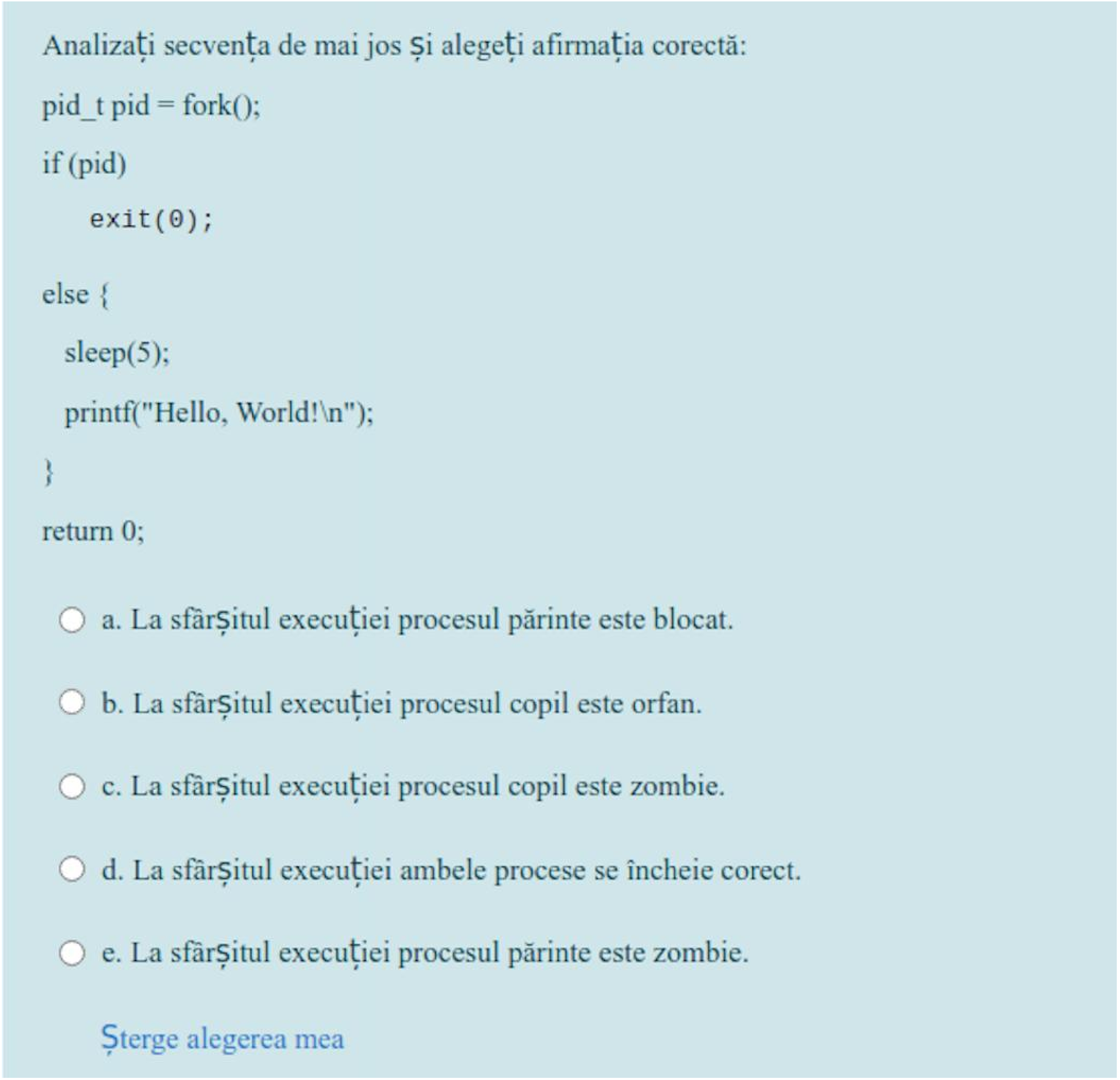
```c
#include <sys/types.h>
#include <unistd.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>


/*
It creates a new process which is a copy of the calling process.
That means that it copies the caller's memory (code, globals, heap and
stack),
registers, and open files.

The calling process returns from fork() with the pid of the newly
created process (which is called the "child" process.
The calling process is called the "parent" process).

The newly created process, as it is a duplicate of the parent,
also returns from the fork() call (this is because it is a duplicate -- it
has the same memory and registers,
and thus the same stack pointer, frame pointer, and program counter, and
thus has to return from the fork() call).
It returns with a value of zero. This is how you know what process you're
in when fork() returns.
*/


/*
getpid() returns the process ID (PID) of the calling process.
getppid() returns the process ID of the parent of the calling process.
*/

// int main() {
//     pid_t pid = fork();
//     if (pid < 0) {
//         return errno;
//     }
//     else if (pid == 0) {
//         // child
//          printf("Child - PID = %d\n", getpid());
//          printf("Child - parent PID = %d\n", getppid());
//      }
//      else {
//          // parent
//          printf("Parent - PID = %d\n", getpid());
//          printf("Parent - parent PID = %d\n", getppid());
//      }
//      return 0;
// }



// // De cate ori se afiseaza "Hi!" ?
// int main() {
//      fork();
//      fork();
//      printf("Hi!\n");
//      return 0;
// }

// // Dar acum?
// int main() {
//      fork();
//   fork();
//      fork();
//   printf("Hi!\n");
//      return 0;
// }



/*
The wait() system call suspends execution of the calling thread until
one  of its children terminates.
*/

// int main(){
//      pid_t pid = fork();
//      if (pid < 0) {
//          return errno;
//      }
//     else if (pid == 0) {
//         // child
//         for (int i = 0; i < 1000000; i++){
//             printf("child: %d\n", i);
//         }
//         exit(0);
//     }
//     else {
//         // parent
//         wait(NULL);
//         // apeland functia de sistem wait, parintele va intra in for
doar dupa ce procesul copil a terminat executia propriului for
//         for (int i = 0; i < 1000000; i++){
//             printf("parent: %d\n", i);
//         }
//     }
//     return 0;
// }



/*
int execve(const char *pathname, char *const argv[], char *const envp[]);

execve() executes the program referred to by pathname.  This causes the
program that is currently being run by the calling process  to  be  re-
placed  with  a  new  program,  with newly initialized stack, heap, and
(initialized and uninitialized) data segments.

argv  is  an  array  of argument strings passed to the new program.
By convention, the first of these strings (i.e., argv[0])  should  contain
