IT5002

Computer Systems and Applications

Lecture 12

Process Management

colintan@nus.edu.sg

National University of Singapore

School of Computing

Introduction

- **In this lecture we will look at:**
  - The difference between a program and a process.
  - Interrupts and How They Work
  - Process States.
  - How to run Multiple Processes in a Single CPU
  - Process Creation, termination and zombies.

Program vs. Process

- **A program consists of:**
  - Machine instructions (and possibly source code).
  - Data
  - A program exists as a file on the disk. E.g. command.exe, winword.exe
- **A process consists of:**
  - Machine instructions (and possibly source code).
  - Data.
  - Context
  - Exists as instructions and data in memory.
  - MAY be executing on the CPU.

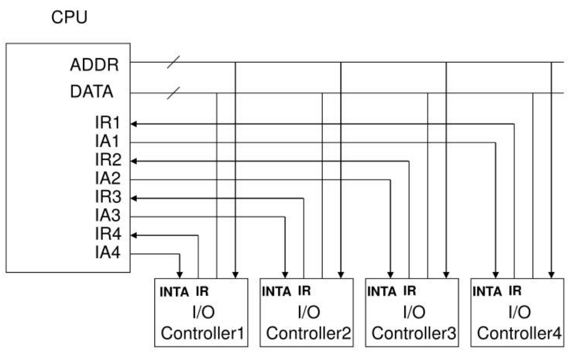
Program vs. Process

- **A single program can produce multiple processes.**
  - E.g. chrome.exe is a single program.
  - But every tab in Chrome is a new process!

Interrupts

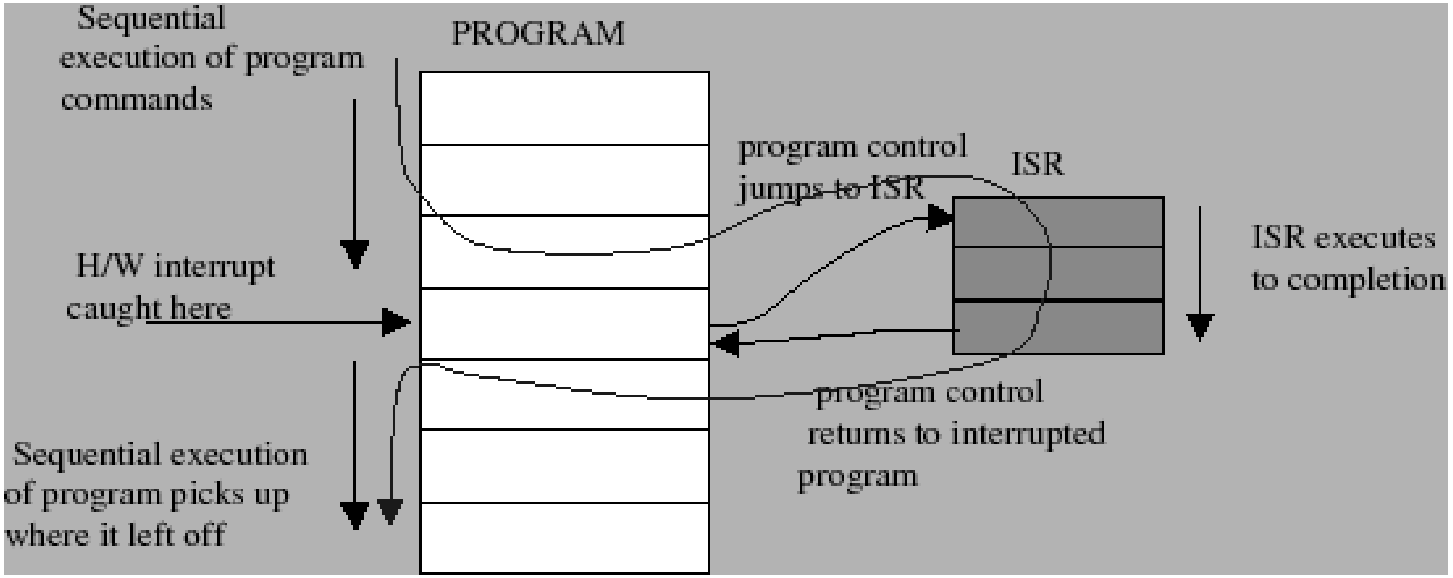
- **When a device needs attention from the CPU, it triggers what is called an "interrupt":**
  - Each device is connected to the CPU via an input called an "Interrupt Request" or IRQ line.
    - ✓**When a device needs attention, it pull the IRQ line HIGH or LOW (dependin on whether line is "active high" or "active low")**

Multiple Interrupt Lines



Interrupts

- This line is checked at the end of the WB stage in the CPU Execution Cycle.
- Diagrams below show the CPU's program execution flow when an interrupt occurs:



Interrupts

- If line has been pulled, CPU interrupts the code it is currently running to run code to attend to the device:
  - ✓**Current PC is pushed onto the stack.**
  - ✓**CPU consults an "interrupt vector table" to look for the address of the "interrupt service routine" or ISR – a small bit of code that will read/write/tend to the device.**
  - ✓**This address is loaded into PC, and the CPU starts executing the handler.**
  - ✓**When the handler exits, the previous PC value is popped off the stack and back into PC, and execution resumes at the interrupted point.**

Interrupts

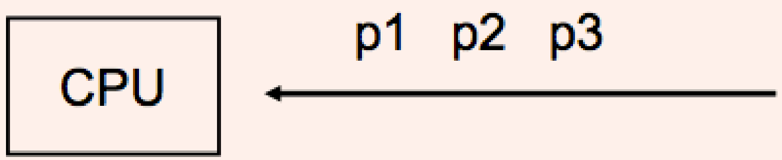
- **The CPU asserts the interrupt acknowledge (IA) line to tell the device that its request has been handled.**
  - Sometimes the CPU will de-assert the IRQ line instead of employing a separate IA line.
- **Interrupts are key to allowing us to run multiple processes on a CPU:**
  - A hardware device called a "timer" will interrupt the CPU every millisecond.
  - Interrupt handler will switch to a new process.

