Solution to HW4

5.9:

If threads are implemented at the user level, when a thread makes a blocking system call all threads of the process would get blocked. Thus a thread should avoid using blocking systems calls such as open, read, and write.

5.10:

Causes of two key state transitions are as follows: The \textit{running} $\rightarrow$ \textit{blocked} state transition can occur for a process (i) when threads other than the currently running thread are waiting for events to occur and the currently running thread also wishes to wait until an event occurs, or (ii) when the currently running thread makes a blocking system call.

Causes of the state transition \textit{blocked} $\rightarrow$ \textit{ready} differ in these two cases. In case (i), the process becomes \textit{ready} when the event awaited by some thread in it occurs, whereas in case (ii) the process becomes \textit{ready} when the event awaited by the thread that made the blocking system call occurs.

5.13(a):

A kernel-level thread provides parallelism but incurs higher overhead. If the system contains more than one CPU, creation of kernel-level threads for CPU-bound activities would exploit the parallelism provided in the system; however, if the system contains a single CPU, it is better to use user-level threads and curtail the overhead.

5.14 (b):

No potential for speed-up exists in the single CPU case as both processes involve only CPU activities. In fact, the execution speed would degrade due to the overhead of creating and synchronizing the processes to compute $a*b$ and $c*d$. Even in the multiple CPU case, the speed-up is uncertain because the extent of parallelism is very small and creation and synchronization of threads would incur overheads.
5.14(d):

$n$ is large, so inherent parallelism among the threads is large. However, the threads contain only CPU activities; hence no speed-up can be obtained when the system contains a single CPU. However, significant speed-up would be obtained in the multiple CPU case.