William Shreeve
Matt Kennedy

CIS 452 Lab 6

1. Sample 1 intends to take the values of shmPtr[0] and shmPtr[1], which is 0 and 1 respectively, and swap the values. It does this by forking before the loop to swap, and both the parent and the child are sent to do the same task. For every iteration of the loop, the parent or child will swap the values of [0] and [1] once. The number of loops is defined by the command line arguments given when the code is executed.
2. The expected result of the program would be for the child to swap values loop number of times and exit, while the parent runs and does the same thing. After the swapping has occurred, the parent will wait for the child to exit with the wait command, then clean up and exit.
3. For lower values of loop, the program typically has the result of one of the values being 0, and the other being 1, which is the expected results. Specifically, the values of shmPtr[1] and [0] end up being the same in every instance for loop values less than or equal to 10,000. Once we started using values for loop greater than 100,000, the values of shmPtr[0] and [1] started having varied results.
    a. The values would sometimes be swapped. Instead of shmPtr[0] and [1] being 1 and 0 respectively everytime the program is ran, sometimes they are swapped to be 0 and 1, respectively.
    b. The other outcome would be that both values of shmPtr[0] and [1] are equal, and either 0 or 1.
4. We had two ‘interesting’ types of outputs for loop values greater than 100,000.
    a. The first situation was that shmPtr[0] and [1] would still have one as zero, but the other as 1, but it was random for which one was which. This is due to the fact that the code runs the loops alongside one another, and eventually one of the loops gets ahead of one another, so the loop of the parent runs more than once in a row, before the child loop would run an iteration, effectively skipping a swap.
    b. The second situation would be shmPtr[0] and [1] being equal, with values of 0 or 1. This occurs when the two processes are running at the same time, and the values of one of the ints becomes the same as the other. The easiest way to explain this is to trace how the program would run, using loop1 and loop2 as the parent and child loops. Assuming shmPtr[0] = 0, and shmPtr[1] = 1 at this point in time, loop 1 and loop 2 would be running at nearly the same time. Loop1 would be writing the value of shmPtr[0] to temp, and assigning the value of shmPtr[0] to shmPtr[1], making [1] = 1. During this, loop 2 would be doing the same, but before loop 1 can assign the value of temp to shmPtr[1] to temp, loop 2 takes the value of shmPtr[1] and assigns it to shmPtr[0], so [0] =1. Since loop 1 didn’t get a chance to assign the value of shmPtr[1] from temp, and loop 2 still ran its process, we end up with both shmPtr[0] and shmPtr[1] as the same value, in this example, 1. Since the loops would just continue swapping 1 with 1, with get the output of the two values being equal. This is done because there was no attempt to synchronize the two loops, or more broadly speaking, the two processes. Since they are not being synchronized, the two loops are running constantly in parallel, which can be a hazard when two or more processes are working with the

same values in memory. In this case, we saw that it is a possibility that the order of reading and writing memory can cause invalid outputs when it is unmanaged.

5. From the man pages:

```
    unsigned short sem_num;  /* semaphore number */
   short        sem_op;   /* semaphore operation */
   short        sem_flg;  /* operation flags */
```

   - sem_num is the identifier for the nth semaphore in the semaphore set.
   - For sem_op, if it is a positive integer, the op adds this value to semval. If sem_op is 0, from the man pages: "if *sem_op* is zero, the process must have read permission on the semaphore set. This is a "wait-for-zero" operation: if *semval* is zero, the operation can immediately proceed." This behavior can vary depending on read/write permission of the users. If sem_op is a negative integer, from the man pages: "*sem_op* is less than zero, the process must have alter permission on the semaphore set. If *semval* is greater than or equal to the absolute value of *sem_op*, the operation can proceed immediately: the absolute value of *sem_op* is subtracted from *semval*, and, if **SEM_UNDO** is specified for this operation, the system updates the undo count (*semadj*) for this semaphore. "
   - sem_flg is used to specify permissions of the user and what is available with the semaphore. For example, in the sem_op section above, there are several instances where access must be specified with sem_flg for a specific part of sem_op to work.

6. From the man pages, "If an operation specifies **SEM_UNDO**, it will be automatically undone when the process terminates... The SEM_UNDO option releases the semaphore when the process exits, waiting until there are less than two processes running concurrently".

   From https://users.cs.cf.ac.uk/Dave.Marshall/C/node26.html:
   "If a process with exclusive use of a semaphore terminates abnormally and fails to undo the operation or free the semaphore, the semaphore stays locked in memory in the state the process left it. To prevent this, the SEM_UNDO control flag makes semop() allocate an undo structure for each semaphore operation, which contains the operation that returns the semaphore to its previous state. If the process dies, the system applies the operations in the undo structures. This prevents an aborted process from leaving a semaphore set in an inconsistent state. If processes share access to a resource controlled by a semaphore, operations on the semaphore should not be made with SEM_UNDO in effect. If the process that currently has control of the resource terminates abnormally, the resource is presumed to be inconsistent. Another process must be able to recognize this to restore the resource to a consistent state. When performing a semaphore operation with SEM_UNDO in effect, you must also have it in effect for the call that will perform the reversing operation. When the process runs normally, the reversing operation updates the undo structure with a complementary

value. This ensures that, unless the process is aborted, the values applied to the undo structure are cancel to zero. When the undo structure reaches zero, it is removed.”