Execution Modes

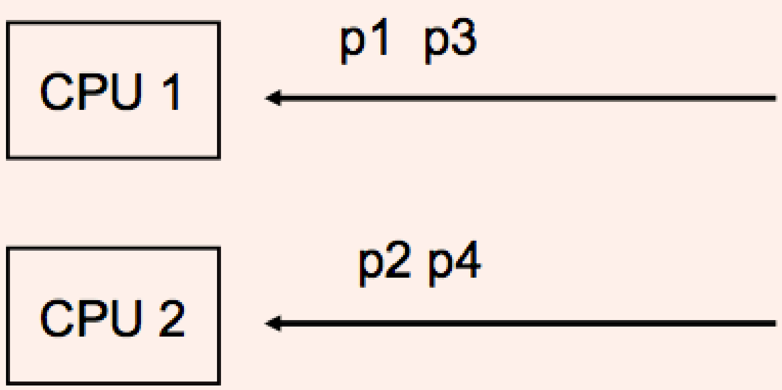
- **Programs usually run sequentially.**
  - Each instruction is executed one after the other.
- **Having multiple cores or CPUs allow parallel ("concurrent") execution.**
  - Streams of instructions with no dependencies are allowed to execute together.
- **A multitasking OS allows several programs to run "concurrently".**
  - Interleaving, or "time-slicing".

Execution Modes

- 1 CPU: timesliced execution of tasks



- multiprocessor: timeslicing on n CPUs



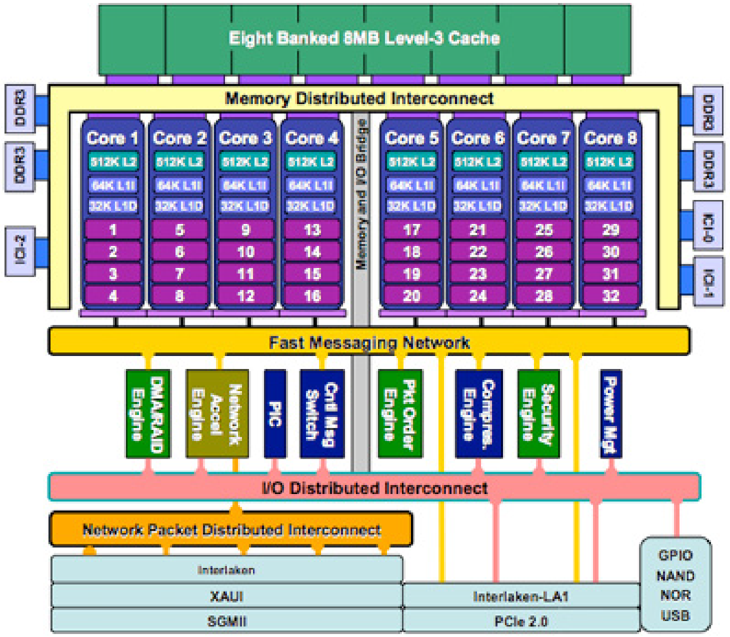
Note: we mostly assume no. processes ≥ no. of CPU otherwise can have idle task

Task Management

PROCESSES AND PROCESS MANAGEMENT

The Process Model

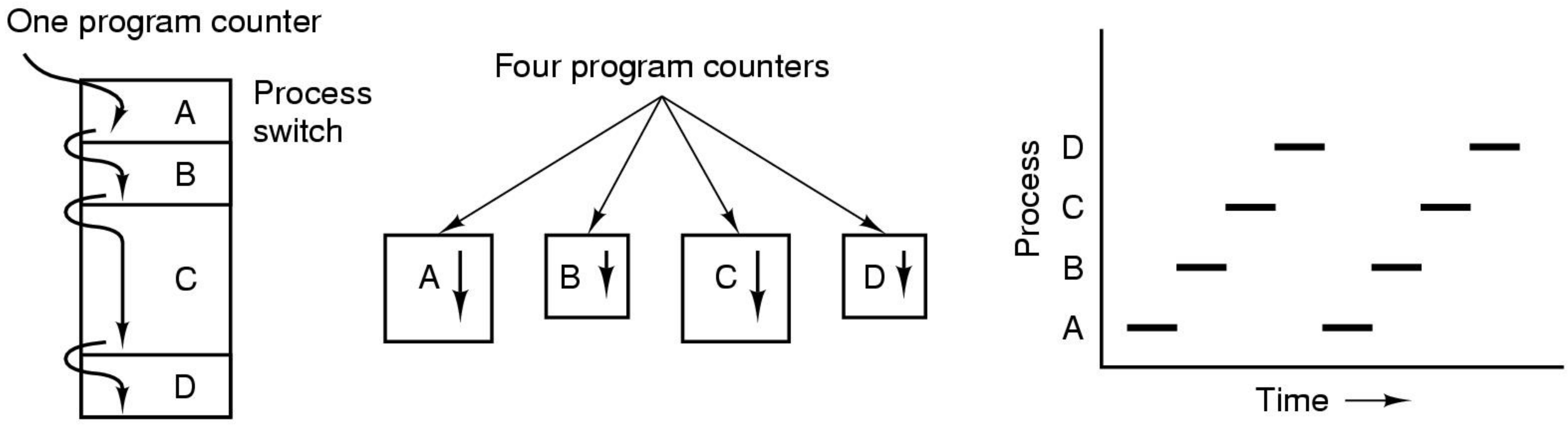
- **In this lecture we will assume a single processor with a single core.**
  - This is a legitimate assumption because in general the number of processes >> the number of cores.
  - So each core must still switch between processes.



