

I/O Programming
Direct Memory Access.

```
copy_from_user(buffer, p, count);
set_up_DMA_controller( );
scheduler( );
```

(a)

```
acknowledge_interrupt( );
unblock_user( );
return_from_interrupt( );
```

(b)

Real Time Operating Systems

SOFTWARE ARCHITECTURES

Real-Time Software Architectures
Round Robin

- **Round Robin architecture works particularly well when:**
  - There are few devices to service.
  - There is no long complicated processing to be done with the data read in.
  - There are no tight time requirements.
    - ✓**E.g. in a multimeter, if a user turns a dial, it is unlikely that he will notice that the loop is currently reading the voltage probes, before it comes round and reads the new dial settings.**

Real-Time Software Architectures
Round Robin

- **However it fails when:**
  - Any device requires attention in less time than it takes for the CPU to go around the loop.
  - If there is lengthy processing to do. In the multimeter example in the book, if it takes 3 seconds to process a voltage reading:
    - ✓**It may take as long as 3 seconds for the multimeter to respond to the user changing the dial setting.**

Real-Time Software Architectures
Round Robin

- **This architecture is also fragile:**
  - If we added as little as just one more device, the performance may no longer be acceptable.
    - ✓**For example, if the new device takes a long time to get a reading.**

Real-Time Software Architectures
Round Robin with Interrupts

- **In this architecture:**
  - When a sensor has data to send to the CPU, it will interrupt the CPU.
  - The interrupt handler reads the device and sets a flag.
  - The main loop polls these flags, and takes action if any of the flags is set.
- **This architecture is an improvement over simple round robin:**
  - Devices get attended to immediately. Unlikely to suffer loss of data.
- **However, it may still take a while for this data to actually be processed!**

Real-Time Software Architectures
Round Robin with Interrupts

- **For example, suppose all 3 devices A, B and C interrupt the CPU:**
  - Data from all 3 devices take 200 ms each in the main loop to process.
  - If the processing sequence is A, B, C, A, B, C, ..., then C will have to wait as long as 400ms to be processed!
- **Even C is a high priority device, it still has to wait for A and B to be done first.**

Real-Time Software Architectures Round Robin with Interrupts

- **Solutions:**
  - Move the processing code for C to its interrupt handler.
    - ✓**Interrupt handler for C will now take 200 ms!**
    - ✓**Unacceptable if the handler is high priority and blocks all lower priority interrupts during this time.**
  - Poll C more often:
    - ✓**Instead of A, B, C, A, B, C,..., we poll A, C, B, C, C, A, C, B, C, C, A, C, B, C, C, ...**

Real-Time Software Architectures Function Queue Scheduling

- **This is similar to Round Robin with Interrupts:**
  - Interrupt handlers read the data from the device.
  - BUT instead of setting a flag, it inserts a pointer to the function to process this data.
- **The main loop then takes a function from this queue and executes it.**

Real-Time Software Architectures Function Queue Scheduling

- **It is easy to enforce priorities in this scheme:**
  - Just have priorities in the way the queue is managed!
  - For example, function_C is always placed at the front of the queue ahead of everyone else.
  - This gives function_C priority over everyone else.

Real-Time Software Architectures
Timed Loops

- **A timed-loop is similar to round-robin.**
  - A while loop repeatedly calls functions to handle processing.
  - Each function is called in a fixed order.
- **Difference:**
  - Functions are called at fixed intervals.
  - Before calling the function, the main loop checks if a sufficient number of clock cycles have passed by.
- **This is useful for routines that must be called at fixed times.**

Real-Time Software Architectures
Timed Loops

- **Example:**
  - PID loops require fixed timings for accurate computation.

$$p(t_k) = K(r(t_k) - y(t_k))$$

$$d(t_k) = \frac{T_d}{T_d + Nh}(d(t_{k-1}) - kN(y(t_k) - y(t_{k-1}))$$

$$i(t_{k+1}) = i(t_k) + \frac{Kh}{T_i}(r(t_k) - y(t_k))$$

$$u(t_k) = p(t_k) + i(t_k) + d(t_k)$$

  - The computations for $d(t_k)$ and $i(t_{k+1})$ compute integrations and differentiations.
    - ✓**Accuracy is dependent on period *h*, which must be fixed.**
    - ✓**In our example, we assume a 50 Hz cycle, so *h=20* ms.**

Real-Time Software Architectures
Real-Time Operating Systems

- Infrastructure to handle multiple tasks.
  - ✓**Most embedded platforms have single CPUs that can only handle one task at a time.**
  - ✓**The RTOS provides services that automatically switch between tasks, to give the illusion that multiple tasks are executing at the same time.**
- Infrastructure to coordinate tasks.
  - ✓**Message passing, protecting critical sections, etc.**
- Other services.
  - ✓**Memory management.**
  - ✓**Disk access.**
  - ✓**Etc.**

Real-Time Software Architectures
Real-Time Operating Systems

- **When are RTOS good?**
  - Complicated applications that involve many parts that have to interact.
    - ✓**Reading sensors, reading keypads, reading operator buttons, sounding alarms, controlling actuators, sending/receivng data over the network, driving multiple displays, performing computations, etc.**

  +RTOS provides a clean, convenient and predictable way to control complex applications.

  +RTOS are often audited by independent organizations to prove that they are reliable.

    - ✓**E.g. uC/OS is certified by EuroCAE to be safe for installing on commercial airliners.**

  - RTOS are relatively huge, must be customized, and can take some time to learn and understand.

- **The rest of this course is about RTOS, so we won't go into any further detail here.**

Real Time Operating Systems

RTOS:
TASK MANAGEMENT

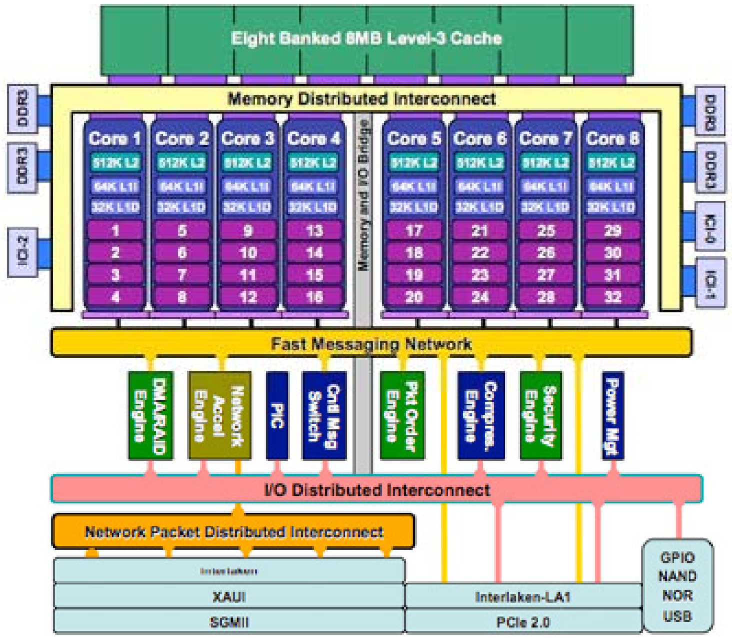
Tasks

- **While interrupts are the backbones of real-time systems, tasks are the workhorses.**
  - ISRs get information from sensors, trigger when uses press a button etc.
  - It is tasks that process these information or act on a button press.

NUS National University of Singapore
School of Computing

The Process Model

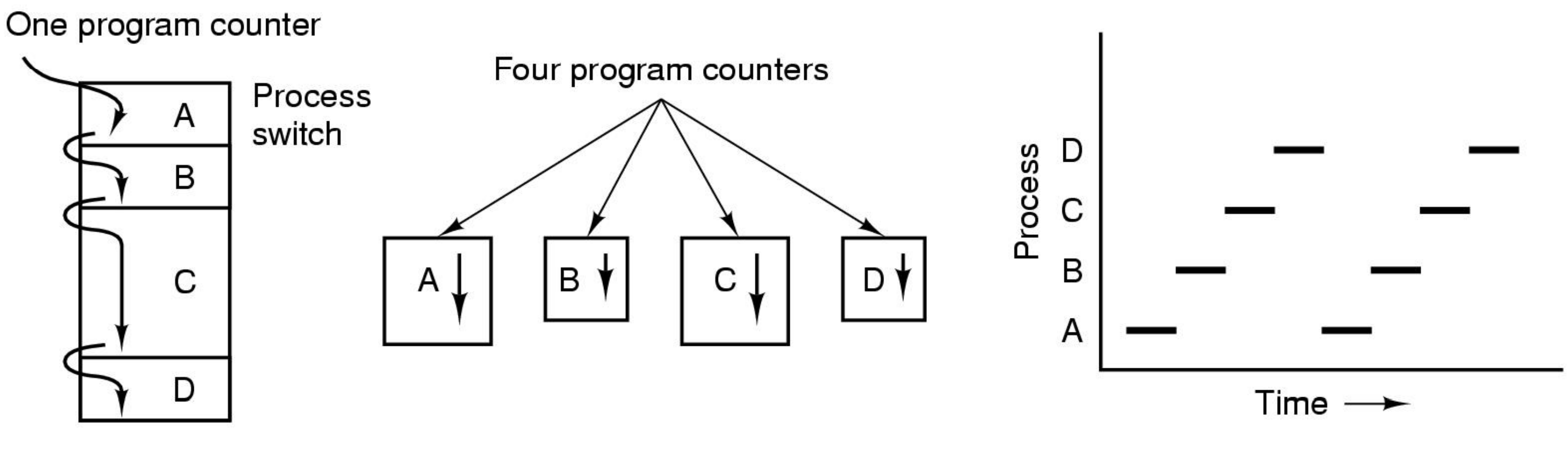
- **In this lecture we will assume a single processor with a single core.**
  - This is very typical of a microcontroller used in a real-time system.
    - ✓**Some exceptions include the ARM Cortex-A9 used in the iPhone.**
  - Most modern desktops however have “multi-core” processors:
    - ✓**Each microprocessor actually consists of multiple execution units.**



XLP832 Processor Block Diagram

The Process Model

- **The materials for this lecture come from Modern Operating Systems.**
  - This book uses the term "process" instead of "task". We will assume that they mean the same thing.
- **Since we have only a single-core single processor:**
  - At any one time, at most one process can execute.



