1 OPERATING SYSTEMS LABORATORY VI Additional - Synchronization and Critical Section, InterProcessCommunications I

Examples&Exercises:

1. **Shared Memory**; code35.c

   - One of the simplest interprocess communication methods is using shared memory.
   - Shared memory allows two or more processes to access the same memory as if they all called **malloc** and were returned pointers to the same actual memory.
   - When one process changes the memory, all the other processes see the modification.
   - Shared memory is the fastest form of interprocess communication because all processes share the same piece of memory and it does not require a system call or entry to the kernel.
   - Because the kernel does not synchronize accesses to shared memory, you must provide your own synchronization. For example, a process should not read from the memory until after data is written there, and two processes must not write to the same memory location at the same time.
   - Analyze the code.

2. **Semaphore, Shared Memory**; code36.c
   Shared memory example to find the sum of the first 1000 numbers in parallel using two processes, one to add even numbered elements and one to add odd numbered elements;

   - Compile the following codes and link them to *code36*.
     code37.c, code38.c, code39.c
   - Execute several times and observe that how the output changes.

3. **Race Conditions**; complete the following program. code30.c

   - Suppose that your program has a series of queued jobs that are processed by several concurrent threads.

- After each thread finishes an operation, it checks the queue to see if an additional job is available.
- If job_queue is non-null, the thread removes the head of the linked list and sets job_queue to the next job on the list.
- Now suppose that two threads happen to finish a job at about the same time, but only one job remains in the queue.
  - The first thread checks whether job_queue is null; finding that it isn't, the thread enters the loop and stores the pointer to the job object in next_job.
  - At this point, Linux happens to interrupt the first thread and schedules the second. The second thread also checks job_queue and finding it non-null, also assigns the same job pointer to next_job.
  - By unfortunate coincidence, we now have two threads executing the same job.
  - To make matters worse, one thread will unlink the job object from the queue, leaving job_queue containing null. When the other thread evaluates $job\_queue- > next$, a segmentation fault will result.
- This is an example of a race condition. Under "lucky" circumstances, this particular schedule of the two threads may never occur, and the race condition may never exhibit itself.

4. **Critical Sections**; complete the following program. code31.c
   - A thread may disable cancellation of itself altogether with the **pthread_setcancelstate** function.
   - The first argument is **PTHREAD_CANCEL_DISABLE** to disable cancellation, or **PTHREAD_CANCEL_ENABLE** to reenable cancellation.
   - The second argument, if not null, points to a variable that will receive the previous cancellation state.
   - Using **pthread_setcancelstate** enables you to implement critical sections.
   - In the following program, the transfer with a function, such as **process_transaction**, disables thread cancellation to start a critical section before it modifies either account balance.

- Note that it's important to restore the old cancel state at the end of the critical section.Your function will leave the cancel state the same way it found it.
- Even assigning a value to a global variable can be dangerous because the assignment may actually be carried out in two or more machine instructions, and a second signal may occur between them, leaving the variable in a corrupted state!!

5. **Condition Variable** (Spin loop); code33.c

   - Suppose that you write a thread function that executes a loop infinitely, performing some work on each iteration.
   - The thread loop, however, needs to be controlled by a flag:
     - the loop runs only when the flag is set;
     - when the flag is not set, the loop pauses.
   - Because the flag is accessed by multiple threads, it is protected by a mutex. This implementation may be correct, but it is not efficient. The thread function will spend lots of CPU whenever the flag is not set, checking and rechecking the flag, each time locking and unlocking the mutex.
   - Complete the program by writing a main function and **do_work** function.
   - Create two threads having to access to **thread_flag**.
   - Is there any possible race conditions? Explain.