XLP832 Processor Block Diagram

The Process Model

- **Since we have only a single-core single processor:**
  - At any one time, at most one process can execute.
  - Figures (a) to (c) below illustrate what happens:



(a) (b) (c)

The Process Model

- **Figure (b) shows what "appears" to be happening in a single processor system running multiple processes:**
  - There are 4 processes each with its own program counter (PC) and registers.
  - All 4 processes run independently of each other at the same time.

The Process Model

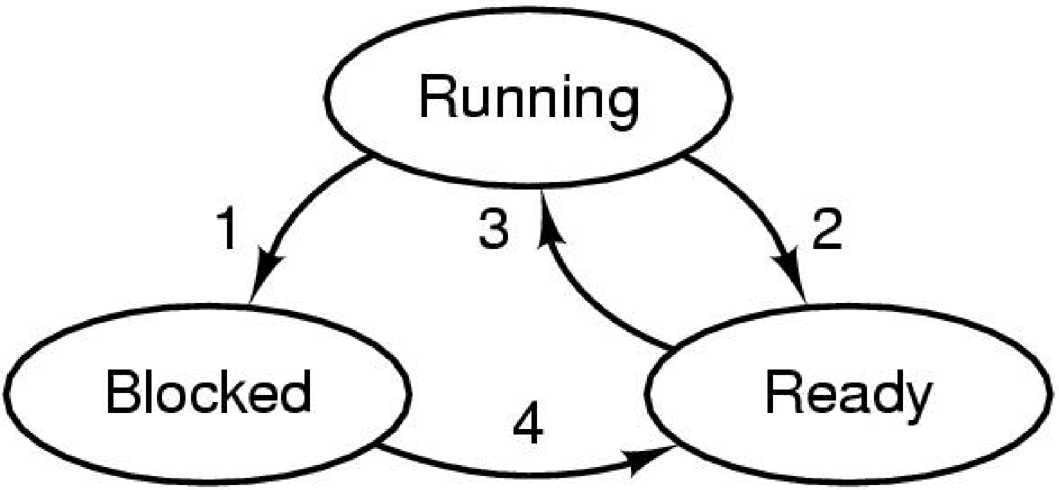
- **Figure (a) shows what actually happens.**
  - There is only a single PC and a single set of registers.
  - When one process ends, there is a “context switch” or “process switch”:
    - ✓**PC, all registers and other process data for Process A is copied to memory.**
    - ✓**PC, register and process data for Process B is loaded and B starts executing, etc.**
  - Figure (c) illustrates how processes A to D share CPU time.

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

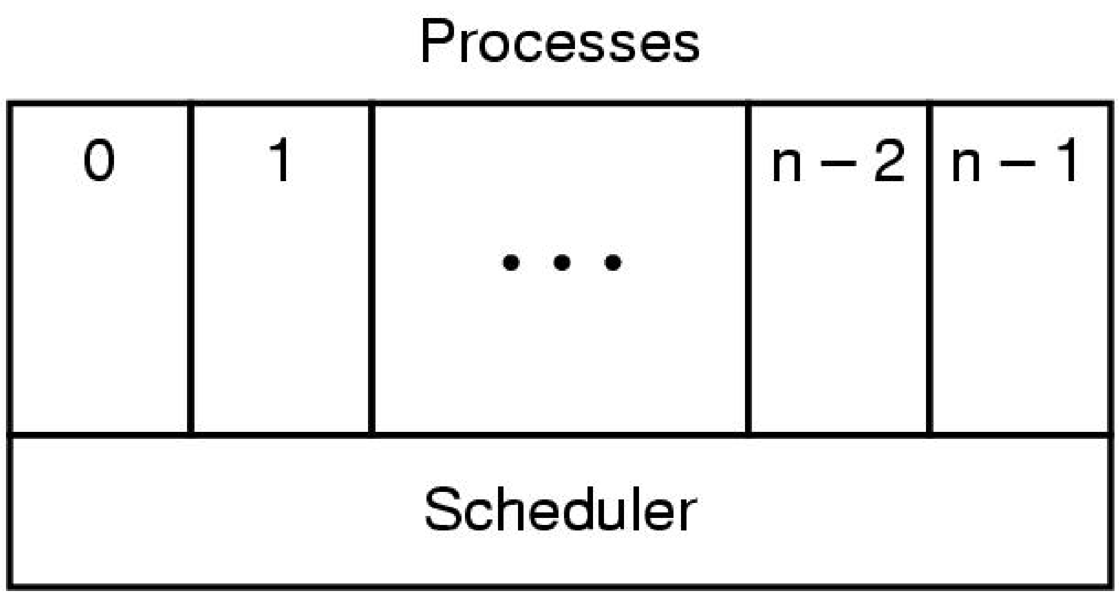
- **The diagram below shows the 3 possible states and the transitions between them.**