(a) (b) (c)

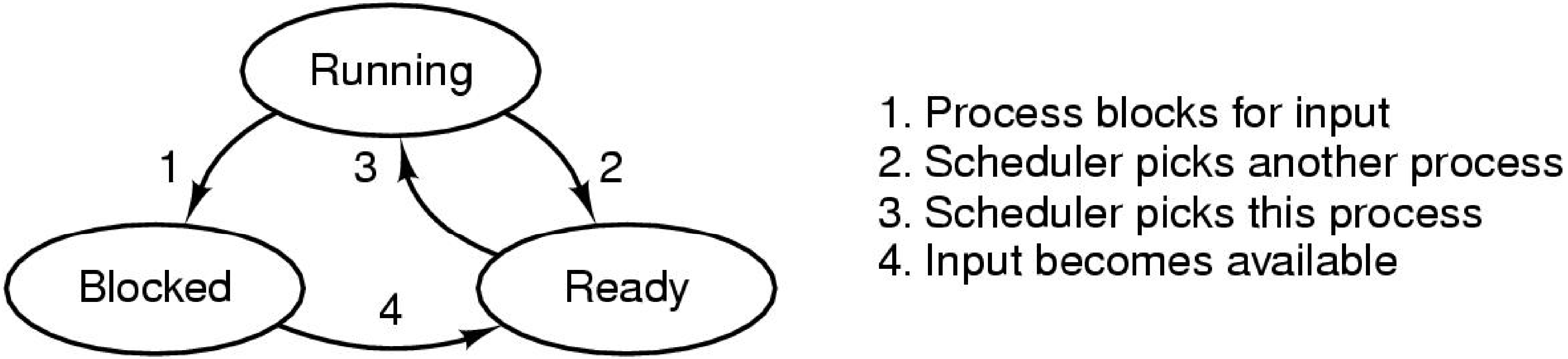
NUS National University of Singapore School of Computing

Process States

- **A process can be in one of 3 possible states:**
  - Running
    - ✓**The process is actually being executed on the CPU.**
  - Ready
    - ✓**The process is ready to run but not currently running.**
    - ✓**A “scheduling algorithm” is used to pick the next process for running.**
  - Blocked.
    - ✓**The process is waiting for “something” to happen so it is not ready to run yet.**
    - ✓**E.g. include waiting for inputs from another process.**

Process States

- **The diagram below shows the 3 possible states and the transitions between them.**



Switching between Processes

- **When a process runs, the CPU needs to maintain a lot of information about it. This is called the "process context".**
  - CPU register values.
  - Stack pointers.
  - CPU Status Word Register.
    - ✓**This maintains information about whether the previous instruction resulted in an overflow or a "zero", whether interrupts are enabled, etc.**
    - ✓**This is needed for branch instructions – assembly equivalents of "if' statements.**

The AVR Status Register – SREG – is defined as:

Bit	7	6	5	4	3	2	1	0	
0x3F (0x5F)	I	T	H	S	V	N	Z	C	SREG
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

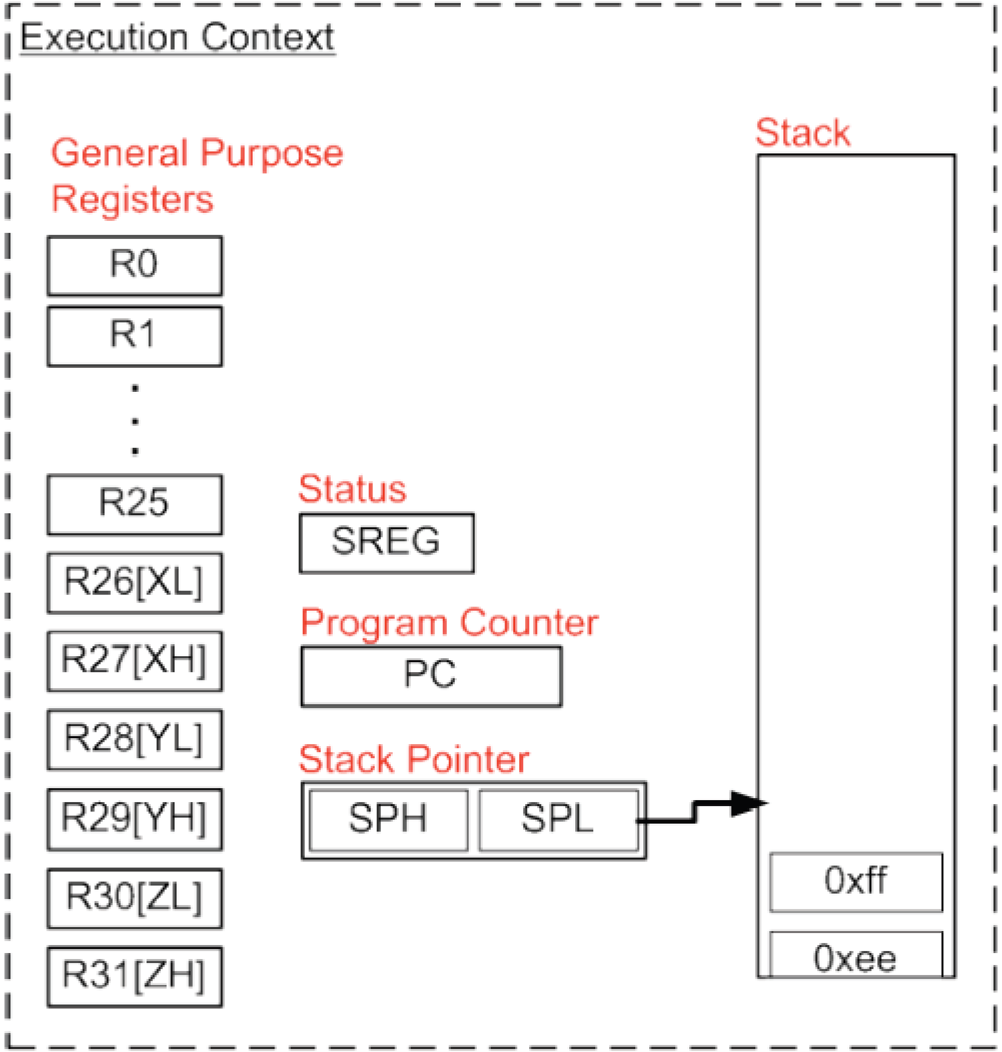
Switching between Processes

- **All of these values change as a process runs.**
- **When a process is blocked or put into a READY state, a new process will be picked to take control of the CPU.**
  - All the information for the current process must be saved!
  - The information for the new process must be loaded into the registers, stack pointer and status registers!
    - ✓**This is to allow the new process to run like as though it was never interrupted!**
- **This process is known as “context switching”.**

NUS National University of Singapore School of Computing

Context Switching on the FreeRTOS Atmega Port

- **Each process is allocated a stack.**
  - ▪Exactly what you learnt in CG1103.
  - ▪The stack pointer are two 8-bit registers SPH and SPL, together forming a single 16-bit SP.
- **The diagram shows the complete Atmega context.**
  - ▪Registers R0-R31, PC.
  - ▪Status register SREG.
  - ▪Stack pointer SPH/SPL.



Context Switching on the FreeRTOS Atmega Port

- **FreeRTOS implements context saving in a macro called “portSAVE_CONTEXT”.**

```
#define portSAVE_CONTEXT()\
asm volatile (\
      "push r0  \n\t"\               // Save R0
      "in r0, __SREG__   \n\t"\      // Read in status register SREG to R0
      "cli      \n\t"\               // Disable all interrupts for atomicity
      "push r0  \n\t"\               // Save SREG
      "push r1  \n\t"\               // Save R1
      "clr r1   \n\t"\               // AVR C expects R1 to be 0, so clear it.
      "push r2  \n\t"\               // Save R2 to R31
      ...
      "push r31 \n\t"\
      "in r26, __SP_L__  \n\t"\      // Read in stack pointer low byte
      "in r27, __SP_H__  \n\t"\      // and high byte
      "sts pxCurrentTCB+1, r27     \n\t"\    // And save it to pxCurrentTCB
      "sts pxCurrentTCB, r26       \n\t"\
      "sei      \n\t" : : :\                 // Re-enable interrupts
);
```

