Multi-threading Models: User, Kernel, and Combined Threads

Context Clarification

This note delves into different types of threading models—**user-level**, **kernel-level**, and **combined threads**—explaining how they function, their pros and cons, and specific implementation strategies, such as system calls like `Pthread_create` and techniques used to predict thread blocking.

Overview of Multi-threading Models

Most modern systems support multi-threading, either at the **user level**, **kernel level**, or through a combination of both. These models determine how threads are managed and executed within the system.

- **User Threads**: Managed at the user level, using a thread library.
- **Kernel Threads**: Managed directly by the operating system kernel.
- **Combined Threads**: Use both user and kernel-level thread management, combining the advantages of both approaches.

User Threads

User threads are managed entirely by the application, not by the kernel. They are created, scheduled, and terminated by a **user-level thread library**.

Key Features:

- **Portable**: Since the kernel is unaware of them, user threads are portable across different platforms.
- **Fast Context Switching**: Context switching between user threads is rapid because it doesn't involve kernel interaction.

Disadvantages:

- The kernel doesn't recognize user threads. Thus, if one user thread blocks (e.g., during an I/O operation), the **entire process is blocked**, not just the thread.
- User threads can only run on a single CPU at a time, which is inefficient for multi-core systems.

Solution to Blocking: Some libraries convert blocking system calls into **non-blocking system calls**, allowing other threads to continue execution.

Kernel Threads

Kernel threads are managed directly by the operating system's kernel. The **kernel scheduler** is responsible for allocating CPU time to threads and processes.

Key Features:

- **Multiprocessing Friendly**: Unlike user threads, kernel threads can take full advantage of **multi-processor systems**, allowing threads from the same process to run on different CPUs simultaneously.
- **No Global Blocking**: If one kernel thread blocks, other threads within the same process can continue execution.

Disadvantages:

- **Context Switching Overhead**: Kernel threads involve higher overhead during context switching compared to user threads because of the need to interact with the kernel.

Combined Threads (Hybrid Model)

The **combined threading model** incorporates aspects of both user and kernel threads. It allows user threads to be mapped onto kernel threads, offering the flexibility and performance benefits of user threads with the power of kernel-level scheduling.

- **Many-to-one**: Multiple user threads map to a single kernel thread.
- **One-to-one**: Each user thread maps directly to a kernel thread.
- **Many-to-many**: A mix where many user threads can be mapped to many kernel threads, depending on resource availability.

Thread Management: System Calls

Threads are created, managed, and terminated using specific system calls in the **POSIX thread library**:

1. `Pthread_create`: Creates a new thread.
2. `Pthread_exit`: Terminates the calling thread.
3. `Pthread_join`: Waits for a thread to finish execution.
4. `Pthread_yield`: Voluntarily releases the CPU to allow another thread to execute.

Each process has its own **thread table** that stores necessary information about each thread's state. If a thread is blocked, the system stores its registers in this table, allowing it to be resumed later efficiently.

Thread State Prediction

Some operating systems, like **UNIX**, offer mechanisms to predict whether a thread will block during certain operations. This is done using the `SELECT` system call, which checks if a subsequent `READ` operation would cause a block. If blocking is predicted: - `SELECT` is executed, and if successful, the **READ** operation is invoked. - If `SELECT` blocks, another thread can be executed while the first thread waits for the block to clear.

Challenges:

- The prediction system is not highly efficient, often requiring parts of the thread library to be rewritten.
- The code executed before and after the system call to handle this is called **wrapper code**.

Page Fault Handling

A **page fault** occurs when a program requests an instruction not currently in memory. In such cases, the operating system retrieves the required page from the disk. If a **thread** within a process causes a page fault: - The entire process is blocked by the kernel, even if other threads in the process could continue. - This leads to performance issues, as potentially ready-to-execute threads are unnecessarily paused.

Conclusion

Different threading models—user, kernel, and combined—offer varying advantages and trade-offs. While **user threads** are lightweight and fast, they suffer from blocking issues. **Kernel threads**, though more resource-intensive, allow better use of multiprocessing environments. The **combined model** offers a balance by utilizing both user and kernel-level management. Understanding the behavior of threads and managing their states efficiently is crucial for optimizing multi-threaded applications.