COMP3430 – Assignment 2

Trevor Bekolay, 6796723

1. 2 processes are currently using the resource. 1 process is blocked on the semaphore.  
     
   This comes quite simply from the definition of a semaphore. The initial value (2) defines how many processes may simultaneously access a resource. Since the current value is negative, that means that the maximum number of processes are currently using that resource: 2, in this case. So once those 2 slots are used up and the semaphore’s value is 0, any further processes trying to access the resource will be blocked and the semaphore decremented. The absolute value of the semaphore (if it is negative) tells us how many processes are waiting for a resource; in this case, 1 process is waiting for the resource.
2. a) An Operating System managed concurrency mechanism will require at least one system call. A system call requires, at the very least, a mode switch (and possibly a context switch, depending on the OS implementation). The switching may also require the OS to do some rescheduling, which takes time. If you will, on average, busy wait for a shorter amount of time than takes to handle the system call, you are probably better off busy waiting.  
     
   b) As posited in (a), research confirms that spinlocks are efficient if threads are only likely to be blocked for a short period of time, because “they avoid overhead from operating system process re-scheduling or context switching.” (Source: <http://en.wikipedia.org/wiki/Spinlock>)   
     
   In the multiprocessor environment, however, it makes significantly more sense to do spinlocks because more than one process can truly be running at one time, and hence, the lock may be released while a process is spinning. It is thus useful to restrict spinlocking to short critical sections that are likely to be executed on separate processors. (Source: <http://www.cse.unsw.edu.au/~cs9242/03/lectures/lect11.pdf>)  
     
   Putting these two ideas together, we can create an algorithm that will give us good performance in various situations. When attempting to access a critical section, the thread starts a spinlock. It keeps track of how long it’s been spinning, and once it has spun longer than some threshold, it blocks the process and switches to the next thread. The amount of time spent spinning before blocking may be related to the OS call overhead, or may be dynamically determined based on previous observations.
3. eagle.cs.umanitoba.ca 102% getconf GNU\_LIBPTHREAD\_VERSION  
   NPTL 2.4  
     
   a) NPTL (Native POSIX Thread Library) replaced a pthreads implementation called LinuxThreads.  
     
   b) One problem was that LinuxThreads interfered with signal handling. LinuxThreads used the clone() system call to create a new process that shared the parent's address space for each thread. However, the cloned process would have a different process ID, making it difficult to do direct inter-process communication. Also, LinuxThreads used the SIGUSR1 and SIGUSR2 signals for thread coordination, meaning that these signals could not be used for other purposes. (Source: http://en.wikipedia.org/wiki/LinuxThreads)  
     
   c) The implementation that lost out to NPTL was NGPT (Next Generation POSIX Threads).  
     
   d) The NPTL implementation of pthreads is a hybrid method because it has features of both user-level threads and kernel-level threads. Like user-level threads, NPTL requires the user to manage a program’s threads; code must be written to support the creation and management of threads. The kernel, however, is aware of those threads, and schedules on a thread-level rather than a process-level. So, it is a hybrid method because both the user and kernel are aware of and explicitly managing threads.  
     
   To be more specific, the NPTL implementation is a 1 x 1 model. This means that, unlike other hybrid methods (those that use the M x N model), every user-level thread has a corresponding kernel-level thread. So when a user calls the pthread\_create method, they are creating a kernel-level thread.