Context Switching on the FreeRTOS Atmega Port

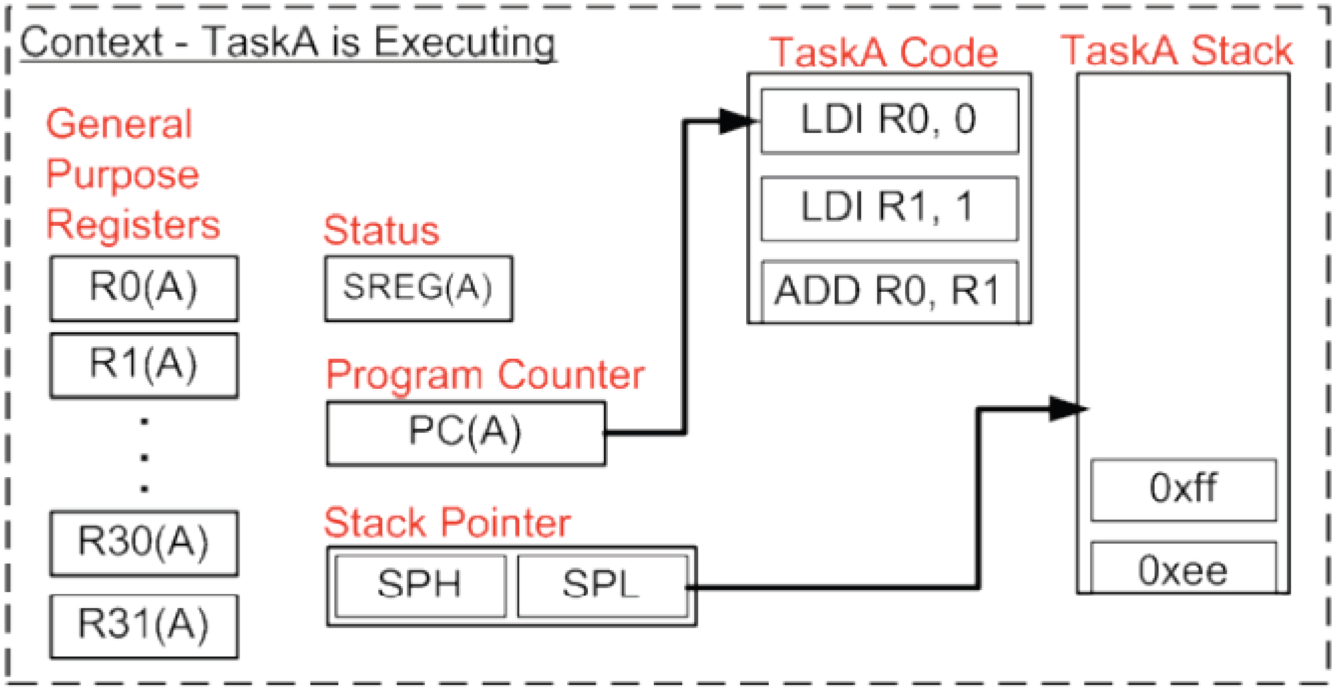
- **The reverse operation is portRESTORE_CONTEXT. The stack pointer for the process being restored must be in pxCurrentTCB.**

```
#define portRESTORE_CONTEXT()\
asm volatile (\
      "out __SP_L__, %A0 \n\t"\    // Copy SP_L and SP_H from the
      "out __SP_H__, %B0 \n\t"\    // pxCurrentTCB variable.
      "pop r31  \n\t"\             // Restore registers r31 to r1.
      ...
      "pop r0   \n\t"\             // Pop out SREG
      "out __SREG__, r0\n\t"\      // And restore it.
      "pop r0   \n\t": : "r" (pxCurrentTCB):\         // Restore R0

      );
```

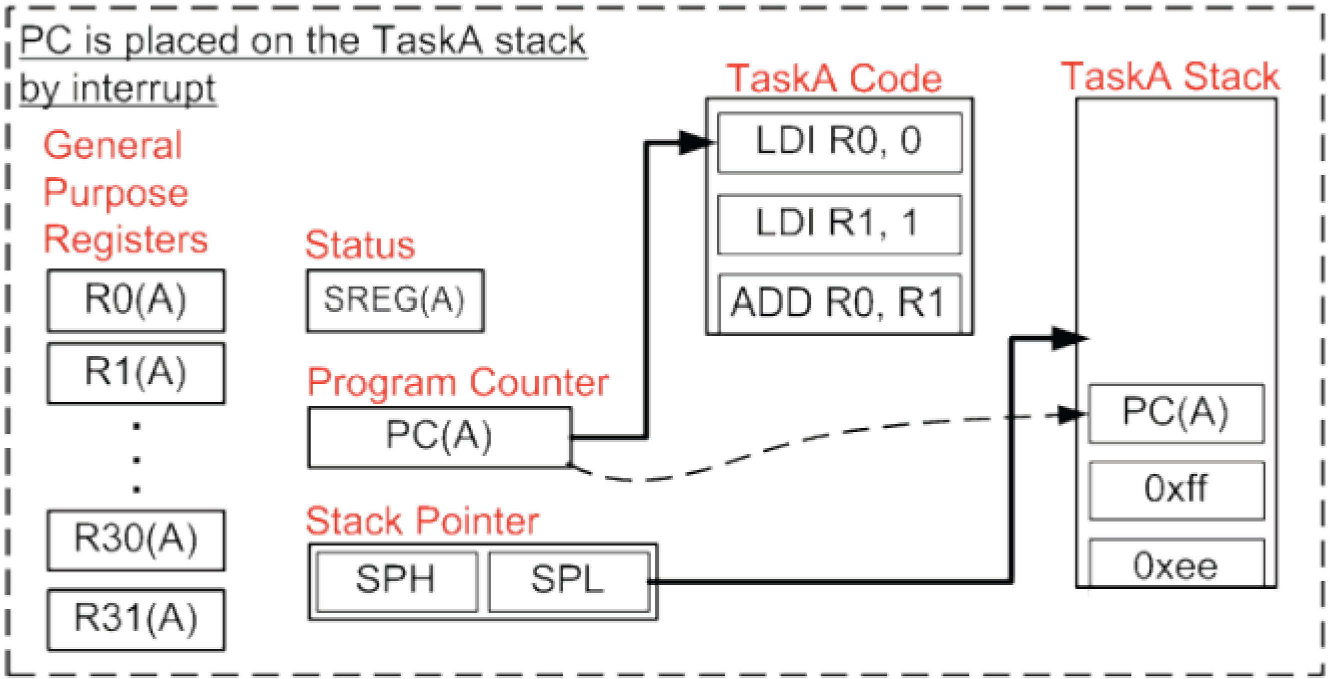
Context Switching on the FreeRTOS Atmega Port

- **We will now see step-by-step how this works.**
- **Assume that at first Task A is executing.**
  - PC would be pointing at Task A code, SPH/SPL pointing at Task A stack, Registers R0-R31 contain Task A data.



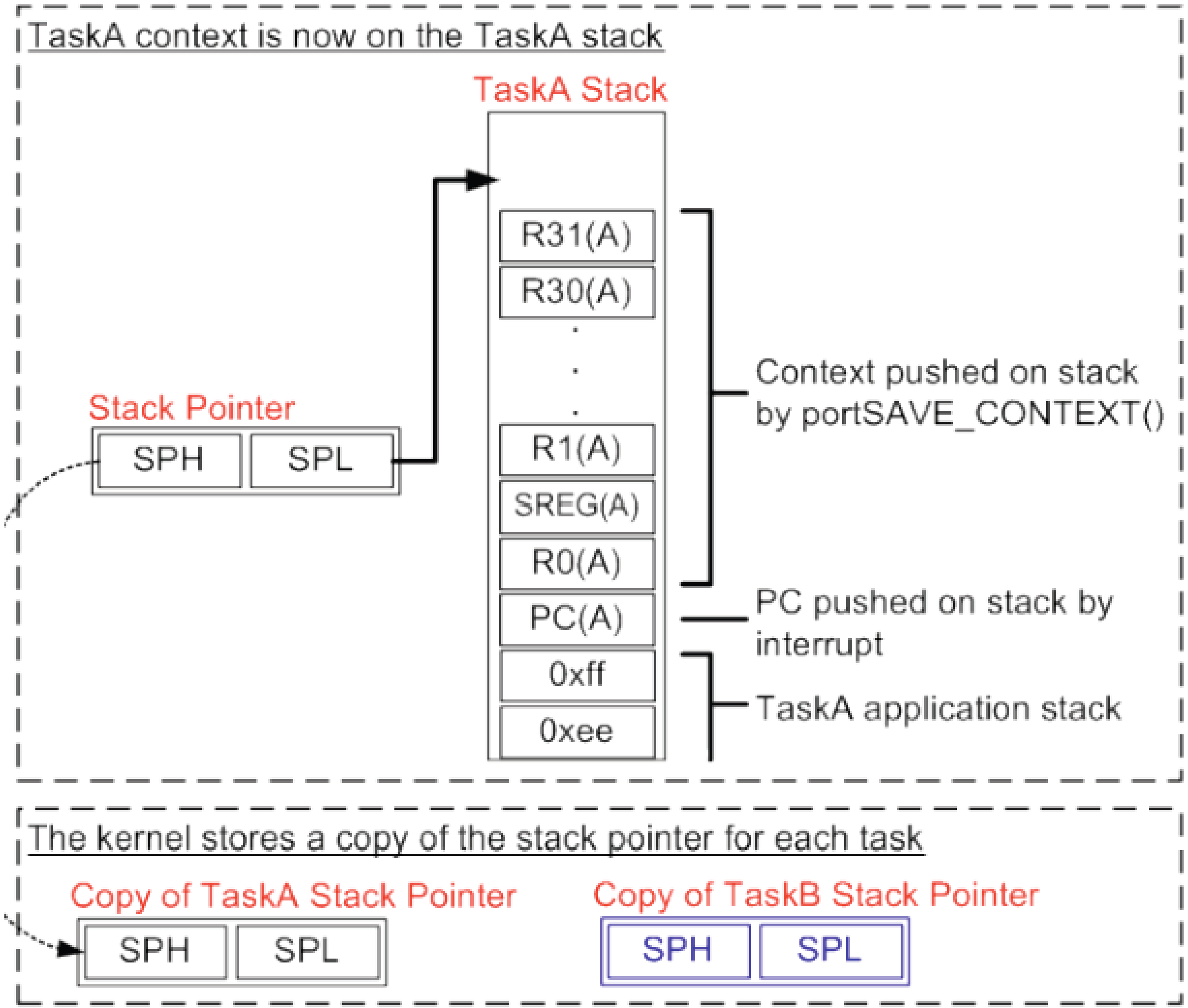
Context Switching on the FreeRTOS Atmega Port

