A Review of Multi-Tasking Strategies

Volkan Yazıcı
Bilkent Univ. Dept. of Comp. Eng.
vyazici@cs.bilkent.edu.tr

CS 541 (Chip Multiprocessors) HW#3 Report, Ankara, 2009

Abstract

- *Multi-tasking strategies* (processes, threads) on *unicore* and *multicore* architectures for various *operating systems*.
- Latest improvements in the area.
- Examples from modern operating systems.

A Brief History

Early Days

1. Giving program instructions to the instructor.
2. Results on punchards after some hours.
3. Was not a problem!

With Gigantic Processing Power

- I/O is the bottleneck.
- New techniques (time sharing systems, etc.) needed for processor utilization.
- Multi-users, multi-processes, multi-jobs... *Multi-tasking*!

Multi-Tasking Strategies

- **Multiprogramming Systems** - Task keeps running until it performs an operation that requires waiting for an external event.
- **Time-Sharing Systems** - Running task is required to relinquish the processor after a specific period of time.
- **Real-Time Systems** - Some waiting tasks are guaranteed to be given the processor when an external event occurs.

Processes

Process

- Base primitive forming *task*s.
- Kernels are developed with both unicore and multicore support.

Capabilities

- Processes can be simulated to be running concurrently.
- Individual processes can be *physically* run on distinct processors.

The Good, The Bad, The Ugly

Advantages

- Concurrency.
- True paralellization!

Disadvantages

- When executing multiple processes on the physically same processor
  - A significant amount of state information. (e.g. file handles, user permissions, etc.)
  - Context switching related overheads.
  - TLB (Translation Lookaside Buffer) flushing etc.
- Separate address spaces, communication (IPC, RPC, etc.) overhead.

Threads and Shared State

Basic Concept

- Threads are lightweight processes.
- Implementation differs from one operating system to another.

Advantages

- Shared address space. (No IPC overhead.)
- Cheaper context switches.

Implementation Strategies

- **1:1** - Threads are one-to-one represented by schedulable entities in the kernel.
  - Easy to implement.
- **N:M** - N threads are mapped to M schedulable entities in the kernel.
  - Easy scheduling, improved context switching performance.
  - Complex implementation – requires work in kernel and user level.
  - *Scheduler Activations* (SA) is a threading mechanism that implements N:M strategy.
- **N:1** - N threads are mapped to a single schedulable entity in the kernel.
  - Clearly fast context switching.
  - Oblivious to hardware capabilities.
  - Generally adopted by programming language implementations.

Thread Scheduling

- **Cooperative** - Threads themselves inform the operating system to relinquish the control.
  - Poorly designed programs can block whole system.
  - Rarely used in modern systems.
- **Preemptive** - Kernel decides when to switch the control between threads.
  - Involves an interrupt mechanism.
  - All processes will get some amount of CPU time at any given time.

Plan 9 Processes

- A single class of process. (Process = Thread = ...)
- Fine control of the process resources. (Memory, file descriptors, etc.)
- Technique is feasible since...
    - Efficient system call interface.
    - Cheap process creation and scheduling.

Erlang Processes

- A functional programming language designed at the Ericsson Computer Science Laboratory.
- Extremely lightweight processes.
- No shared memory and communicate by asynchronous message passing. (Silver bullet in distributed computing!)
- Very large numbers of concurrent processes.
- Process spawning and scheduling are managed by the Erlang.

Fibers

- User-level threads are working in a cooperative way. (No preemption!)
- Fibers *yield* themselves to run another fiber while executing.
- Actually everything is sequential. (Hence no need for thread-safety, locking, etc.)
- Cannot benefit from benefit from multi-core, multi-processor architectures.

Questions?