the filename associated with the file being executed.  envp is an array
of strings, conventionally of the form key=value, which are  passed  as
environment to the new program.  The argv and envp arrays must each in-
clude a null pointer at the end of the array.
*/

// int main(){
//     pid_t pid = fork();
//     if (pid < 0) {
//         return errno;
```

```c
//      }
//      else if (pid == 0) {
//          // child
//          char *argv[] = {"/bin/ls", "-l", NULL};
//          execve("/bin/ls", argv, NULL);
//      }
//      else {
//          // parent
//          wait(NULL);
//      }
// }


/*
Sa se creeze un program care primeste ca argumente din linia de comanda un
numar de intregi si
afiseaza pentru fiecare numar n, primele n numere din sirul Fibonacci.
Pentru fiecare numar n,
sa se creeze un nou proces care calculeze sirul.
*/

int main(int argc, char* argv[])
{
   printf("Starting parent %d\n",getpid());
   for (int i = 1; i < argc; i++) {
       pid_t child = fork(); // un copil pt fiecare task
       if (child < 0) {
           perror("Eroare la fork");
           return -1;
       }
       else if(child == 0)
       {
           int n = atoi(argv[i]);
           printf("%d: ",n);
           if (n < 1) {
               printf("n has to be > 0\n");
           }
           else {
               int f1 = 0, f2 = 1, i;
               printf("%d ", f1);
```

```
                for (i = 1; i < n; i++) {
                    printf("%d ", f2);
                    int next = f1 + f2;
                    f1 = f2;
                    f2 = next;
                }
                printf("\n");
            }
            printf("Done. Parent = %d, Me = %d\n", getppid(), getpid());
            exit(0); //normal process termination
            // trebuie sa terminam procesul pentru ca altfel continua in
for
            /*
                Q:copilul vede doar in block-ul in care i s-a facut fork?
                A:Nu. Tot codul. E efectiv o copie a parintelui. Dupa fork
executa
                fiecare instructiune care urmeaza, la fel ca parintele.
Singura diferenta
                e ca daca intrebi copilul ce a returnat forkul de mai sus
zice ca 0. Parintele
                o sa zica ca a returnat un nr diferit de 0, mai exact
pid-ul copilului.
                Conform definitiei lui fork, se copiaza inclusiv, stack
pointerul, frame pointerul, etc
            */
        }
    }
    for(int i=1;i<argc;i++)
    {
        //parintele asteapta terminarea executiei fiecarui copil
        wait(NULL);
        //daca se apeleaza wait si nu exista copil de asteptat cred ca se
returneaza -1
    }
    printf("Parent done. Parent = %d, Me = %d\n", getppid(), getpid());

    return 0;
}

/*
```

```
./a.out 4 5 6
Starting parent 358376
4: 0 1 1 2
Done. Parent = 358376, Me = 358377
5: 0 1 1 2 3
Done. Parent = 358376, Me = 358378
6: 0 1 1 2 3 5
Done. Parent = 358376, Me = 358379
Parent done. Parent = 350414, Me = 358376
*/
```