- **FreeRTOS relies on regular interrupts from Timer 0 to switch between tasks. When the interrupt triggers, PC is placed onto Task A's stack.**



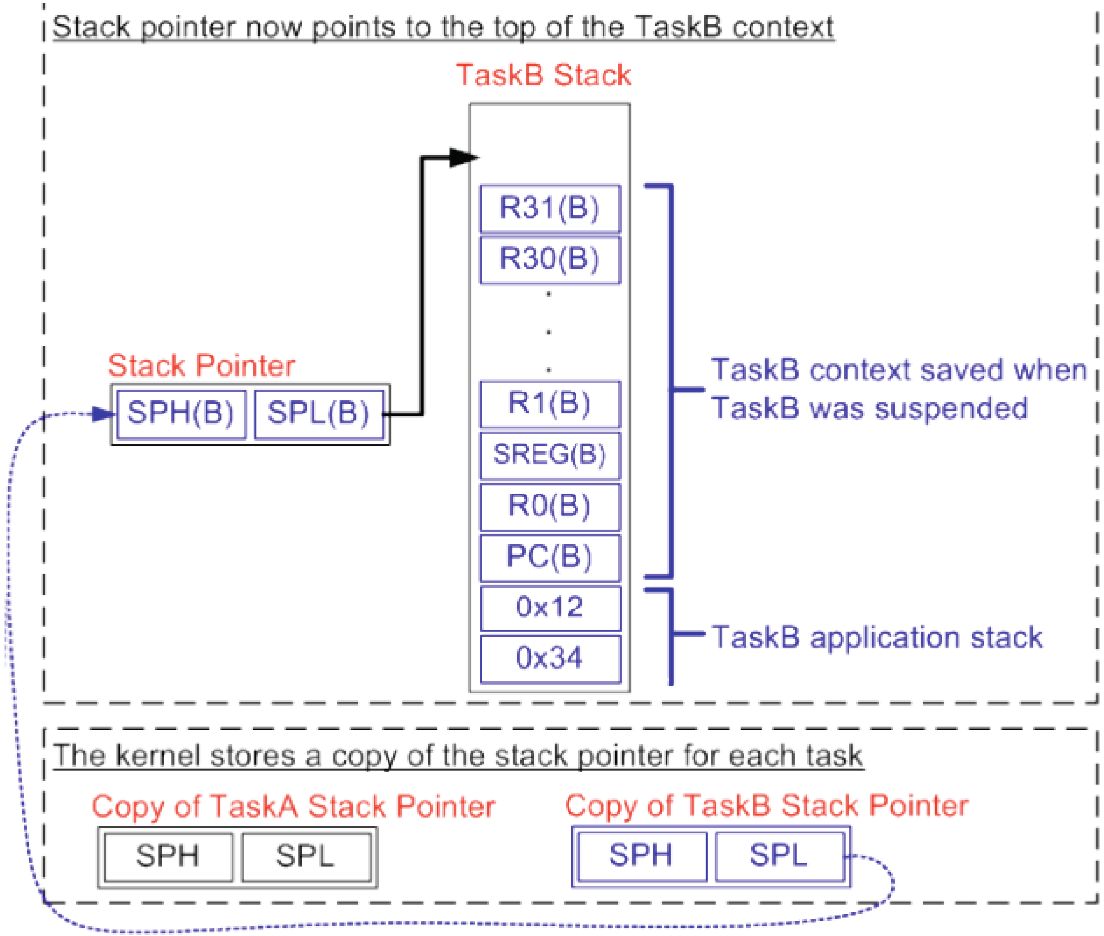
Context Switching on the FreeRTOS Atmega Port

- **The ISR calls portSAVECONTEXT, resulting in Task A's context being pushed onto the stack.**
- **pxCurrentTCB will also hold SPH/SPL _after_ the context save.**
  - This must be saved by the kernel.



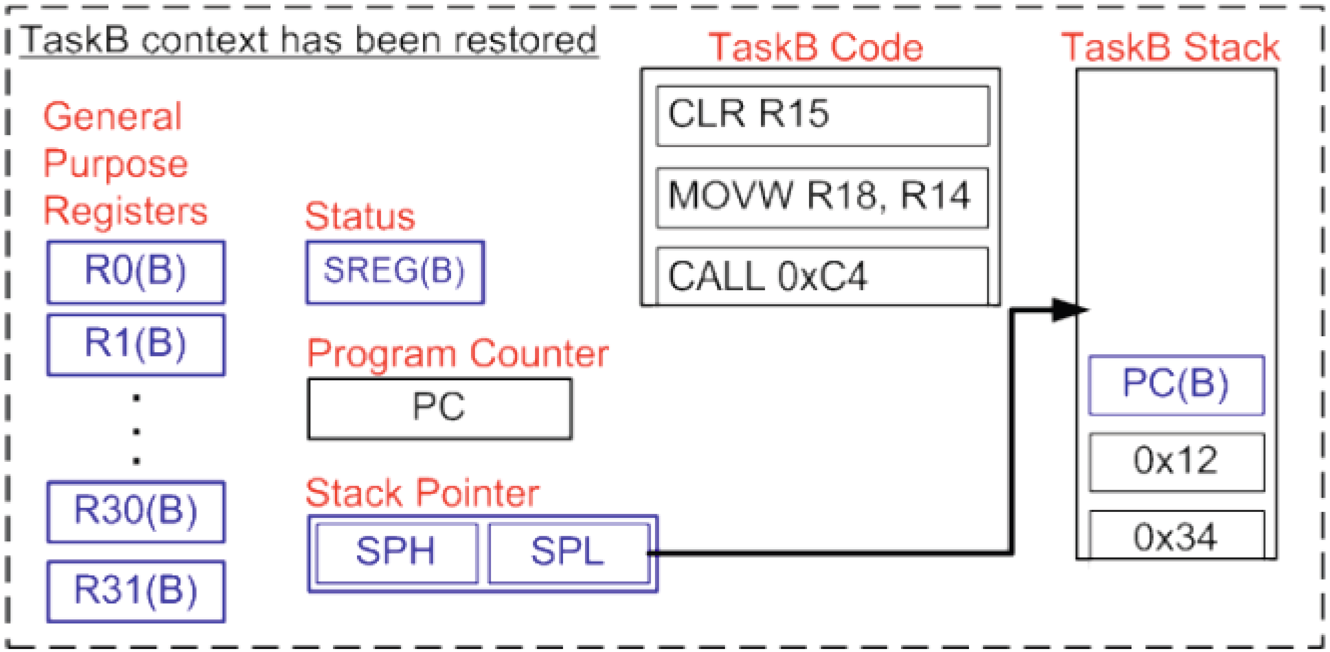
Context Switching on the FreeRTOS Atmega Port

- **The kernel then selects Task B to run, and copies its SPH/SPL values into pxCurrentTCB and calls portRESTORE_CONTEXT.**
  - The first two lines will copy pxCurrentTCB into SPH/SPL, causing SP to point to Task B's stack.



Context Switching on the FreeRTOS Atmega Port

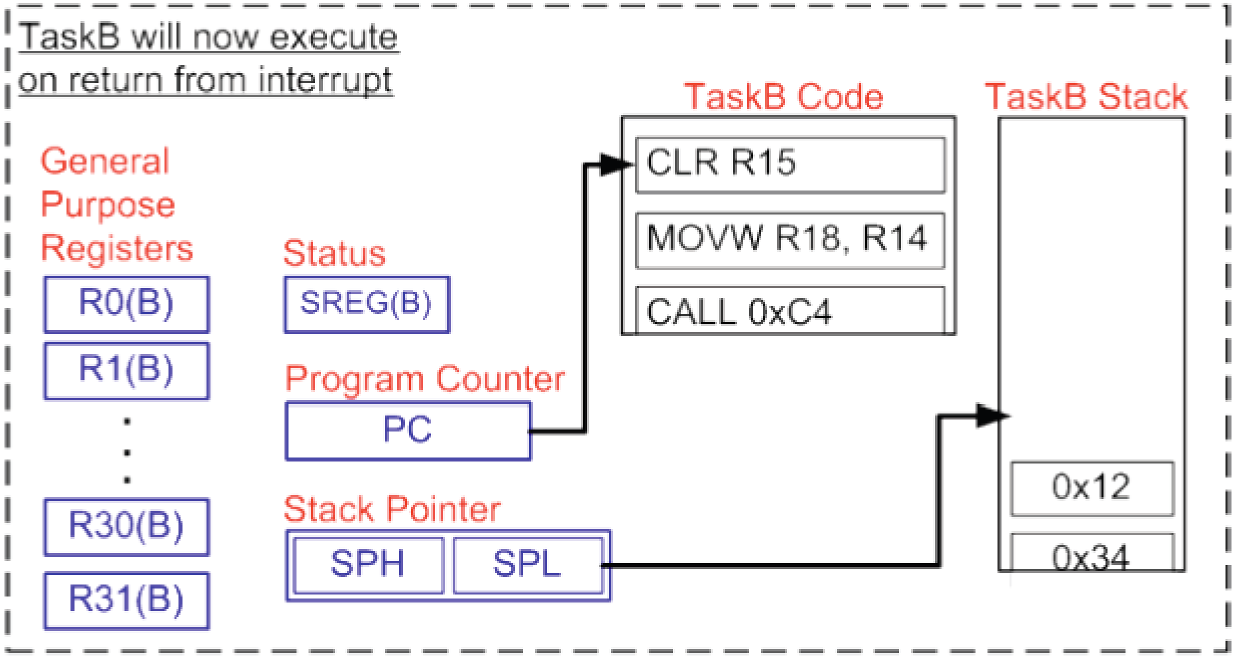
- **The rest of portRESTORE_CONTEXT is executed, causing Task B's data to be loaded into R31-R0 and SREG.**
  - Now Task B can resume like as though nothing happened!