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

Process States

- **The figure below shows how the processes are organized.**
  - The lowest layer selects (schedules) which process to run next.
    - ✓**This is subject to "scheduling policies" which we will look at in a later lecture.**



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	

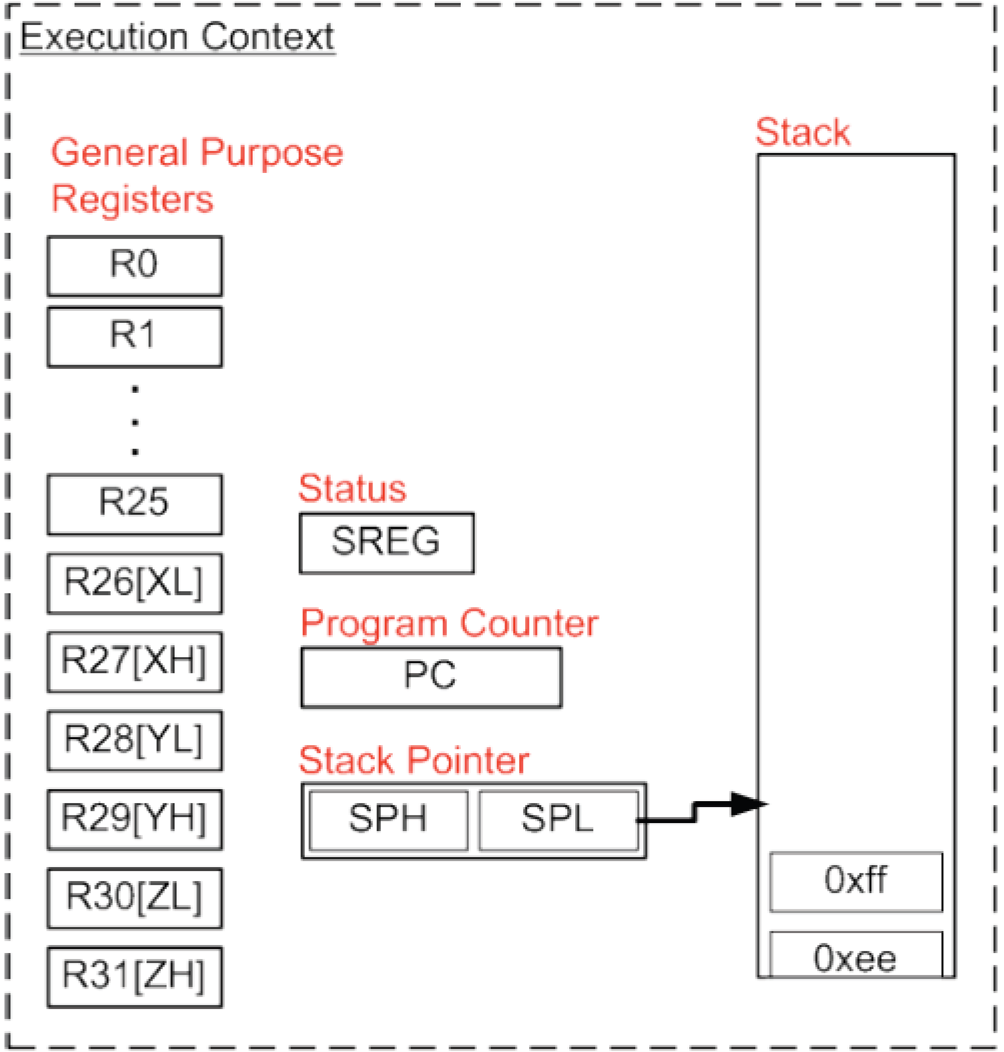
Switching between Processes

- **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 IT5003
  - We use the term “task” instead of process here but means the same thing.
- **The diagram shows the complete Atmega context.**
  - Registers R0-R31, PC.
  - Status register SREG.
  - Stack pointer SPH/SPL.



