**Question 1**

1. Inconsistency can occur in the line : result = result + 1

Two possible results are 100 and 200

100 : Since we are running on a single processor the following scenario is possible : when the first thread is created it starts and goes to the first iteration of the for-loop , in the line result = result + 1, it calculates first the value of the expression result + 1, that is 1 on the first iteration . Then thread is preempted before it finishes the assignment. Suppose that even after pthread\_join the first thread doesn’t return to execution but the second one begins to run (result value is still 0) and terminates after 100 iterations. Now the result value is 100. After that the first thread returns to run to the point it was preempted at and assigns 1 to the result variable, i.e. it continues to behave like result was never changed till it’s termination. After 99 remaining loops the value of the result will be incremented by one 99 times so it will be equal to 100.

200: Suppose that one of the threads terminates before another one stars a first iteration of the for-loop . That means that after first thread the result value will be 100. Then the second thread will begin from this value and increment it to 200.

1. We can define a global mutex before the beginning of the main :

pthread\_mutex\_t m;

int main(){

…

…

}

And then use it in the critical section:

pthread\_mutex\_lock(&m);

result = result + 1;

pthread\_mutex\_unlock(&m);

**Question 2**

1. Each thread has a unique task\_struct. Threads have independent stacks that are not shared, however a pointer to the stack could be hold by another thread and in this way a stack could also be accessed by another thread, registers are also unique for threads. The list of shared resources determined by the clone functions arguments. For example a result of the call to  
   clone(CLONE\_VM | CLONE\_FS| CLONE\_FILES | CLONE\_SIGHAND, 0)

will result in creation of a thread that shares the address space, filesystem resources, file descriptors, and signal handlers with the thread that called to clone() .

1. Both threads and processes are tasks . The main difference is that threads share more types of resources than processes. Threads share memory, file descriptors and signal handlers. Interaction between threads is much simplier – just write/read to/from the . shared variables. This also means complexities – we have to prevent corruption of the . data structures and shared variables.
2. Multithreading seems to be useless when, for example , we are talking about series of simple calculation in which every result is based on previous results in the series for example like calculating of the Fibonacci series. In order to calculate the i-th member i-1 and i-2 members should have been already calculated and so on. From the other side any new computation could not be started before the i-th will be finished. So the whole calculation should be done in series.

Multithreading could take advantage of multiprocessor machine for instance to multiply two matrices. For example for NxN matrices NxN threads could be created. Each thread will compute one number in the product matrix.

Multithreading could also be effective in uniprocessor system for example if there is a constantly running application like text editor (one process) and it’s work is based on the input that could be handled by another process. Since this input must be shared between two processes, threads would be an appropriate solution.

1. 1:1 threading model : each usel-mode thread corresponds to one kernel-mode thread . Each pair interact by means of syscalls and signals. Therefore every user-thread is controlled by kernel scheduler.  
   Advantages : - the simplest to implement at the library level  
    - threads can run on number of processors   
    - threads do not block each other  
   Disadvantages: - need a lot of kernel resources to create a lot of threads   
     
   1:N model : maps all the user threads to a single kernel-space thread.  
   Advantages : - portability  
    - thread creation, termination, cleanup, and synchronization is cheap  
    - possible to create huge nmbers of threads

Disadvantages: - can’t utilize a kernel scheduler, don’t use multiprocessor scheduling  
 - blocking I/O operations can block all the user threads

M:N model : M kernel threads for N user threads, is a model that is a hybrid of the previous two models.  
Advantages : - Take advantage of multiple CPUs  
 - Not all threads are blocked by blocking system calls  
 - Cheap creation, execution, and cleanup  
 - not consuming kernel resoures for the user threads that are not runnable  
 - The library-level scheduler can switch between threads much faster because it doesn’t have to enter the kernel

Disadvantages: - Difficult to write, maintain, and debug code  
 - scheduling occurs both in the kernel and in the libraries

1. 1) In this case all the advantages of multiple processors could not be used since when all kernel threads are scheduled there are still available cores that could be used for the application purposes.

2) Seems to be the best solution to give a process just as many kernel threads as there are processors. But if a user thread executes a blocking system call (such as reading from an I/O device) or suffers a page fault, then its underlying kernel thread also blocks--another user thread may be ready to execute, but no kernel thread is available to be assigned to it.   
However there exists an approach called scheduler activations that provides an explicit means for the user-level and kernel schedulers to cooperate .

3) It is not en effective approach because kernel will be wasting time doing the context switching, even though the application has no need for such time slicing.

**Question 3**

1. 16742 is the process that runs main. Then it starts a thread group leader – a thread which tgid equals to it’s pid that is 16743 in this case. This is the "manager thread," which is part of the internal implementation of GNU/Linux threads. The manager thread is created the first time a program calls pthread\_create to create a new thread. After that we manually created two additional threads that are created by 16743 thread their ids are 16744 and 16745 (since Linux threads still gives a unique pid to each thread), they were cloned by 16743 that’s why they’ve got ppid (that is actually equals to tgid) = 16743 i.e. the pid of the thread that cloned them.   
   Let us note that in NPTL it would be another output since getpid() was changed to return tgid (when in Linux threads it returned a pid as we see).

* The manager thread is responsible for handling and terminating all his threads. Here <http://comet.lehman.cuny.edu/jung/cmp426697/NPTL.pdf> is written that a fatal signal is able to kill all the threads. **Once a process receives a fatal signal, the thread manager kills all the other threads (processes) with the same signal.** Which means that if you kill the thread manager, it won't get a chance to kill the other threads.
* In McCracken’s article it’s written that solutions that are compatible with POSIX have problems when a signal is blocked by a task. Each task has it’s own signal flag and if its blocked the signal will remain pending despite there could be tasks in the process that didn’t block the signal.
* The signal system is far from perfect, therefore synchronization primitives based on them will be very slow and complicated.
* Because of broken signal handling signals that are responsible for termination a multi-threaded process will not work. Debuggers are also have a same problem.
* Having a unique ID for each thread leads to incompatibility problems.
* A small for some purposes maximal number of threads – 8192.
* Because of enormous amount of threads per process /proc is barely usable
* Kernel does not support the signal implementation causing problems in delivering some special signals.
* Since signals are not implemented well it causes problems in ensuring synchronization.

1. In order to take an advantage of multi-processor machines kernel threads had to be used so 1:N model was eliminated . In M-N model multi-processor concept could be supported by scheduling user-level threads on the available kernel-threads. Promising and successful approach based on two schedulers called Schedule Activations was proposed , but finally due to high complicacy of coding developers gave preference to 1:1 threading model.

A manager thread concept is used in Kernel Threads but as it was shown manager concept is dangerous for synchronization in SMP and NUMA systems furthermore all problems that thread manager was designed to solve could be also solved by kernel itself or by thread library. Having no manager thread reduce context-switch rate and simplifies the design so the goal was to avoid using manager.

Maintaining a list of all threads could not be entirely avoided because of complications with fork function. But the procedure of deletion if the thread therefore task termination was significantly simplified by decrementing the counter specified to each thread, rather than walking through the list

A new system call called futex that could be used to implement basic locking primitives was introduced.

Optimizations of memory allocation include merging necessary blocks of data for threads and placing them together on the stack. Some direct de-allocations also avoided , furthermore the values of “freed” threads continue to be used.

The POSIX signal handling is implemented.

Time complexity of function exit was significantly improved.

The maximal number of threads was extended up to 2 billion on IA32. The filesystem fixed to support more than 64K processes.