Integrated Technical Education Cluster
At AlAmeeria

E-626-A
Real-Time Embedded Systems (RTES)

Lecture #10
Context Switching & Performance Optimization

Instructor:
Dr. Ahmad El-Banna

Spring 2015

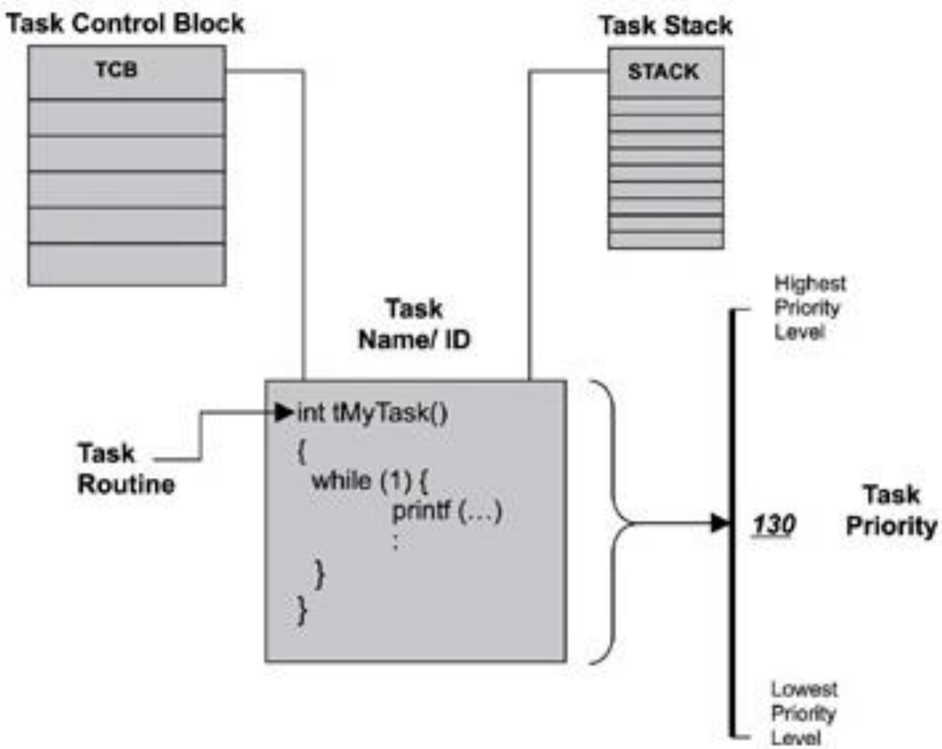
Agenda

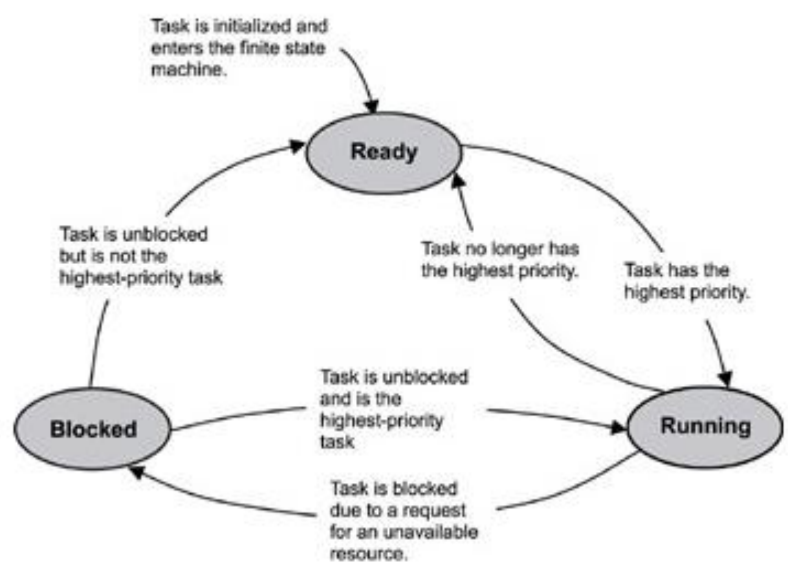
- Introduction to Context Switching
- Software Optimization levels
- Optimization Trade-offs
- Pareto Principle

Context Switch

- A **context switch** (a process switch or a task switch) is the **switching of** the central processing unit, **CPU**, **from one process** or thread **to another**.
- A **process** (a task) is an executing (i.e., **running**) **instance** of a program.
- In **Linux**, threads are **lightweight processes** that can run in **parallel** and **share** an address space (i.e., a range of **memory** locations) and other **resources with** their **parent** processes (i.e., the processes that created them).

Task Parameters and States





Context Switching

- A **context** is the **contents of a CPU's registers** and **program counter** at any point in time.
- **Context switching** can be described in slightly more detail as the **kernel** (i.e., the core of the operating system) **performing** the **following activities** with regard to processes (including threads) on the CPU:
  - (1) **suspending** the **progression of one process** and **storing** the CPU's state (i.e., the **context**) for that process somewhere in memory,
  - (2) **retrieving** the **context of the next process from memory** and restoring it in the CPU's registers and
  - (3) returning to the location indicated by the program counter (i.e., **returning to the line of code at which the process was interrupted)** in order to resume the process.

