Important Notes:

0. It is the responsibility of the wait function in the parent process to clean up its zombie children. If the child process finishes before the parent process calls wait,the child process becomes a zombie. When the parent process calls wait, the zombie child's termination status is extracted, the child process is deleted, and the wait call returns immediately.

1. Two folded functionality of waitpid – (1) synchronization between parent process and child process and (2) cleaning up zombie child

2. Semaphore provides both process and thread synchronization to access shared memory. Semaphore is applicable for both process and thread.

3. Children processes created by calls to fork inherit attached shared segments; they can detach the shared memory segments, if desired.

4. Thread self terminates: void pthread_exit(void* thread_return)
   thread terminates a peer thread: void pthread_cancel(pthread_t tid)

5. If thread A wants to receive the return value of thread B, A has to truly wait for B to terminate via pthread_join(void** thread_return)

6. The stack area is divided by threads. Each thread has its own stack.
   Threads share heap and global variable (on initialized or uninitialized memory segments)

7. If the main thread calls pthread_exit, it waits for all other peer threads to terminate and then it terminates the main thread and the entire process with a return value of thread return. Calling pthread_exit in the main thread is a common way to ensure that all threads finish. Thus it provides an implicit thread synchronization mechanism.

8. Mutex can prevent (1) accessing data at the same time and (2) race condition.

9. Mutexes can only be applied to threads in a single process and do not work between processes as do semaphores.

10. Waiting for a mutex
(10.1) pthread_mutex_lock() acquire a lock on the specified mutex variable. If the mutex is already locked by another thread, this call will block the calling thread until the mutex is unlocked.

(10. 2) pthread_mutex_trylock() (Lock Mutex with No Wait) Attempt to lock a mutex or will   return error code if the mutex is already locked by another thread.

11. Purpose of condition variable: to wake up **sleeping threads (not process!)**
You don't wake sleeping threads directly instead you invoke the condition variable, which then will wake up one (or all) threads that are sleeping inside the condition variable. Threads sleeping inside a condition variable are woken up by calling pthread_cond_broadcast (wake up all) or pthread_cond_signal (wake up one). Note despite the function name, this has nothing to do with POSIX signals!

12. After a new child process is created, both processes will execute the next instruction following the fork() system call. Therefore, we have to distinguish the parent from the child. This can be done by testing the returned value of fork().

13. The fork-exec-wait trilogy
A common programming pattern is to call fork followed by exec and wait. The original process calls fork, which creates a child process. The child process then uses exec to start execution of a new program (exec() replaces the program in the current process with a brand new program, by specifying an executable or or an interpreter script to it). Meanwhile the parent uses wait (or waitpid) to wait for the child process to finish.

14. Signals are software generated interrupts that are sent to a **process** (**not thread!**) when an event happens. For instacen, signal(SIGINT, SIG_IGN); and signal(SIGINT, SIG_DFL); The first argument is signal type, the second argument is the signal handler that responds to the signal.