# Theoretical exercise for Chapter 3

Student name: Doan Cao Thanh Long

Student ID: 20162513

Class: ICT-02.k61

## Question 1: Compare process vs thread

| Process | Thread |
|---|---|
| Is a program in execution | Is a segment of process |
| A process can has one or more threads | There is no thread that exist separately, thread must be assigned to a process |
| Each process is isolated from each other | Threads in one process share data and memory with each other |
| Processes are not lightweight | Threads are lightweight |
| Take more time for context switching | Take less time for context switching |
| Consume more resources and take longer time to terminate | Consume fewer resources and take less time to terminate |
| Responsibility to separate resource to processes is belong to OS | Responsibility to separate resource to threads is belong to app developer |
| Communication between processes take more time compare to communication between threads | Communication between threads take less time compare to communication between processes |

## Question 2: Would it make sense to limit the number of threads in a server process?

It would make sense because more threads can cause more problems as they share finite hardware resources.

## Question 3: Why do we consider lightweight processes as a solution of combining advantages of deploying thread packages in user and kernel space? Having only a single lightweight process per process is also not such a good idea. Why not?

Threads in user mode are cheap to create and destroy, and it is easy to switch context.

However, invocation of a blocking system call will block the entire process. If implementing threads in kernel mode, it can avoid this problem

We consider the lightweight process (LWP) as a solution of combining advantages of deploying thread packages in user and kernel space because each LWP is assigned to a thread in user mode while LWP also belongs to the main thread, which is in kernel mode.

If a thread calls a blocking system call, it will block that thread and its LWP, but it does not block another LWP, which is another thread because LWPs belong to the kernel space.

Having only a single lightweight process per process is also not such a good idea because there is only one user-level thread, blocking this thread will also block the entire process.

## Question 4: What are the advantages of the multi-threaded server compared to the single-threaded server?

Multi-threaded server can execute many client requests at a time, while single-threaded server can only execute 1 request at a time sequentially

## Question 5: Give the advantages and disadvantages of each of three models of multithreaded server: Thread-per-request, Thread-per-connection, and Thread-per-object?

| | Thread-per-request | Thread-per-connection | Thread-per-object |
|---|---|---|---|
| | If the server receives a request, it will create a new thread. | For each new connection, the server creates a new thread. | For each remote object, the server creates a new thread. |
| Advantages | This model can make use of bandwidth because of the ability to sync unlimited between worker threads, also there is no need for a request queue. | The number of threads is stable, there are not many tasks of create and destroy thread. | The number of threads is stable, there are not many tasks of create and destroy thread. |
| Disadvantages | If there are too many requests, the number of threads can be huge => memory resources exhausted. Also, there are many tasks of create and destroy threads => risk of having too many overheads of system. | Client could be served with high latency because of resource/ load balancing as there are threads that serve too many requests while other threads have nothing to do. | Client could be served with high latency because of resource/ load balancing as there are threads that serve too many requests while other threads have nothing to do. |

## Question 6: Give an example of a server following Finite state machine. Explain why this kind of server is single thread but high scalability?

Example of a server following Finite state machine: a nodeJS server

A finite state machine server is single-threaded but high scalability because of following reasons

- The client requests are queued and processed. At a time, the server performs operations in the queue.
- Single thread perform task sequentially, asynchronous, event driven.
- Non blocking itself to wait for other things.

- No need multithreading.
- Simulating threads and their stacks

## Question 7: Why do the clients in Distributed Systems also need to be multi-threaded? Give an example.

Client in Distributed Systems also need to be multi-threaded because:

- Multi-threaded client separate user interface and processing.
- Solve the problems of tasks waiting for each other.
- Increase the system-performance when working with different servers.

Example: web browsers.

- Threads in the backend send requests to web server to receive contents then send to the threads of the frontend.
- Threads in the backend continously receive contents from the server.
- Frontend threads display the received contents from the backend.

## Question 8: Why do we need virtualization technology?

We need the virtualization technology because:

- Virtualization allows legacy software to run on expensive mainframe hardware.
- Hardware changes very fast compared to software => need virtualization to make all software to run on the new type of hardware infrastructure.
- Diversity of platforms and machines can be reduced by letting each software run on its own virtual machine.

## Question 9: Why is X-Window system suitable for thin client architecture?

X-Window system is suitable for thin client architecture for these reasons:

- X-Window system provides dedicated instruction for applications to control the display.
- X-Window system helps building inexpensive terminal for many users to simultaneously use the same large computer server and execute application programs as clients.

## Question 10: Considering the problem of finding server, compare the daemon server and superserver

*Daemon server*: run behind the system, every server process of the machine must register its endpoint to the daemon, this daemon will store all the endpoints to the endpoint table, clients send requests to daemon process to get the endpoint to appropriate server. Server process must run always => costly in system performance

*Superserver*: is a kind of daemon server but after registering to the superserver, the server process can stop itself, after that, when a client sends a request to superserver, it will execute the requested server.