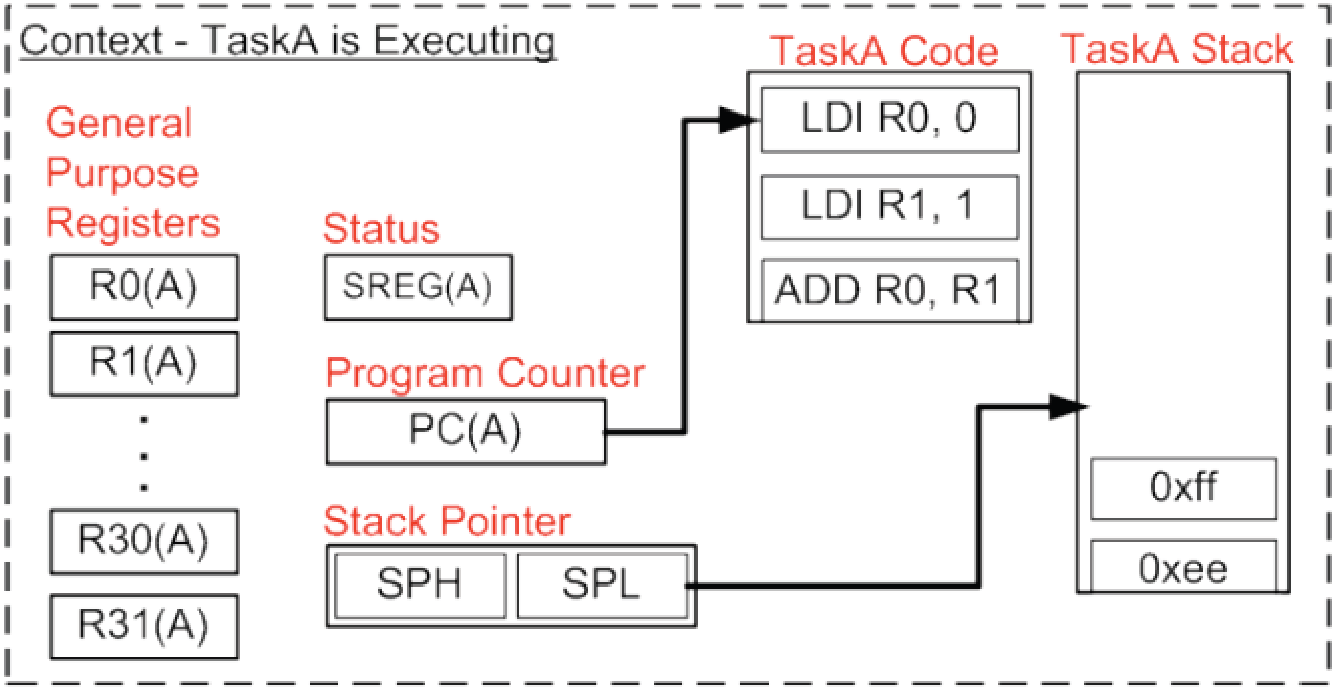
Context Switching on the FreeRTOS Atmega Port

