Information text

A barrier synchronization is a function that does not return control to the caller until all **p** threads of a multithreaded process have called it.

A possible implementation of the barrier function uses a shared **counter** variable that is initialized to 0 at program start and incremented by each barrier-invoking thread, and the barrier function returns if counter has reached value **p**.

We assume here that **p** is fixed and can be obtained by calling a function `get_nthreads()`, that load and store operations perform atomically, and that each thread will only call the barrier function once.

The following code is given as a starting point:

```
static volatile int counter = 0;  // shared variable

void barrier( void )
{
  counter = counter + 1;
  while (counter != get_nthreads())
     ;  // busy waiting
  return;
}
```

Question 1

Identify the critical section(s) in this implementation, and use a *mutex lock* to properly protect the critical section(s), without introducing a deadlock. Show the resulting pseudo code.

Question 2

Show by an example scenario with **p=2** threads (i.e., some unfortunate interleaving of thread execution over time) that this implementation of `barrier` may cause a program calling it (such as the following) to hang.

```
void main( void )
{
 ... // create p threads in total
 ...
 barrier();
 ...
}
```

Question 3

Today, many processors offer some type of atomic operation(s). Can you use here an atomic `fetch_and_add` operation instead of the mutex lock to guarantee correct execution? If yes, show how to modify the code above and explain. If not, explain why.

Question 4

Suggest a suitable way to extend the (properly synchronized) code from question 2 to avoid busy waiting. Show the resulting pseudocode.

Question 5

Consider the following (Unix) C program.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main() {
  printf("A");
  pid_t pid1 = fork();
  pid_t pid2 = fork();
  if (pid2 == 0) {
    printf("A");
  }
  printf("A");
}
```

How many times will the letter A be printed to the standard output when executing this program?

Question 6

Every process is associated with a number of areas in memory used to store the program code, variables, etc. For each memory item below connect it with correct allocation option. Note that that options can be used more than once and not all options must be chosen.

The page table of a process

The compiled machine code of a program

The memory used to store a local variable declared in a function

The memory used to store the content of a function parameter

Memory allocated using `malloc`

A global (static) variable

Options: Allocated in kernel memory, Allocated in registers, Allocated on the stack, Allocated in shared memory, Allocated in kernel memory, Allocated on the heap, Allocated In the data section, Allocated in the text section, Allocated in function memory

Question 7

Give an example of a situation (table with jobs) where the SJF algorithm provides better performance as measured by average turnaround time compared to FCFS. There must be at least 4 jobs in the table and your example must not be copied from anywhere. Motivate by showing the average turnaround time for both algorithms.

Question 8

[Fill in the blanks]

Banker's algorithm is a deadlock [] algorithm. Freedom from deadlocks is a [] condition for ensuring progress. The algorithm guarantees that only so-called [] states will be reached by checking every resource allocation request. If a resource allocation leads to an undesired state, the request is [].

Question 9

Explain how paging supports sharing of memory between processes.
paging makes it possible to process to allocate data in empty addresses in the physical memory

Question 10

Explain why page faults occur, under what circumstances and what happens after. Describe the set of events step by step, considering also the possibility of there being no free frames. For each step describe where the logic that executes this step is located (e.g., OS kernel, MMU, program code).