Analizați secvența de mai jos și alegeți afirmația corectă:

```
pid_t pid = fork();
if (pid)
    exit(0);
else {
  sleep(5);
  printf("Hello, World!\n");
}
return 0;
```

○ a. La sfârșitul execuției procesul părinte este blocat.

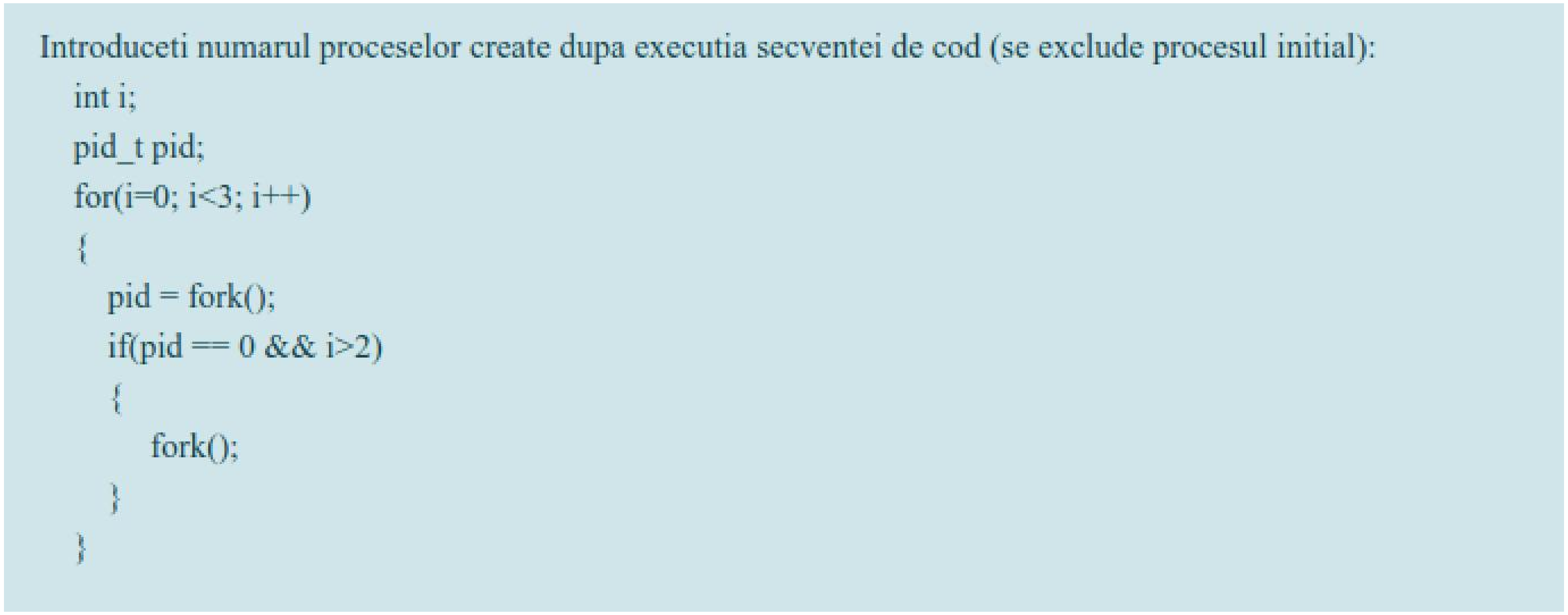
○ b. La sfârșitul execuției procesul copil este orfan.

○ c. La sfârșitul execuției procesul copil este zombie.

○ d. La sfârșitul execuției ambele procese se încheie corect.

○ e. La sfârșitul execuției procesul părinte este zombie.

Șterge alegerea mea



Introduceti numarul proceselor create dupa executia secventei de cod (se exclude procesul initial):

```
int i;
pid_t pid;
for(i=0; i<3; i++)
{
    pid = fork();
    if(pid == 0 && i>2)
    {
        fork();
    }
}
```

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int value = 5;

int main()
{
pid_t pid;

   pid = fork();

   if (pid == 0) { /* child process */
      value += 15;
      return 0;
   }
   else if (pid > 0) { /* parent process */
      wait(NULL);
      printf("PARENT: value = %d",value); /* LINE A */
      return 0;
   }
}
```

Figure 3.30 What output will be at Line A?

```
#include <stdio.h>
#include <unistd.h>

int main()
{
    /* fork a child process */
    fork();

    /* fork another child process */
    fork();

    /* and fork another */
    fork();

    return 0;
}
```

Figure 3.31 How many processes are created?

```
#include <stdio.h>
#include <unistd.h>

int main()
{
    int i;

    for (i = 0; i < 4; i++)
       fork();

    return 0;
}
```

Figure 3.32 How many processes are created?

3.5 When a process creates a new process using the `fork()` operation, which of the following states is shared between the parent process and the child process?

a. Stack

b. Heap

c. Shared memory segments