- **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
);
```

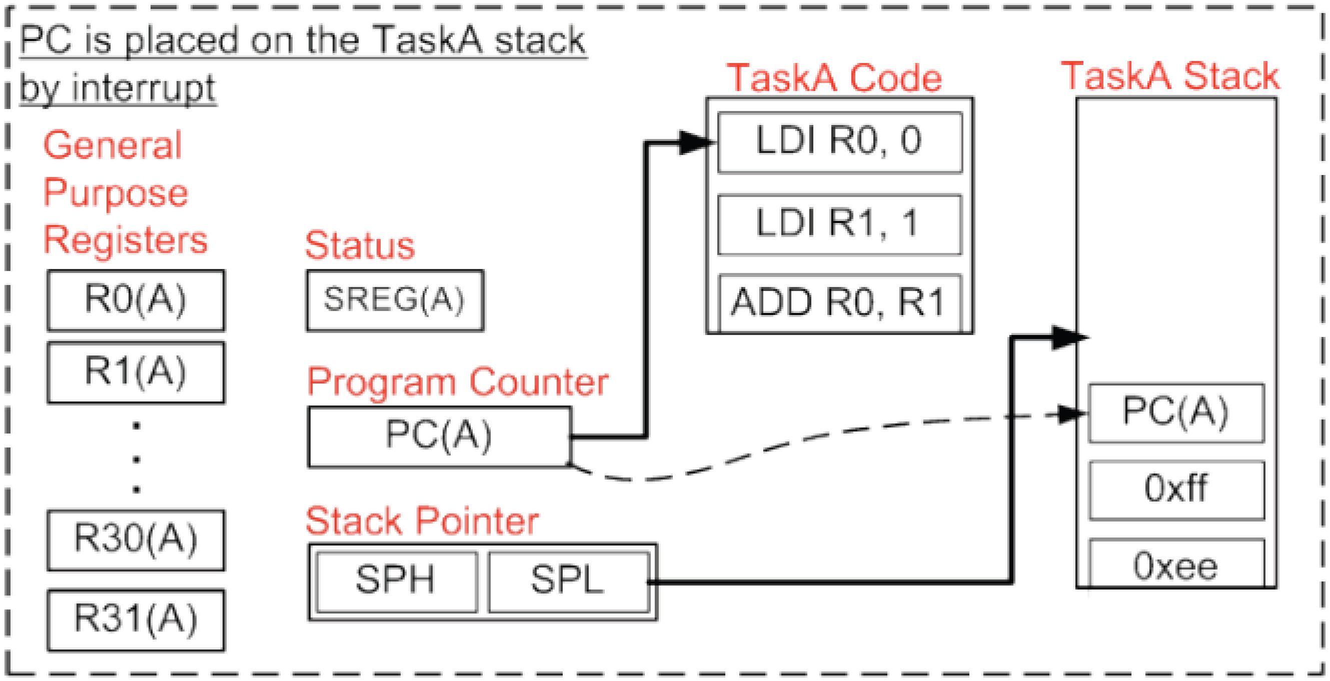
Context Switching on the FreeRTOS Atmega Port

- **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.



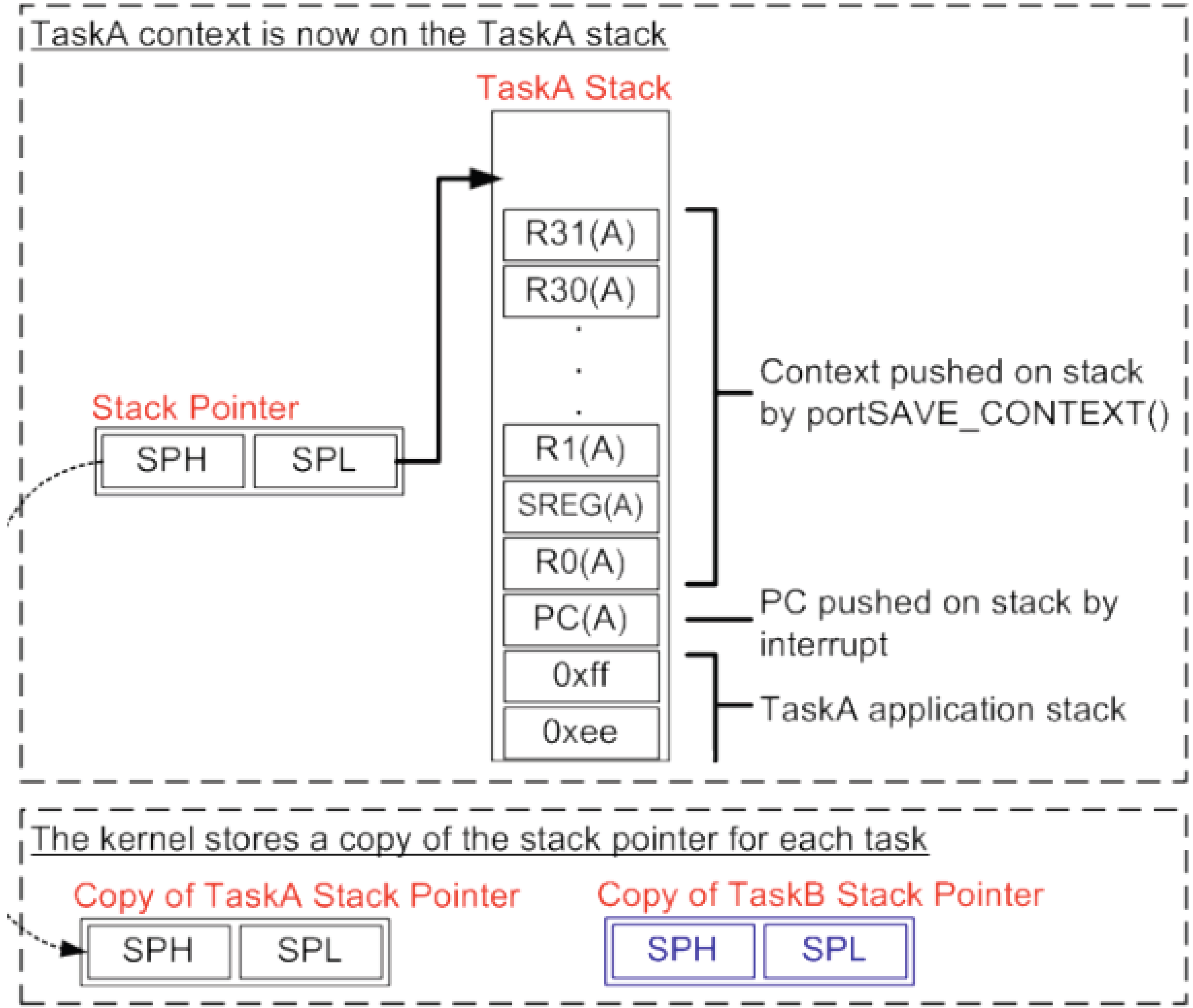
Context Switching on the FreeRTOS Atmega Port

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



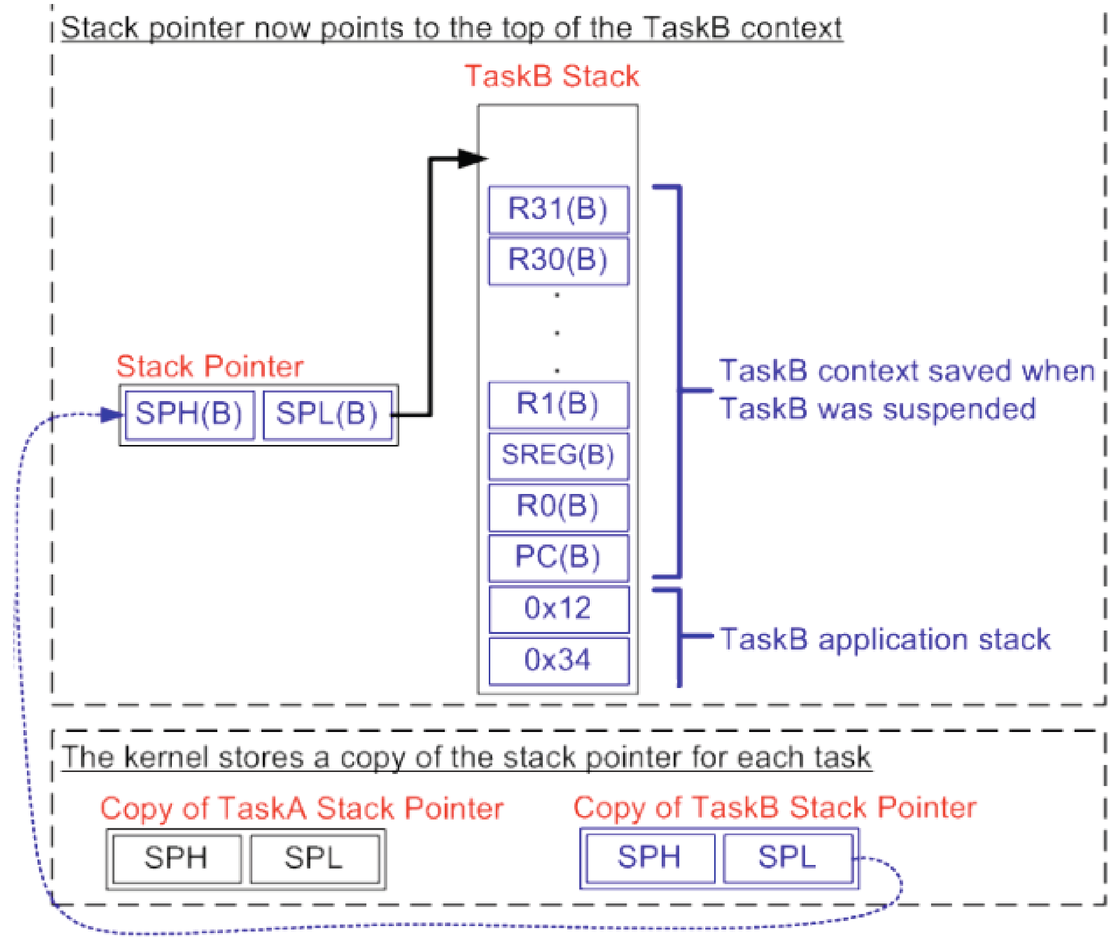