Only the program counter remains on the stack.

Context Switching on the FreeRTOS Atmega Port

- **Only Task B's PC remains on the stack. Now the ISR exits, causing this value to be popped off onto the AVR's PC.**
  - PC points to the next instruction to be executed.
  - End result: Task B resumes execution, with all its data and SREG intact!



Context Switching on the FreeRTOS Atmega Port

- **Here we looked at context switching controlled by a timer.**
  - It can also be triggered by other things:
    - ✓**Currently running processed waiting for input.**
    - ✓**Currently running task blocking on a synchronization mechanism (see next lecture).**
    - ✓**Currently running task wants to sleep for a fixed period.**
    - ✓**Higher priority task becoming "READY".**
    - ✓**…**

Real Time Operating Systems

RTOS:
TASK SCHEDULING

Fixed Priority

- **In fixed priority:**
  - Every task is assigned a unique priority when the task is created (using os_create_task for example).
    - ✓**Standard practice is to have a range of 0 to n, with 0 being the highest priority.**
  - When one task completes, the highest priority READY task is picked to run.

NUS National University of Singapore School of Computing

Fixed Priority

	Ci	Pi
P1	1	4
P2	2	8
P3	3	12

P1	P2	P3
4	8	12
8	16	24
12	24	36
16	32	48
20	40	60
24	48	72
28	56	84
32	64	96
36	72	108
40	80	120

Time	Process	Deadline	Priority	Comments
0	P1	P1,P2,P3	P1P2P3	
1	P2		P1P2P3	
2	P2		P1P2P3	
3	P3		P1P2P3	
4	P3	P1	P1P2P3	
5	P3		P1P2P3	
6	P1		P1P2P3	
7			P1P2P3	
8	P1	P1, P2	P1P2P3	
9	P2		P1P2P3	
10	P2		P1P2P3	
11			P1P2P3	
12	P1	P1, P3	P1P2P3	
13	P3		P1P2P3	
14	P3		P1P2P3	
15	P3		P1P2P3	
16	P1	P1, P2	P1P2P3	

Fixed Priority

- **Fixed priority depends on the programmer to assign priorities to tasks.**
  - How to assign?
    - ✓**Based on importance of a task?**
    - ✓**Based on periods?**
    - ✓**…??**
  - In addition two different programmers may not agree on the relative importance of tasks.
    - ✓**Can be subjective.**
  - Nonetheless fixed priority is simple to implement and therefore popular.
    - ✓**E.g. uC/OS-II.**

Rate Monotonic Scheduling

- **In the previous slide, Task P1 missed its deadline because of it's tight period, and as a result, tight deadline.**
  - Notice that the t=32 deadline was skipped completely.
- **One way around this problem is to prioritize tasks according to their periods P_i. This results in the schedule shown in the next slide.**
  - P1 has the smallest period of 4, and has the highest priority. P2 is next with 8, P3 is last with a period of 12.

Pre-Emptive Scheduling

- **In pre-emptive scheduling, a task higher priority task that becomes due to run can pre-empt a currently running task.**
  - In our previous example, when P1 became ready at time 4, it can pre-empt P3 and P2.

NUS National University of Singapore
School of Computing

Rate Monotonic Scheduling Pre-Emptive Case

Time	Process	Priority	Deadline
0	P1	P1P2P3	P1P2P3
1	P2	P1P2P3	
2	P2	P1P2P3	
3	P3	P1P2P3	
4	P1	P1P2P3	P1
5	P3	P1P2P3	
6	P3	P1P2P3	
7		P1P2P3	
8	P1	P1P2P3	P1P2
9	P2	P1P2P3	
10	P2	P1P2P3	
11		P1P2P3	
12	P1	P1P2P3	P1P3
13	P3	P1P2P3	
14	P3	P1P2P3	
15	P3	P1P2P3	
16	P1	P1P2P3	P1P2
17	P2	P1P2P3	
18	P2	P1P2P3	
19		P1P2P3	
20	P1	P1P2P3	P1
21		P1P2P3	
22		P1P2P3	
23		P1P2P3	
24	P1	P1P2P3	P1P2P3
25	P2	P1P2P3	
26	P3	P1P2P3	
27	P3	P1P2P3	
28	P1	P1P2P3	P1
29	P3	P1P2P3	
30		P1P2P3	
31		P1P2P3	
32	P1	P1P2P3	P1P2
33	P2	P1P2P3	
34	P2	P1P2P3	
35		P1P2P3	
36	P1	P1P2P3	P1P3

Pre-Emptive RMS Unschedulable Example

- **Consider 3 tasks with the following characteristics:**

	Ci	Pi
P1	1	3
P2	2	6
P3	3	8

NUS National University of Singapore School of Computing

Unschedulable Pre-emptive RMS

Time	Process	Priority	Deadline
0	P1	P1P2P3	P1P2P3
1	P2	P1P2P3	
2	P2	P1P2P3	
3	P1	P1P2P3	P1
4	P3	P1P2P3	
5	P3	P1P2P3	
6	P1	P1P2P3	P1P2
7	P2	P1P2P3	
8	P2	P1P2P3	P3
9	P1	P1P2P3	P1
10	P3*	P1P2P3	
11	P3	P1P2P3	
12	P1	P1P2P3	P1P2
13	P2	P1P2P3	
14	P2	P1P2P3	
15	P1	P1P2P3	P1
16	P3*	P1P2P3	P3
17	P3*	P1P2P3	
18	P1	P1P2P3	P1P2
19	P2	P1P2P3	
20	P2	P1P2P3	
21	P1	P1P2P3	P1
22	P3	P1P2P3	
23	P3	P1P2P3	
24	P1	P1P2P3	P1P2P3
25	P2	P1P2P3	
26	P2	P1P2P3	
27	P1	P1P2P3	P1
28	P3*	P1P2P3	
29		P1P2P3	
30	P1	P1P2P3	P1P2
31	P2	P1P2P3	
32	P2	P1P2P3	P3
33	P1	P1P2P3	P1
34	P3	P1P2P3	
35	P3	P1P2P3	
36	P1	P1P2P3	P1P2

The runs marked with * (e.g. P3*) are runs that have already exceeded their deadlines.

Unschedulable Pre-Emptive RMS

- **In the previous example, P3 keeps missing its deadline.**
  - This is because P3 keeps getting pre-empted by P1 and P2.
  - Eventually P3 gets delayed so much that it misses the deadline.

Liu and Layland Criteria for Schedulability of RMS

- **Liu and Leyland (1973) states that for a set of *n* tasks, the tasks are schedulable provided that:**

$$\sum_{i=1}^{n} \frac{C_i}{P_i} \leq n(2^{\frac{1}{n}} - 1)$$

L&L Thresholds

- **This table shows the L&L thresholds for various values of *n*.**
  - Notice that *n* converges towards 69.3% for large *n*.
    - ✓**Thus, in general, as long as the CPU is used less than 69.3% of the time, a schedule is guaranteed.**

n	Threshold
1	100%
2	82.84%
3	77.79%
4	75.68%
100	69.55%
Large	69.3%

NUS National University of Singapore
School of Computing

L&L Threshold

- **Analysing our two problems:**
  - L&L Threshold for 3 tasks = 77.79%

	Ci	Pi
P1	1	4
P2	2	8
P3	3	12

	Ci	Pi
P1	1	3
P2	2	6
P3	3	8

$$\sum_{i=1}^{n} \frac{C_i}{P_i} = \frac{1}{4} + \frac{2}{8} + \frac{3}{12} = 75\%$$

$$\sum_{i=1}^{n} \frac{C_i}{P_i} = \frac{1}{3} + \frac{2}{6} + \frac{3}{8} = 104.2\%$$

Limits of the L&L Criteria

- **The L&L criteria can only tell you if a set of tasks is guaranteed to be schedulable.**
  - As long as the criteria is met, all tasks will meet their deadlines in a pre-emptive system.
- **However, tasks that don’t meet the criteria are not necessarily unschedulable!**
  - Relatively easy to construct a counter-example.

Rate Monotonic Scheduling

- **In the previous slide, Task P1 missed its deadline because of it's tight period, and as a result, tight deadline.**
  - Notice that the t=32 deadline was skipped completely.
- **One way around this problem is to prioritize tasks according to their periods P_i. This results in the schedule shown in the next slide.**
  - P1 has the smallest period of 4, and has the highest priority. P2 is next with 8, P3 is last with a period of 12.

Pre-Emptive Scheduling

- **In pre-emptive scheduling, a task higher priority task that becomes due to run can pre-empt a currently running task.**
  - In our previous example, when P1 became ready at time 4, it can pre-empt P3 and P2.

