OBL2-OS

September 29, 2023

1. Processes and threads

1. A process is the execution of an application program with restricted rights. It is the abstraction for protection provided by the operating system kernel. In simpler terms, a process can be defined as an independent, self-contained program that runs in its own memory space and has its own dedicated resources, including file handlers and network connections. Processes are typically isolated from each other, and communication between them usually relies on inter-process communication mechanisms like pipes or sockets. Each process has its own address space, which defines the set of rights or permissions it possesses. These permissions determine what memory, files, and system calls the process can access. This separation of address spaces means that processes do not directly share memory with one another.

   A thread is a lightweight, smaller unit of a process that runs within the context of the process. It can also be described as sequence of instructions within a process. Threads within the same process share the same memory space and resources, including variables and data structures. Threads can communicate with each other directly through shared memory, making communication more efficient compared to processes. However, threads within a process are not isolated from each other, which means that one thread can directly access and modify the memory of other threads in the same process.

2. Threads are desirable when you have a task that can be divided into smaller, parallelizable subtasks and you want to take advantage of multicore processors. For example, in a web server, each incoming client request can be handled by a separate thread, allowing multiple requests to be processed simultaneously. This approach can improve the overall performance and responsiveness of the server.

   Processes are useful when you need to run independent and potentially unrelated tasks in parallel. For instance, in a cluster computing environment, you might have multiple nodes running separate processes to perform different computations or simulations independently. This isolation is beneficial because it ensures that failures or issues in one process don't affect others.

3. Each thread requires a TCB (Thread Control Block) because it needs a data structure to represent a thread's state. The TCB holds two types of per-thread information: The first is the state of the computation being performed by the thread. And the second is the metadata about the tread that is used to manage the thread. This involves information about the thread's execution context, such as priority, registers, program counter, and stack pointer which is crucial

for switching threads during scheduling and context switches.

4. In cooperative threading, threads voluntarily yield control to other threads. Threads runs without interruption from the operating system scheduler until it explicitly relinquishes control of the processor to another thread. An advantage for cooperative threading is increased control over the interleavings among threads. For example, in most cooperative multi-threading system when one thread runs at a time, other running threads cannot affect the system state. But the disadvantages are that a thread can monopolize the processor and starve other threads, making the system's user interface sluggish or nonresponsive. The steps necessary for a context switch with cooperative threading are the following:
    - The executing thread needs to yield control.
    - The scheduler selects next thread to run.
    - The scheduler updates the TCBs to switch to the selected thread.

    In pre-emptive threading it is the operating system schedular that forcibly interrupts the current running thread and switches to another. The major difference is that the OS has now the ability to interrupt the execution of a thread and allocate CPU time to other threads. Unlike cooperative threading, pre-emptive threading is designed to ensure fair allocation of CPU resources among multiple threads and prevent any single thread from monopolizing the CPU.
    The steps necessary for a context switch with pre-emptive threading are the following:
    - A timer interrupt or event must trigger a context switch.
    - The operative system saves the state if the currently executing thread including register and counter.
    - The scheduler selects next thread to run.
    - The scheduler updates the TCBs to switch to the selected thread.

2. C program with POSIX threads

1. It is the go function that that is executed when a thread runs. The go function prints first a "Hello from thread n" message and then calls the pthread_exit($100 + n$). This function quits the current thread, cleans up, wake up joiner, and returning an exit value that is the sum of 100 and the thread number n.

2. The order of the messages can change each time because threads run concurrently, and the operating system's thread scheduler determines when each thread gets CPU time. The scheduler's decisions are influenced by various factors like the number of CPU cores, thread priorities, and other system load conditions. Therefore, the order in which threads execute is not guaranteed, but

rather random.

3. The minimum threads that could exist when thread 8 prints “Hello from thread 8” is 2. This involves thread 8 and the main thread. The maximum number of threads that could exist is 11. This involves alle the 10 threads and the main thread.

4. The pthread_join function is used to wait for a specific thread to complete its execution. It works similarly as a parent process waits for its child process to finish. In the threadHello.c program, it is used in a loop to wait for all the threads created by pthread_create to finish and retrieve the exit value returned by the thread.

5.

```
void *go (void *n) {
    printf("Hello from thread %ld\n", (long)n);
    if(n == 5)
        sleep(2); // Pause thread 5 execution for 2 seconds
    pthread_exit(100 + n);
    // REACHED?
}
```

If the go function is changed to the code above, thread 5 will sleep/wait for 2 seconds after it prints “Hello from thread 5”. The other threads will continue running without waiting for thread 5. Eventually thread 5 will exit, and the program will proceed with the pthread_join calls. The hello messages from the other threads will not be affected, but the return messages after thread 5 will appear with 2 seconds delay because the messages order are determined in the main function.

6. When pthread_join returns for thread X, thread X will be in the finished state. Thread X has finished its execution and therefore exited. The pthread_join function collects the exit status and releases resources associated with it.