Context Switching on the FreeRTOS Atmega Port

- **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.



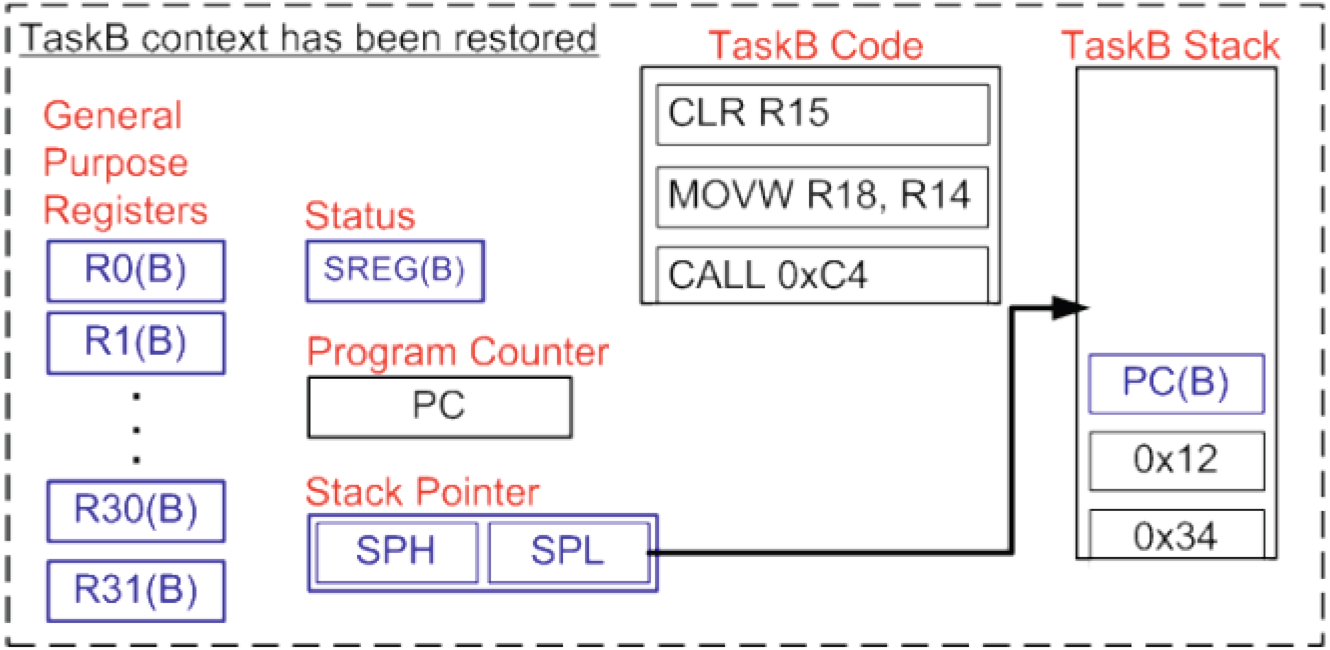
Context Switching on the FreeRTOS Atmega Port

- **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

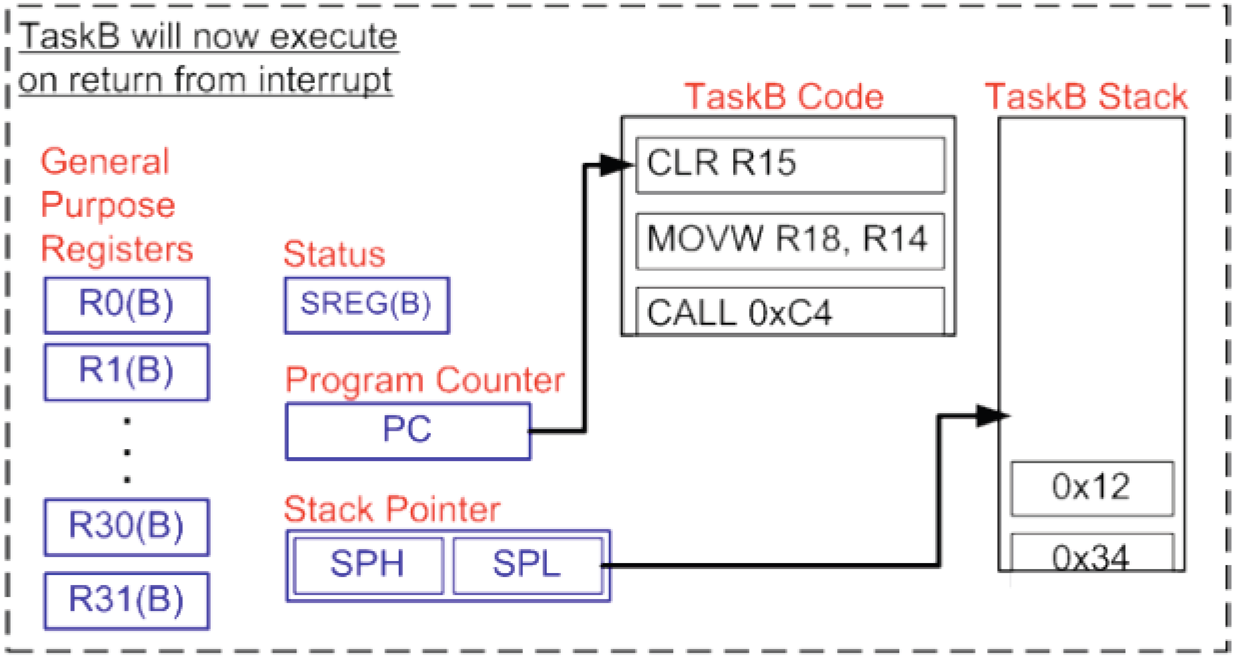
- **The reverse operation is portRESTORE_CONTEXT. The stack pointer for the process being restored must be in pxCurrentTCB.**