Rate Monotonic Scheduling Pre-Emptive Case

Time	Process	Priority	Deadline
0	P1	P1P2P3	P1P2P3
1	P2	P1P2P3	
2	P2	P1P2P3	
3	P3	P1P2P3	
4	P1	P1P2P3	P1
5	P3	P1P2P3	
6	P3	P1P2P3	
7		P1P2P3	
8	P1	P1P2P3	P1P2
9	P2	P1P2P3	
10	P2	P1P2P3	
11		P1P2P3	
12	P1	P1P2P3	P1P3
13	P3	P1P2P3	
14	P3	P1P2P3	
15	P3	P1P2P3	
16	P1	P1P2P3	P1P2
17	P2	P1P2P3	
18	P2	P1P2P3	
19		P1P2P3	
20	P1	P1P2P3	P1
21		P1P2P3	
22		P1P2P3	
23		P1P2P3	
24	P1	P1P2P3	P1P2P3
25	P2	P1P2P3	
26	P3	P1P2P3	
27	P3	P1P2P3	
28	P1	P1P2P3	P1
29	P3	P1P2P3	
30		P1P2P3	
31		P1P2P3	
32	P1	P1P2P3	P1P2
33	P2	P1P2P3	
34	P2	P1P2P3	
35		P1P2P3	
36	P1	P1P2P3	P1P3

Pre-Emptive RMS Unschedulable Example

- **Consider 3 tasks with the following characteristics:**

	Ci	Pi
P1	1	3
P2	2	6
P3	3	8

Unschedulable Pre-emptive RMS

Time	Process	Priority	Deadline
0	P1	P1P2P3	P1P2P3
1	P2	P1P2P3	
2	P2	P1P2P3	
3	P1	P1P2P3	P1
4	P3	P1P2P3	
5	P3	P1P2P3	
6	P1	P1P2P3	P1P2
7	P2	P1P2P3	
8	P2	P1P2P3	P3
9	P1	P1P2P3	P1
10	P3*	P1P2P3	
11	P3	P1P2P3	
12	P1	P1P2P3	P1P2
13	P2	P1P2P3	
14	P2	P1P2P3	
15	P1	P1P2P3	P1
16	P3*	P1P2P3	P3
17	P3*	P1P2P3	
18	P1	P1P2P3	P1P2
19	P2	P1P2P3	
20	P2	P1P2P3	
21	P1	P1P2P3	P1
22	P3	P1P2P3	
23	P3	P1P2P3	
24	P1	P1P2P3	P1P2P3
25	P2	P1P2P3	
26	P2	P1P2P3	
27	P1	P1P2P3	P1
28	P3*	P1P2P3	
29		P1P2P3	
30	P1	P1P2P3	P1P2
31	P2	P1P2P3	
32	P2	P1P2P3	P3
33	P1	P1P2P3	P1
34	P3	P1P2P3	
35	P3	P1P2P3	
36	P1	P1P2P3	P1P2

The runs marked with * (e.g. P3*) are runs that have already exceeded their deadlines.

Unschedulable Pre-Emptive RMS

- **In the previous example, P3 keeps missing its deadline.**
  - This is because P3 keeps getting pre-empted by P1 and P2.
  - Eventually P3 gets delayed so much that it misses the deadline.

Liu and Layland Criteria for Schedulability of RMS

- **Liu and Leyland (1973) states that for a set of *n* tasks, the tasks are schedulable provided that:**

$$\sum_{i=1}^{n} \frac{C_i}{P_i} \leq n(2^{\frac{1}{n}} - 1)$$

L&L Thresholds

- **This table shows the L&L thresholds for various values of *n*.**
  - Notice that *n* converges towards 69.3% for large *n*.
    - ✓**Thus, in general, as long as the CPU is used less than 69.3% of the time, a schedule is guaranteed.**

n	Threshold
1	100%
2	82.84%
3	77.79%
4	75.68%
100	69.55%
Large	69.3%

NUS National University of Singapore

School of Computing

L&L Threshold

- **Analysing our two problems:**
  - L&L Threshold for 3 tasks = 77.79%

	Ci	Pi
P1	1	4
P2	2	8
P3	3	12

	Ci	Pi
P1	1	3
P2	2	6
P3	3	8

$$\sum_{i=1}^{n} \frac{C_i}{P_i} = \frac{1}{4} + \frac{2}{8} + \frac{3}{12} = 75\%$$

$$\sum_{i=1}^{n} \frac{C_i}{P_i} = \frac{1}{3} + \frac{2}{6} + \frac{3}{8} = 104.2\%$$

Limits of the L&L Criteria

- **The L&L criteria can only tell you if a set of tasks is guaranteed to be schedulable.**
  - As long as the criteria is met, all tasks will meet their deadlines in a pre-emptive system.
- **However, tasks that don’t meet the criteria are not necessarily unschedulable!**
  - Relatively easy to construct a counter-example.

A Stronger Schedulability Test Critical Instant Analysis

- **Leedham and Seow give a better test in pages 168-173.**
- **To use this test, assume:**
  - $T = \{T_1, T_2, T_3, \ldots\}$ is the set of tasks. In the previous slides we used P1, P2, etc. to denote tasks. T_i denote the same tasks, but sorted by periods. T_1 is the task with the shortest period, T_2 has the next shortest, etc.
  - C_i is the running time of T_i assuming it is not pre-empted.
  - P_i is the period of task T_i.

–

Critical Instant Analysis

1. Sort T by period of each task, if T is not already sorted. We will assume that T_1 has the shortest period, T_2 has the 2nd shortest, etc.
2. For each task $T_i \in T$, recursively compute S_{i0}, S_{i1}, … where:
   a. $S_{i,0} = \sum_{j=1}^{i} C_j$
   b. $S_{i,(x+1)} = C_i + \sum_{j=1}^{i-1} C_j \times \left\lceil \frac{S_{i,x}}{P_j} \right\rceil$

   Stop when $S_{i,(x+1)} = S_{i,x}$. Call this $S_{i,(x+1)}$ the final value $S_{i,F}$. I.e. let $S_{i,F} = S_{i,(x+1)}$ when the iteration terminates.

3. If $S_{i,F} < D_i$, then the task *i* is schedulable and will not miss its deadlines.

Rate Monotonic Scheduling

- **In the previous slide, Task P1 missed its deadline because of it's tight period, and as a result, tight deadline.**
  - Notice that the t=32 deadline was skipped completely.
- **One way around this problem is to prioritize tasks according to their periods P_i. This results in the schedule shown in the next slide.**
  - P1 has the smallest period of 4, and has the highest priority. P2 is next with 8, P3 is last with a period of 12.

Pre-Emptive Scheduling

- **In pre-emptive scheduling, a task higher priority task that becomes due to run can pre-empt a currently running task.**
  - In our previous example, when P1 became ready at time 4, it can pre-empt P3 and P2.

Rate Monotonic Scheduling Pre-Emptive Case

Time	Process	Priority	Deadline
0	P1	P1P2P3	P1P2P3
1	P2	P1P2P3	
2	P2	P1P2P3	
3	P3	P1P2P3	
4	P1	P1P2P3	P1
5	P3	P1P2P3	
6	P3	P1P2P3	
7		P1P2P3	
8	P1	P1P2P3	P1P2
9	P2	P1P2P3	
10	P2	P1P2P3	
11		P1P2P3	
12	P1	P1P2P3	P1P3
13	P3	P1P2P3	
14	P3	P1P2P3	
15	P3	P1P2P3	
16	P1	P1P2P3	P1P2
17	P2	P1P2P3	
18	P2	P1P2P3	
19		P1P2P3	
20	P1	P1P2P3	P1
21		P1P2P3	
22		P1P2P3	
23		P1P2P3	
24	P1	P1P2P3	P1P2P3
25	P2	P1P2P3	
26	P3	P1P2P3	
27	P3	P1P2P3	
28	P1	P1P2P3	P1
29	P3	P1P2P3	
30		P1P2P3	
31		P1P2P3	
32	P1	P1P2P3	P1P2
33	P2	P1P2P3	
34	P2	P1P2P3	
35		P1P2P3	
36	P1	P1P2P3	P1P3

Pre-Emptive RMS Unschedulable Example

- **Consider 3 tasks with the following characteristics:**

	Ci	Pi
P1	1	3
P2	2	6
P3	3	8

NUS National University of Singapore School of Computing

Unschedulable Pre-emptive RMS

Time	Process	Priority	Deadline
0	P1	P1P2P3	P1P2P3
1	P2	P1P2P3	
2	P2	P1P2P3	
3	P1	P1P2P3	P1
4	P3	P1P2P3	
5	P3	P1P2P3	
6	P1	P1P2P3	P1P2
7	P2	P1P2P3	
8	P2	P1P2P3	P3
9	P1	P1P2P3	P1
10	P3*	P1P2P3	
11	P3	P1P2P3	
12	P1	P1P2P3	P1P2
13	P2	P1P2P3	
14	P2	P1P2P3	
15	P1	P1P2P3	P1
16	P3*	P1P2P3	P3
17	P3*	P1P2P3	
18	P1	P1P2P3	P1P2
19	P2	P1P2P3	
20	P2	P1P2P3	
21	P1	P1P2P3	P1
22	P3	P1P2P3	
23	P3	P1P2P3	
24	P1	P1P2P3	P1P2P3
25	P2	P1P2P3	
26	P2	P1P2P3	
27	P1	P1P2P3	P1
28	P3*	P1P2P3	
29		P1P2P3	
30	P1	P1P2P3	P1P2
31	P2	P1P2P3	
32	P2	P1P2P3	P3
33	P1	P1P2P3	P1
34	P3	P1P2P3	
35	P3	P1P2P3	
36	P1	P1P2P3	P1P2

The runs marked with * (e.g. P3*) are runs that have already exceeded their deadlines.

Unschedulable Pre-Emptive RMS

- **In the previous example, P3 keeps missing its deadline.**
  - This is because P3 keeps getting pre-empted by P1 and P2.
  - Eventually P3 gets delayed so much that it misses the deadline.

Liu and Layland Criteria for Schedulability of RMS

- **Liu and Leyland (1973) states that for a set of *n* tasks, the tasks are schedulable provided that:**

$$\sum_{i=1}^{n} \frac{C_i}{P_i} \leq n(2^{\frac{1}{n}} - 1)$$

L&L Thresholds

- **This table shows the L&L thresholds for various values of *n*.**
  - Notice that *n* converges towards 69.3% for large *n*.
    - ✓**Thus, in general, as long as the CPU is used less than 69.3% of the time, a schedule is guaranteed.**

n	Threshold
1	100%
2	82.84%
3	77.79%
4	75.68%
100	69.55%
Large	69.3%

NUS National University of Singapore School of Computing

L&L Threshold

- **Analysing our two problems:**
  - L&L Threshold for 3 tasks = 77.79%

	Ci	Pi
P1	1	4
P2	2	8
P3	3	12

	Ci	Pi
P1	1	3
P2	2	6
P3	3	8

$$\sum_{i=1}^{n} \frac{C_i}{P_i} = \frac{1}{4} + \frac{2}{8} + \frac{3}{12} = 75\%$$

$$\sum_{i=1}^{n} \frac{C_i}{P_i} = \frac{1}{3} + \frac{2}{6} + \frac{3}{8} = 104.2\%$$

Limits of the L&L Criteria

- **The L&L criteria can only tell you if a set of tasks is guaranteed to be schedulable.**
  - As long as the criteria is met, all tasks will meet their deadlines in a pre-emptive system.
- **However, tasks that don’t meet the criteria are not necessarily unschedulable!**
  - Relatively easy to construct a counter-example.

