Part 1. Fun with forks

(a) What is the output generated by this program? In fact the output is not uniquely defined, i.e., it is not always the same. So please give three examples of possible outputs.

```
int i = 0;

main()
{
    printf("%d
", i); /* print i on a line */
    fork();
    i = i + 1;
    printf("%d
", i);
    fork();
    i = i + 1;
    printf("%d
", i);
}
```

(b) Briefly justify/explain your answer for (a). Try to characterize the set of all possible outputs.
(a) Alice and Bob are back together. Today Alice wants to send Bob a message that is secret and also authenticated, so that Bob “knows” the message came from Alice. Alice and Bob have keypairs and each knows the other’s public key.

How should Alice send the message? How should Bob validate the message? Briefly explain why your scheme works, and note any additional assumptions.

(b) Alice, Bob, and their friends interact via the social networking service Faith. They protect their communication using secure sockets (SSL) for transport-layer security. Nancy wants to spy on their communication. Assume that Nancy can subvert any network provider and so intercept and/or inject packets at any point in the network. Suppose further that Nancy obtains the private key of a Certifying Authority that is trusted by Alice and Bob. How can she use the key to mount an attack?
Part 3. Threads and concurrency control

This problem asks you to write the core of a program to issue and service disk requests. **As always:** “Any kind of pseudocode is fine as long as its meaning is clear. You may assume standard data structures, e.g., linked lists: don’t write code for those.”

There are \( N \) requester threads that issue requests and one servicer thread to service the requests. Requesters issue requests by placing them on a queue. The queue is bounded to at most MAX requests: a requester waits if the queue is full. Requests are synchronous; each requester may have at most one pending request. Observe the following constraint: the servicer handles requests only when the queue is full or all living requesters have a pending request.

Write procedures for the requesters and servicer. You will need to use some control concurrency: use monitors, mutexes, condition variables, and/or semaphores (your choice). Also, please briefly discuss the request scheduling policy used by the servicer thread, and the impact that it might have on performance.
Part 4. Caching

Caching is a common technique in system software. Operating systems manage primary memory as a cache for objects drawn from various sets while those objects are in active use: virtual memory pages, file blocks, directory name entries, IP address mappings on the local network, file metadata descriptors (inodes or vnodes), and so on.

Outline the data structures that an operating system might use to represent such a cache. What are the important operations on these structures? In this question I am asking you to generalize from the examples and outline what is common across these software-managed caching structures, and also highlight some differences.
Part 5. Isolation

One important design goal of modern operating systems is isolation. Isolation means that application processes cannot interfere with one another or with the operating system itself. In particular, untrusted user code cannot access or modify any piece of data unless the system policy permits it. The operating system also allocates resources (e.g., memory, processing time, network bandwidth) fairly: a user process can use only the resources that the system policy grants to it.

This question asks you to list and summarize how the hardware and the kernel software work together to enforce isolation. Your answer should refer to important hardware mechanisms (e.g., protected mode execution, virtual address mapping, traps, faults, interrupts) and how the OS uses them to isolate processes from one another.