```
#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

- **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".**
    - ✓**…**

NUS National University of Singapore School of Computing

Process Creation

- **A process can be created in Python by using a fork() call:**

```
# Python code to create child process
import os

def parent_child():
    n = os.fork()

    # n greater than 0  means parent process
    if n > 0:
        print("Parent process and id is : ", os.getpid())

    # n equals to 0 means child process
    else:
        print("Child process and id is : ", os.getpid())

# Driver code
parent_child()
```

- **The creating process is called a "parent" process, while the created process is called a "child" process.**

Process Creation

- **When you run a program in your OS shell, the shell uses the OS to create a new process, then run the program in the new process.**
- **In Python this is done by using subprocess.call:**

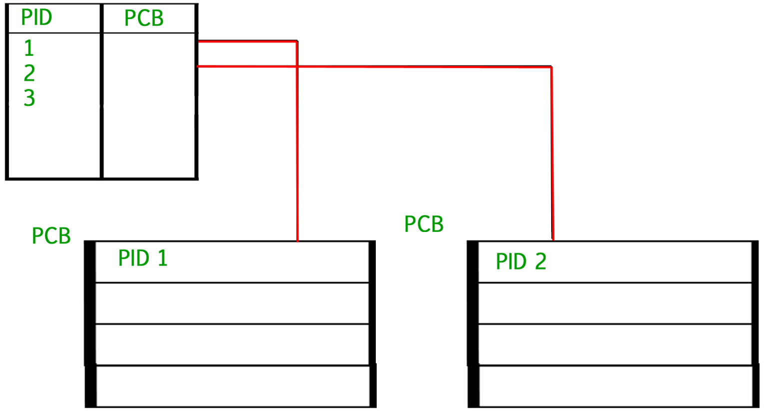
```
import subprocess

subprocess.call(["ls", "-lha"])
```

  - The launched is thus a child process of the shell.
- **Ultimately, all UNIX processes are children of "init", the main starting process in UNIX.**

Process Control Blocks

- **When a process is created, the Operating System also creates a data structure to maintain information about that process:**
  - Called a "Process Control Block" (PCB) and contains:
    - ✓**Process ID (PID)**
    - ✓**Stack Pointer**
    - ✓**Open files**
    - ✓**Pending signals**
    - ✓**CPU usage**
    - ✓**…**



Process table and process control block

  - PCB is stored in a table called a "Process Table".
    - ✓**One Process Table for entire system.**
    - ✓**One PCB per process.**

Terminating A Process

- **When a process terminates:**
  - Most resources like open files, etc., can be released and returned to the system.
  - However the PCB is retained in memory:
    - ✓ **Allows child processes to return results to the parent.**
  - Parent retrieves the results using a "wait" function call, afterwhich the PCB is released.
- **What if the parent never calls "wait"?**
  - PCB remains in memory indefinitely.
  - Child becomes a "zombie". Eventually process table will run out of space and no new process can be created.