A Stronger Schedulability Test Critical Instant Analysis

- **Leedham and Seow give a better test in pages 168-173.**
- **To use this test, assume:**
  - $T = \{T_1, T_2, T_3, \ldots\}$ is the set of tasks. In the previous slides we used P1, P2, etc. to denote tasks. T_i denote the same tasks, but sorted by periods. T_1 is the task with the shortest period, T_2 has the next shortest, etc.
  - C_i is the running time of T_i assuming it is not pre-empted.
  - P_i is the period of task T_i.

–

Critical Instant Analysis

1. Sort T by period of each task, if T is not already sorted. We will assume that T_1 has the shortest period, T_2 has the 2^{nd} shortest, etc.
2. For each task $T_i \in T$, recursively compute S_{i0}, S_{i1}, … where:

   a. $S_{i,0} = \sum_{j=1}^{i} C_j$

   b. $S_{i,(x+1)} = C_i + \sum_{j=1}^{i-1} C_j \times \left\lceil \frac{S_{i,x}}{P_j} \right\rceil$

   Stop when $S_{i,(x+1)} = S_{i,x}$. Call this $S_{i,(x+1)}$ the final value $S_{i,F}$. I.e. let $S_{i,F} = S_{i,(x+1)}$ when the iteration terminates.

3. If $S_{i,F} < D_i$, then the task *i* is schedulable and will not miss its deadlines.

Earliest Deadline First Scheduling

- **In this scheduling algorithm, tasks are prioritized according to whose deadline is the closest.**
  - Unlike RMS, the priorities can and do change dynamically.
  - This is because a process may be delayed or pre-empted by currently higher priority processes, and as a result end closer to their deadlines.

Earliest Deadline First Scheduling

	Ci	Pi
P1	1	4
P2	2	6
P3	3	8

Time	Process	Deadline	Oustanding
0	P1	P1P2P3	P2P3
1	P2		P3
2	P2		P3
3	P3		-
4	P3	P1	P1
5	P3		P1
6	P1	P2	P2
7	P2		
8	P2	P1P3	P1P3
9	P1		P3
10	P3		-
11	P3		-
12	P3	P1P2	P1P2
13	P1		P2
14	P2		-
15	P2		-
16	P1	P1P3	P3
17	P3		-
18	P3	P2	P2
19	P3		P2
20	P1	P1	P2
21	P2		-
22	P2		-
23			
24	P1	P1P2P3	P2P3
25	P2		
26	P2		
27	P3		
28	P3	P1	P1
29	P3		P1
30	P1	P2	P2
31	P2		-
32	P2	P1P3	P1P3
33	P1		P3
34	P3		-
35	P3		-
36	P3	P1P2	P1P2

Aperiodic Tasks

- **Not all tasks occur at fixed periods.**
  - E.g. a task to read a key from the keyboard will run only when someone presses a key.
- **We need to make an aperiodic task become periodic in order to apply our periodicity analysis.**

NUS National University of Singapore School of Computing

Aperiodic Tasks

- **Two Approaches, both based on fixing the aperiodic tasks to a period.**
  - Conservative
    - ✓**Assume the worst case that the sporadic tasks occur at their maximum rate, i.e., their shortest periods.**
  - Optimistic
    - ✓**Assume that sporadic tasks occur at their average rates.**
    - ✓**This can cause some tasks to miss their deadline, even if the analysis guarantees schedulability.**
      - –*Transient Overload*

Aperiodic Task

- **Which approach should we use?**
  - Approach 1: We take worse case assumption that tasks run at their maximum rates.
    - ✓**Guarantees no missed deadlines.**
    - ✓**Good for hard real-time system**
    - ✓**BUT may result in significant CPU underutilization.**
  - Approach 2: Take average rates.
    - ✓**Better CPU utilization, BUT**
    - ✓**If task activates at a higher than average rate, some tasks will miss their deadlines.**
      - *System is temporarily overloaded.*

Real Time Operating Systems

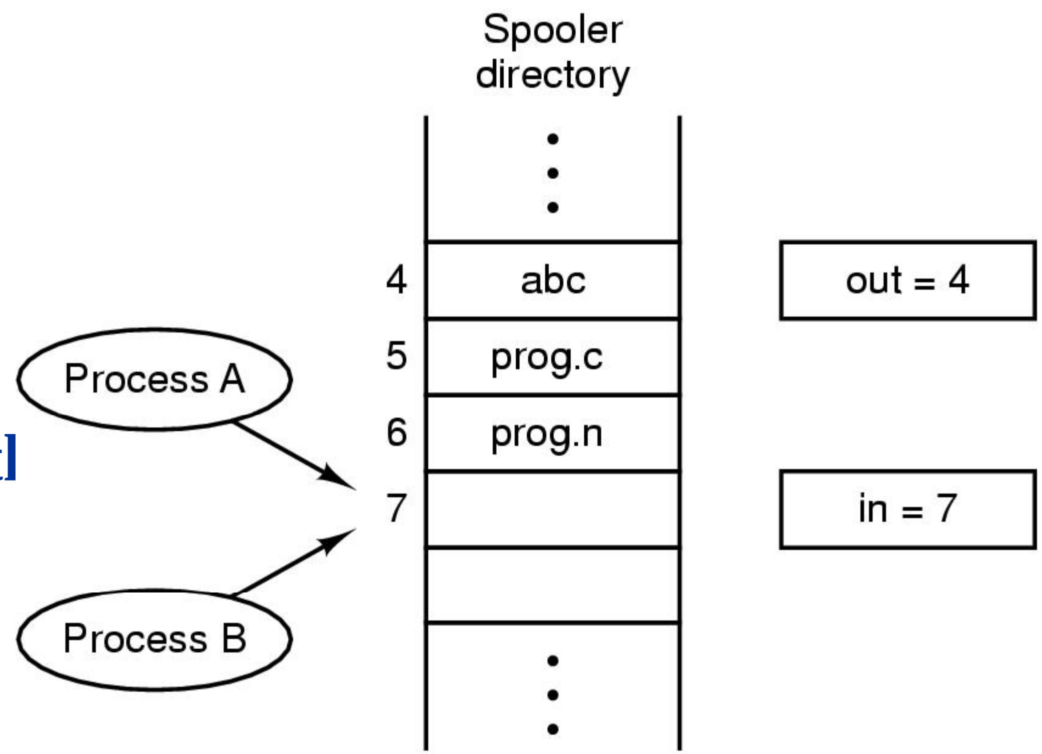
RTOS:
TASK COORDINATION AND COMMUNICATION

Race Conditions

- **Race conditions occur when two or more processes attempt to access shared storage.**
  - This causes the final outcome to depend on who runs first.
  - "Shared storage" can mean:
    - ✓**Global variables.**
    - ✓**Memory locations.**
    - ✓**Hardware registers.**
      - *This refers to configuration registers rather than CPU registers.*
    - ✓**Files.**
  - To understand race conditions, we will consider the example of a queue in a print spooler.

Race Conditions

- **A process that wants to print enters the name of the file into a print directory.**
- **A print daemon (printd) periodically checks the directory, prints out the next file and removes its entry.**
- **Two variables keep track of tl queue:**
  - IN: Next available slot.
  - OUT: Next file for printing.



Mutex Methods

Mechanism	Effective?	Why/Why not?	Advantages	Disadvantages
Disabling interrupts	Y	Without interrupts, there will be no process switching, hence only one process is running. However this will not work in a multi-processor environment since disabling interrupts only affects one CPU.	Simple to implement IF you have kernel privileges to disable interrupts.	Only works in single-processor environments. If a program disables interrupts and hangs, the entire system will no longer work since it cannot switch tasks.
Lock variables	N	Race condition on the lock variable.	-	-

Mutex Methods

Strict alternation	Y	Processes update the "turn" variable to allow the OTHER process to run. A race condition therefore does not affect the algorithm. Suppose this is a time-slicing OS that gives each process a certain amount of CPU time then switches to another. Process 1 is currently in the critical section (turn=1), and the OS switches to process 0. Towards the end of process 0's turn, this can happen: Process 0: Reads in turn=1, gets pre-empted. Process 1: finishes critical section, sets turn=0. Gets pre-empted or voluntarily gives up CPU time. Process 0: Sees that turn=1, loops, reads in turn=0, enters critical section. We can see that the worst case that happens in this race condition is that process 0 loops one extra time. However the mutex remains protected.	Simple to implement.	One task not in its critical section can block out another task that wants to enter the critical section (see notes). Busy waiting.

Mutex Methods

Peterson's Solution	Y	Consider process 0 wanting to enter the critical section, with neither process currently in there. **Process 0:** interested[0] is set to 1. turn is set to 0 Loads in interested[1]. Since neither process is in the CS, interested[1]=0, however process 0 is pre-empted. **Process 1:** interested[1] is set to 1. turn is set to 1 Loads in interested[0], sees it is 1, loops until it is pre-empted. **Process 0:** "Knows" from earlier on that interested[1]=0, so enters CS without looping. CS is secure.	Does not require hardware support for mutual exclusion (neither does strict alternation). Does not suffer from problem where a process in a non-critical section is able to lock out another process, since there is no implicit assumption of whose turn it is to go.	Busy waiting. Relatively difficult to understand.

