Solaris

CS-4448 – Operating Systems

Group Members:

Tran Long Vu – 1677501

Tran The Hung – ...

Overview of Program Execution in Solaris

In Solaris, program execution is based on threads rather than entire processes. A process mainly provides a protected environment that contains memory space, open files, and shared resources. The actual execution of instructions on the CPU is performed by threads. This design allows Solaris to efficiently utilize multicore processors by running multiple threads in parallel.

When a program generates several processes and several threads, Solaris manages them using a combination of processes, threads, Lightweight Processes (LWPs), and kernel threads. These components work together to ensure that execution is fair, responsive, and scalable.

Process Creation and Initialization

When a program starts, Solaris creates a new process and assigns it a unique Process ID (PID). This process initially contains at least one thread, known as the main thread. The process itself does not execute code directly; instead, it acts as a container that holds the resources required by its threads.

If the program creates additional processes using the `fork()` system call, Solaris duplicates the parent process. Each child process receives its own PID and its own execution context. After creation, these processes enter the READY state, meaning they are prepared to run but are waiting for CPU time.

Thread Creation and Lightweight Processes (LWPs)

Within each process, the program may create multiple threads, for example to perform different tasks concurrently. In Solaris, each user-level thread is associated with

a Lightweight Process (LWP). An LWP represents the execution context that the kernel can schedule.

This one-to-one mapping between threads and LWPs is important because it allows each thread to be independently scheduled by the kernel. As a result, different threads from the same process can run at the same time on different CPU cores, which improves performance on multicore systems.

Scheduling Classes and Priority Management

Solaris uses scheduling classes to determine how threads are scheduled. Most user applications run in the Time-Sharing (TS) class, which aims to balance fairness and responsiveness. In this class, thread priorities are adjusted dynamically based on recent CPU usage.

For tasks that require strict timing guarantees, Solaris provides the Real-Time (RT) scheduling class. Threads in this class have fixed priorities and can preempt time-sharing threads. System-level kernel threads usually run in the System (SYS) class.

When selecting a thread to run, the scheduler always chooses the READY thread with the highest global priority, regardless of its scheduling class.

Execution on a Multicore System

On a multicore system, Solaris can run multiple threads simultaneously. Each CPU core selects a runnable thread and executes it independently. Threads from the same process or from different processes may be running at the same time on different cores.

This parallel execution model allows programs that create multiple processes and threads to fully utilize multicore hardware. For example, one thread may be performing computation while another thread waits for input, and both can progress efficiently without blocking each other.

Context Switching and Thread States

During execution, a thread moves between several states. When a thread is READY, it is waiting to be scheduled. When it is RUNNING, it is currently executing on a CPU. If it needs to wait for an I/O operation or an event, it enters the BLOCKED state. When the thread finishes execution, it moves to the TERMINATED state.

If a thread uses up its time quantum or if a higher-priority thread becomes READY, Solaris performs a context switch. During a context switch, the kernel saves the current thread's execution state and restores the state of another thread. This allows multiple threads to share CPU time efficiently while maintaining responsiveness.

Summary

In conclusion, Solaris executes programs by scheduling threads rather than entire processes. Processes provide the execution environment, while threads and LWPs represent the actual units of execution. By using scheduling classes, priorities, and context switching, Solaris can efficiently manage multiple processes and multiple threads on a multicore system, enabling parallel execution and fair CPU sharing.