Cost of Context Switching

- Context switching is generally **computationally intensive**.
- That is, it **requires considerable processor time**, which can be on the order of *nanoseconds for each of the tens or hundreds of switches per second*.
- Thus, context switching represents a **substantial *cost*** to the system in terms of CPU time and can, in fact, be **the most costly operation on an operating system**.
- Consequently, a major **focus in the design of operating systems** has been to **avoid unnecessary context switching** to the extent possible.
- However, this has **not** been **easy** to accomplish **in practice**.

PERFORMANCE OPTIMIZATION

Program Optimization

- Program optimization or **software optimization** is the process of **modifying a software system to make some aspect of it work more efficiently** or use fewer resources.

- In general, a computer **program** may be **optimized** so that :
  - it **executes** more **rapidly**, or
  - is capable of operating with **less memory** storage or other resources, or
  - draw **less power**.

Levels of optimization

- Optimization can occur at a **number of levels**.
- Typically **the higher levels have greater impact**, and are **harder to change later** on in a project, requiring significant changes or a **complete rewrite if they need to be changed**.
- Thus optimization can typically proceed via **refinement from higher to lower,** with initial gains being larger and achieved with less work, and later gains being smaller and requiring more work.

Levels of optimization..

- **Design level**
  - At the highest level, the design may be optimized to make **best use of the available resources**, given goals, constraints, and expected use/load.
- **Algorithms and data structures**
  - Given an overall design, a good choice of efficient algorithms and data structures, and **efficient implementation of these algorithms** and data structures comes next.
- **Source code level**
  - Beyond general algorithms and their implementation on an abstract machine, **concrete source code level choices** can make a significant difference.

Levels of optimization…

- **Build level**
  - **Between the source and compile level**, directives and **build flags** can be used to tune performance options in the source code and compiler respectively.
- **Compile level**
  - Use of an **optimizing compiler** tends to ensure that the executable program is optimized at least as much as the compiler can predict.
- **Assembly level**
  - At the lowest level, **writing code using an assembly language**, **designed for a particular hardware platform** can produce the most efficient and compact code if the programmer takes advantage of the full repertoire of machine instructions.
- **Run time**
  - **Just-in-time compilers** can produce **customized machine code based on run-time data**, at the cost of compilation overhead.

Trade-offs

- Optimization will generally focus on **improving just one or two aspects of performance**: execution time, memory usage, disk space, bandwidth, power consumption or some other resource.
- This will usually require a **trade-off** — where **one factor is optimized at the expense of others**.
- For example, increasing the size of cache improves runtime performance, but also increases the memory consumption. Other common trade-offs include code clarity and conciseness.

- There are instances where the programmer performing the optimization must decide to **make the software better for some operations but at the cost of making other operations less efficient**.
- These trade-offs may sometimes be of a **non-technical nature**

When to optimize?

- Optimization can **reduce readability** and add code that is used only to improve the performance.
- This may **complicate programs** or systems, making them **harder to maintain and debug**.
- As a **result**, optimization or performance tuning is often performed **at the end of the development stage**.

Bottlenecks

- Optimization may include **finding a bottleneck in a system** – a component that is **the limiting factor on performance**.
- In terms of code, this will often be a hot spot – a **critical part of the code that is the primary consumer of the needed resource** – though it can be **another factor**, such as **I/O latency** or **network bandwidth**.

Pareto Principle

- <u>Definition:</u>
  - The **Pareto principle** (also known as the **80–20 rule**, the **law of the vital few,** and the **principle of factor sparsity**) states that, for many events, roughly 80% of the effects come from 20% of the causes.

Distribution of world GDP, 1989[6]

Quintile of population	Income
Richest 20%	82.70%
Second 20%	11.75%
Third 20%	2.30%
Fourth 20%	1.85%
Poorest 20%	1.40%

- In <u>**computer science**</u>, **resource consumption often follows a form of power law distribution**, and the Pareto principle can be applied to resource optimization by observing that 80% of the resources are typically used by 20% of the operations.
- In <u>**software engineering**</u>, it is often a better **approximation** that **90%** of the execution time of a computer program is spent executing 10% of the code (known as the **90/10 law** in this context).

Pareto Principle

- <u>In Software:</u>
- In computer science and engineering control theory, such as for electromechanical energy converters, **the Pareto principle can be applied to optimization efforts**.
- For **example**, Microsoft noted that by fixing the top 20% of the most-reported bugs, 80% of the related errors and crashes in a given system would be eliminated.
- In **load testing**, it is common practice to estimate that 80% of the traffic occurs during 20% of the time.
- In **software engineering**, Lowell Arthur expressed a **corollary principle**: "20 percent of the code has 80 percent of the errors. Find them, fix them!"

- For more details, refer to:
  - **Program Optimization**, wiki.
  - **Context Switch Definition**, Online tutorial, http://www.linfo.org/context_switch.html
  - **Pareto Principle Basics**, wiki.
- The lecture is available online at:
  - http://bu.edu.eg/staff/ahmad.elbanna-courses/12134
- For inquires, send to:
  - ahmad.elbanna@feng.bu.edu.eg