Mutex Methods

Sleep/wakeup	N	This is a coordination mechanism to allow one task to sleep while a condition is not met, and another to wake it when a condition is met. There is no attempt to see if anyone is in the CS.	-	-

Semaphores

- **A semaphore is a special lock variable that counts the number of wake-ups saved for future use.**
  - A value of “0” indicates that no wake-ups have been saved.
- **Two ATOMIC operations on semaphores:**
  - DOWN, PEND or P:
    - ✓**If the semaphore has a value of >0, it is decremented and the DOWN operation returns.**
    - ✓**If the semaphore is 0, the DOWN operation blocks.**
  - UP, POST or V:
    - ✓**If there are any processes blocking on a DOWN, one is selected and woken up.**
    - ✓**Otherwise UP increments the semaphore and returns.**

Using Semaphores in the Producer/Consumer Problem

```
#define N 100                      /* number of slots in the buffer */
typedef int semaphore;             /* semaphores are a special kind of int */
semaphore mutex = 1;               /* controls access to critical region */
semaphore empty = N;               /* counts empty buffer slots */
semaphore full = 0;                /* counts full buffer slots */

void producer(void)
{
     int item;

     while (TRUE) {                /* TRUE is the constant 1 */
          item = produce_item( );  /* generate something to put in buffer */
          down(&empty);            /* decrement empty count */
          down(&mutex);            /* enter critical region */
          insert_item(item);       /* put new item in buffer */
          up(&mutex);              /* leave critical region */
          up(&full);               /* increment count of full slots */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                /* infinite loop */
          down(&full);             /* decrement full count */
          down(&mutex);            /* enter critical region */
          item = remove_item( );   /* take item from buffer */
          up(&mutex);              /* leave critical region */
          up(&empty);              /* increment count of empty slots */
          consume_item(item);      /* do something with the item */
     }
}
```

- EMPTY – # of empty slots.
- FULL – # of full slots.
- MUTEX – Prevents simultaneous access to the buffer.

Mutual Exclusion with Semaphores

- **When a semaphore's counting ability is not needed, we can use a simplified version called a "mutex".**
  - 1 = Unlocked.
  - 0 = Locked.
- **Two processes can then attempt do DOWN the semaphore.**
  - Only one will succeed. The other will block.
  - When the successful process exits the critical section, it does an UP, waking the other process up.

NUS National University of Singapore School of Computing

Problems with Semaphores

Deadlock

- **Our producer/consumer solution has a problem:**
  - If we swapped the semaphores for empty/full with the mutex semaphore, we have a potential deadlock:

```
#define N 100                         /* number of slots in the buffer */
typedef int semaphore;                /* semaphores are a special kind of int */
semaphore mutex = 1;                  /* controls access to critical region */
semaphore empty = N;                  /* counts empty buffer slots */
semaphore full = 0;                   /* counts full buffer slots */

void producer(void)
{
     int item;

     while (TRUE) {                   /* TRUE is the constant 1 */
          item = produce_item( );     /* generate something to put in buffer */
          down(&mutex);               /* decrement empty count */
          down(&empty);               /* enter critical region */
          insert_item(item);          /* put new item in buffer */
          up(&mutex);                 /* leave critical region */
          up(&full);                  /* increment count of full slots */
     }
}

void consumer(void)
{
     int item;

     while (TRUE) {                   /* infinite loop */
          down(&mutex);               /* decrement full count */
          down(&full);                /* enter critical region */
          item = remove_item( );      /* take item from buffer */
          up(&mutex);                 /* leave critical region */
          up(&empty);                 /* increment count of empty slots */
          consume_item(item);         /* do something with the item */
     }
}
```

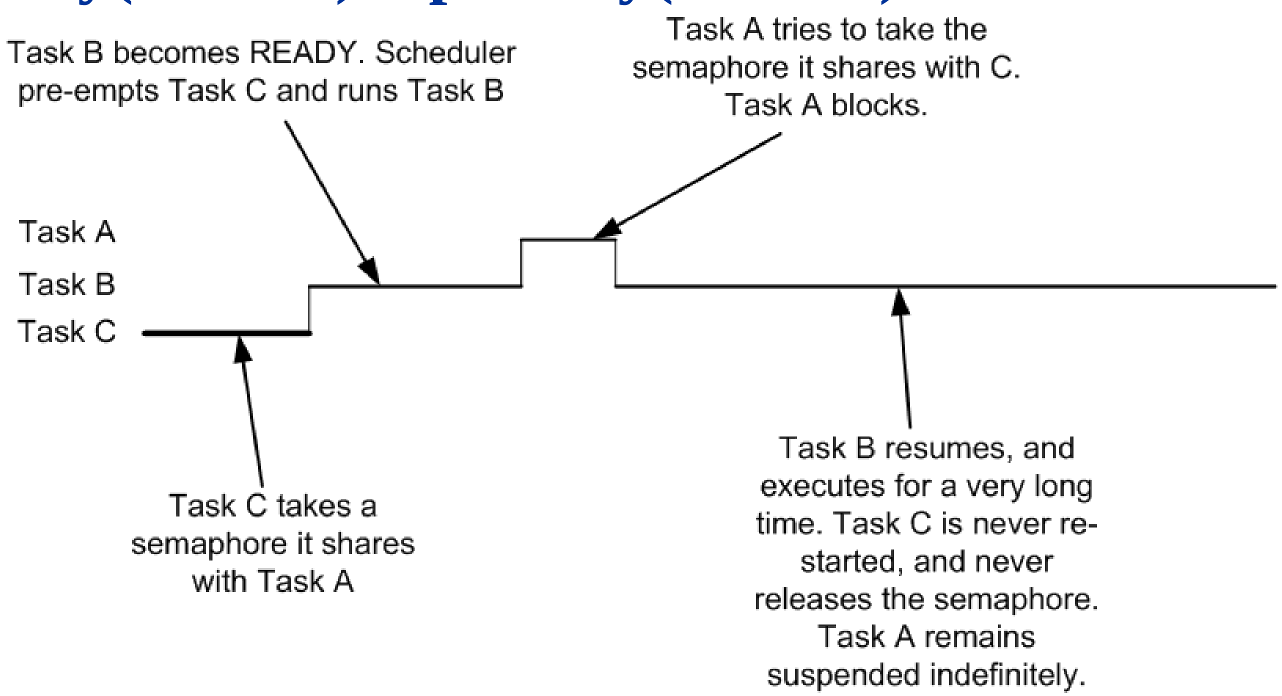
Problems with Semaphores
Deadlock

- **This can happen:**
  - Producer successfully DOWNs the mutex.
  - Producer DOWNs “empty”. However the queue is full so this blocks.
  - Consumer DOWNs mutex and blocks.
    - ✓**Consumer now never reaches the UP for “empty” and therefore cannot unblock the producer.**
    - ✓**The producer in turn never reaches the UP for mutex and cannot unblock the consumer.**
    - ✓**Deadlock!**

NUS National University of Singapore
School of Computing

Problems with Semaphores
Priority Inversion

- **In the diagram on the following page, priority(Task C) < priority(Task B) < priority(Task A).**



- **Task B effectively blocks out Task A, although Task A has higher priority!**

Monitors

- **A monitor is similar to a class or abstract-data type in C++ or JAVA:**
  - Collection of procedures, variables and data structures grouped together in a package.
    - ✓**Access to variables and data possible only through methods defined in the monitor.**
  - However, only one process can be active in a monitor at any point in time.
    - ✓**I.e. if any other process tries to call a method within the monitor, it will block until the other process has exited the monitor.**

Monitors and Condition Variables

- **Monitors achieve mutual exclusion, but we also need other mechanisms for coordination.**
  - E.g. in our producer/consumer problem, mutual exclusion alone is not enough to prevent the producer from proceeding when the buffer is full.
- **We introduce "condition variables".**
  - One process WAITs on a condition variable and blocks, until..
  - Another process SIGNALs on the same condition variable, unblocking the WAITing process.

Monitors and Condition Variables

- **Implementing the Producer/Consumer problem with semaphores and condition variables:**
  - When the buffer is full (count==N), producer will WAIT on a full condition.
  - When buffer is empty (count==0), consumer will WAIT on empty.

```
monitor ProducerConsumer
    condition full, empty;
    integer count;
    procedure insert(item: integer);
    begin
        if count = N then wait(full);
        insert_item(item);
        count := count + 1;
        if count = 1 then signal(empty)
    end;
    function remove: integer;
    begin
        if count = 0 then wait(empty);
        remove = remove_item;
        count := count − 1;
        if count = N − 1 then signal(full)
    end;
    count := 0;
end monitor;

procedure producer;
begin
    while true do
    begin
        item = produce_item;
        ProducerConsumer.insert(item)
    end
end;
procedure consumer;
begin
    while true do
    begin
        item = ProducerConsumer.remove;
        consume_item(item)
    end
end;
```

Monitors and Condition Variables

- **When a process encounters a WAIT, it is blocked and another process is allowed to enter the monitor.**
- **Problem:**
  - When there's a SIGNAL, the sleeping process is woken up.
  - We will potentially now have two processes in the monitor at the same time:
    - ✓**The process doing the SIGNAL (the signaler).**
    - ✓**The process that just woke up because of the SIGNAL (the signaled).**

Monitors and Condition Variables

- **We have 3 ways to resolve this:**
  - We require that the signaler exits immediately after calling SIGNAL.
  - We suspend the signaler immediately and resume the signaled process.
  - We suspend the signaled process until the signaler exits, and resume the signaled process only after that.

Barriers

- **A “barrier” is a special form of synchronization mechanism that works with groups of processes rather than single processes.**