Process Execution in Solaris on a Multicore System

Solaris Threading Architecture

Oracle Solaris implements a four-level threading model that differs from traditional operating systems. This architecture provides efficient mapping between application threads and hardware execution units on multicore systems.

The Four Levels

Processes

Processes serve as containers for all execution resources. Each process contains a private address space isolated from other processes, separate user and kernel mode stacks, and a Process Control Block (PCB) storing critical information including process ID, credentials, open file descriptors, signal handlers, and resource limits. All threads within a process share the same memory space and file handles, enabling efficient inter-thread communication through shared memory.

User-Level Threads (ULTs)

User-level threads are created and managed entirely in user space by threading libraries such as pthreads. The kernel has no knowledge of these threads - they exist only within the application's runtime library. This approach offers fast context switching since no kernel involvement is required. Thread creation, destruction, and synchronization occur without system call overhead.

However, a significant limitation exists: when one ULT makes a blocking system call (e.g., disk read), the entire process blocks from the kernel's perspective. Since the kernel only sees the process, not individual ULTs, all user threads must wait even though other threads could potentially continue executing.

Lightweight Processes (LWPs)

Solaris addresses the blocking limitation through Lightweight Processes. LWPs bridge user threads and kernel threads, providing kernel visibility while maintaining shared memory spaces. Each LWP can be scheduled independently - if one blocks on I/O, others continue running.

Three mapping models exist: - Many-to-one: all ULTs on one LWP (poor parallelism) - One-to-one: each ULT has its own LWP (maximum parallelism, high overhead) - Many-to-many: ULTs multiplexed onto an LWP pool (optimal balance)

Solaris primarily uses the many-to-many model, allowing dynamic LWP allocation based on application behavior.

Kernel Threads

Kernel threads represent the actual schedulable entities managed by the dispatcher. A one-to-one correspondence exists between LWPs and kernel threads. These threads enable true parallelism as different kernel threads execute simultaneously on different cores. Each maintains its own kernel stack, register set, and scheduling priority. The dispatcher schedules kernel threads onto physical CPU cores.

Execution Flow

Process Creation

When fork() is called, the kernel allocates a new PCB with a unique process ID, establishes an address space, and creates an initial LWP with its kernel thread. The process enters the ready state awaiting scheduling.

Thread Creation

Inside a process, pthread_create() causes the thread library to create a user-level thread. Based on current conditions, the library may request a new LWP from the kernel, which then creates the LWP and corresponding kernel thread. This kernel thread joins the dispatcher's ready queue.

Multicore Execution

Each CPU core runs an independent dispatcher that selects kernel threads from the ready queue. At any moment, Core 1 might execute a thread from Process A, Core 2 a thread from Process B, Core 3 another thread from Process A, and Core 4 a thread from Process C. True parallelism occurs as threads from the same process run simultaneously on different cores.

When a kernel thread's time slice expires, context switching occurs and another thread is selected. If a kernel thread blocks awaiting I/O, it moves to blocked state and is removed from the CPU. The dispatcher immediately selects another kernel thread - possibly from the same process. This prevents one blocking thread from halting the entire process.

Blocking Operations

Consider Process A with 4 ULTs mapped to 3 kernel threads on different cores. When one ULT makes a read() system call, its kernel thread blocks and enters blocked state. The remaining kernel threads continue executing on their cores - the process as a whole maintains progress.

Upon I/O completion, a hardware interrupt triggers. Solaris converts interrupts into high-priority threads for handling. The blocked kernel thread returns to ready state, and the dispatcher reschedules it when a core becomes available. The ULT resumes from where it stopped. This contrasts sharply with single-threaded systems where the entire process blocks during I/O.

Multiprocessing Features

Kernel Preemption

Solaris implements a fully preemptable kernel where kernel threads can be interrupted even during kernel code execution. Higher-priority threads can immediately displace lower-priority ones, enabling responsive real-time performance.

Interrupt Handling

Interrupts are converted into high-priority kernel threads rather than using traditional mechanisms. The dispatcher schedules these interrupt threads on any available core. This approach eliminates special interrupt masking code and enables normal thread synchronization primitives, while distributing interrupt processing across all cores for better load balancing.

Scheduler Activations

When an LWP blocks, the kernel performs an "upcall" to the thread library. The library then immediately schedules another ULT onto a different LWP, maintaining application concurrency even during blocking operations.

Example: Web Server on 4-Core System

A multithreaded web server handling 10 concurrent requests demonstrates the architecture:

1. **Initialization**: Server creates main process with PCB and address space. Kernel creates initial kernel thread.
2. **Thread Pool**: Application calls pthread_create() ten times, creating 10 ULTs. Thread library requests 4 LWPs (matching core count). Kernel creates 4 LWPs with corresponding kernel threads.

3. **Thread Mapping**: ULTs 1-3 multiplex onto LWP 1; ULTs 4-5 onto LWP 2; ULTs 6-8 onto LWP 3; ULTs 9-10 onto LWP 4.
4. **Parallel Execution**: Core 0 executes kernel thread 1 (running ULT 1), Core 1 executes kernel thread 2 (ULT 4), Core 2 executes kernel thread 3 (ULT 6), Core 3 executes kernel thread 4 (ULT 9).
5. **I/O Handling**: ULT 1 reads from disk. Its kernel thread blocks. Thread library immediately schedules ULT 2 onto the same LWP. Core 0 continues without pause while other cores maintain execution.
6. **I/O Completion**: Interrupt thread handles disk completion. Blocked kernel thread returns to ready state. Dispatcher reschedules it on next available core. ULT 1 resumes execution.
7. **Load Balancing**: Dispatcher monitors core utilization and migrates kernel threads between cores to maintain balance.

Performance Analysis

Advantages: - Efficient multicore scalability - Thousands of ULTs with manageable kernel thread count - I/O blocking isolation (process continues executing) - Tunable ULT-to-LWP ratios - Real-time responsiveness via preemptable kernel - Efficient memory sharing with independent execution

Overhead: - Context switch costs: ULT (~100ns), kernel thread (~1-5us), process (~10-100us) - Memory per ULT: 8-16 KB; per kernel thread: 24-32 KB; per process: 100+ KB - Synchronization: fast for ULTs (user space), slower across LWPs (kernel involvement)

Conclusion

The four-level architecture (Process $\rightarrow$ ULT $\rightarrow$ LWP $\rightarrow$ Kernel Thread) separates resource ownership, application concurrency, kernel visibility, and hardware execution. When applications create multiple processes and threads on multicore systems, the dispatcher distributes kernel threads across CPU cores for true parallel execution. The preemptable kernel and interrupt-as-thread model enhance responsiveness, making Solaris effective for high-performance concurrent server applications.