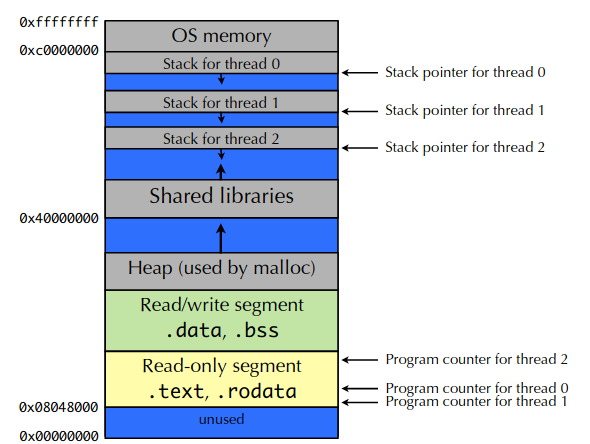
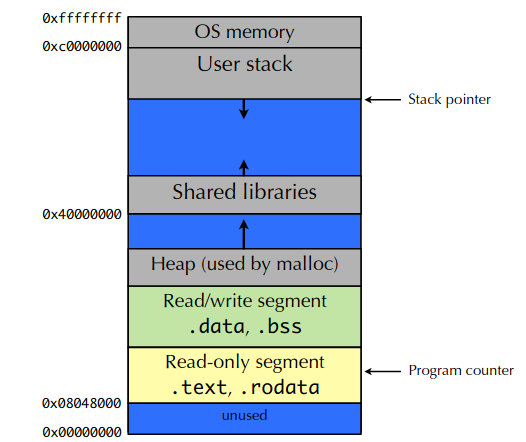
**What are threads? What is the difference between user level thread and kernel level threads? During execution, how does it benefit to have kernel level threads? Describe in detail how threads, user-level or kernel-level are treated for concurrency.**

**THREADS**

Each thread has its own stack pointer, program pointer and registers. All threads share the common address space and open file descriptors (this allows them to communicate directly). Each thread is bound to a particular process but a process can have multiple threads.

Below images depicts the change in a process image from having no threads to having 3 threads running in it.



**(OLD) PROCESS ADDRESS SPACE (NEW) ADDRESS SPACE WITH 3 THREADS**

**Why use threads?**

Threads allow a particular application to perform multiple tasks at a time. Threads are faster to create and destroy. They also do not require separate address space as they all share the common address space. We can overlap computation and I/O with the use of threads.

Example – In MS word, we can create separate threads for separate tasks like, for reading input from keyboard, for writing document in disk, etc.

|  |  |
| --- | --- |
| **KERNEL LEVEL THREADS** | **USER LEVEL THREADS** |
| These are implemented by OS | These are implemented by user |
| Context switch is comparatively slower | Context switch is faster |
| These are visible to OS | These are not visible to OS. |
| If any thread makes a blocking system call than other threads can still work without any affect | If a thread makes a blocking system call than other threads are also blocked |
| These are scheduled by the OS which consists of a lightweight context switch as the memory is shared by all thread. Only program counter, stack pointer and registers need to be saved. | These types of threads exists within a process are therefore scheduled to run within the timeslice of that process. They are not scheduled by the kernel. |
| Kernel level threads are scheduled by OS and so each of them have their own timeslice. Scheduler can smartly schedule them considering the factors like IO bound. | Thread switching is much faster for these threads as there is no context switch. However, an application-based scheduling algorithm can be used for scheduling them. |

**During execution, how does it benefit to have kernel level threads?**

Kernel Level Threads are visible to OS. Therefore, they can be scheduled in an intelligent manner by the OS as compared to user level threads. For example, for threads which are IO bound (I.e, threads that are working on doing IO most of the time), if these are scheduled in case of user level threads then OS doesn’t get to know this. As a result of which, threads cannot be scheduled in an effective manner which ultimately leads to taking more time for execution. On the other hand, this problem is better handled for kernel level threads as they are visible to OS.

Another advantage is that if any one kernel level thread makes a blocking system call then other threads keeps working without any effect. On the contrary, for user level thread all other threads are also blocked in this case. This is because OS cannot see the different user level threads. It only sees the process which contains all these threads. Therefore, if any gets blocked OS blocks the entire process including all user level threads.

**How threads, user-level or kernel-level are treated for concurrency.**

In multithreaded environment, where threads access shared variables, global variables, open files and common address space, it becomes important to synchronize the threads and coordinate their execution.

When shared variables are read/written by multiple threads at a time, then it’s access must be restricted to only one thread at a time. Failing to do this, unexpected results will be obtained. Concurrent access to shared resources without synchronization leads to race conditions. To prevent this mutual exclusion need to be used to allow only one thread to access the shared resource at a time. When one thread leaves the critical section then only other gets entry.

While doing this we must ensure that the 3 conditions for effective handling of critical sections should be ensured – mutual exclusion, progress requirement and bounded waiting. We can even use software based solutions to schedule threads properly in critical sections. Software solutions makes use of busy waiting(or spin locks) to allow only one thread to enter critical section.

**REFERENCES**

[[1]](#endnote-2) <https://cs61.seas.harvard.edu/wiki/images/0/0e/Lec17-Threads.pdf>

[[2]](#endnote-3) <https://people.cs.pitt.edu/~jacklange/teaching/cs2510-f15/obsolete_lectures/03-ProcAndThrds.pdf>

[[3]](#endnote-4)[http://www.cs.iit.edu/~cs561/cs450/ChilkuriDineshThreads/dinesh%27s%20files/Processes%20and%20Threads.html](http://www.cs.iit.edu/~cs561/cs450/ChilkuriDineshThreads/dinesh's files/Processes and Threads.html)

[[4]](#endnote-5) <https://cs61.seas.harvard.edu/wiki/images/6/65/Lec18-Synchronization.pdf>

1. <https://cs61.seas.harvard.edu/wiki/images/0/0e/Lec17-Threads.pdf> [↑](#endnote-ref-2)
2. <https://people.cs.pitt.edu/~jacklange/teaching/cs2510-f15/obsolete_lectures/03-ProcAndThrds.pdf> [↑](#endnote-ref-3)
3. [http://www.cs.iit.edu/~cs561/cs450/ChilkuriDineshThreads/dinesh%27s%20files/Processes%20and%20Threads.html](http://www.cs.iit.edu/~cs561/cs450/ChilkuriDineshThreads/dinesh's files/Processes and Threads.html) [↑](#endnote-ref-4)
4. <https://cs61.seas.harvard.edu/wiki/images/6/65/Lec18-Synchronization.pdf> [↑](#endnote-ref-5)