1

The following code compiles on Linux. It has a number of problems, however. Please locate as many of those problems as you are able and provide your recommendations regarding how they can be resolved.

At first of all I assume that access to list methods doesn't happend in different threads, in other case we must prevent race condition using lock/lock-free algorithms.

```
typedef struct list_s {
        struct list_s *next; /* NULL for the last item in a list */
        int data;
} list_t;
```

1.1 Counts the number of items in a list

```
/* Counts the number of items in a list. */
int count_list_items(const list_t *head)
{
        if (head->next) { /* (1) */
                return count_list_items(head->next) + 1; /* (2) */
        } else {
                return 1; /* (3) */
        }
}
```

- segfault on head == NULL (1)
- stack usage without tail call optimization (2);
- function doesn't return 0, list can't be empty? (3)

Solution

```
/* Counts the number of items in a list. */
int count_list_items(const list_t *head)
{
        int count = 0;

        while (head) {
                ++count;
                head = head->next;
        }

        return count;
}
```

1.2 Inserts a new list item after the one specified as the argument

```
/* Inserts a new list item after the one specified as the argument. */
void insert_next_to_list(list_t *item, int data)
{
        (item->next = malloc(sizeof(list_t)))->next = item->next; /* (1) */
        item->next->data = data;
}
```

- unchecked result of `malloc` (1).
- lost `item->next` (or UB???) (1).
- remark: what should this function do if it gets null pointer as item argument? how caller does know about failed malloc?

Solution

```
/* Inserts a new list item after the one specified as the argument.
   \return new item pointer
 */
list_t * insert_next_to_list(list_t *item, int data)
{
        list_t *n = malloc(sizeof(list_t));
        if (!n)
                return NULL;

        n->data = data;

        if (!item) {
                n->next = NULL;
        } else {
                n->next = item->next;
                item->next = n;
        }

        return n;
}

/* example */
list_t * create_random_list(int count)
{
        list_t *head = NULL, *next = NULL;

        if (count <= 0)
                return NULL;

        next = head = insert_next_to_list(head, rand());
        while (next && --count > 0) {
                next = insert_next_to_list(next, rand());
        }

        return head;
}
```

1.3 Removes an item following the one specificed as the argument

```
/* Removes an item following the one specificed as the argument. */
void remove_next_from_list(list_t *item)
{
        if (item->next) { (1)
                free(item->next);
                item->next = item->next->next; (2)
        }
}
```

- segfault on `item == NULL` (1).
- usage-after-free (2)

```
/* Removes an item following the one specificed as the argument. */
void remove_next_from_list(list_t *item)
{
        if (item && item->next) {
                list_t *next = item->next;
                item->next = next->next;
                free(next);
        }
}
```

1.4 Returns item data as text

```
/* Returns item data as text. */
char *item_data(const list_t *list)
{
        char buf[12];
        sprintf(buf, "%d", list->data);
        return buf; /* (1) */
}
```

- returning a pointer to automatic storage

Solution 1: usage static TLS (pros: no tracking for leaks; cons: deep copy for sequential calls).

```
/* Returns item data as text. */
char *item_data(const list_t *list)
{
        static __thread char buf[12];
        sprintf(buf, "%d", list->data);
        return buf;
}

void example()
{
        list_t *head;

        // .. creation of list ...

        // ok and fast
        printf("%s\n", item_data(head));

        // invalid, deep copy is required
        printf("%s, %s\n", item_data(head), item_data(head->next));
}
```

Solution 2: returning allocated buf.

```
/* Returns item data as text. */
char *item_data(const list_t *list)
{
        char *buf = malloc(12);
        sprintf(buf, "%d", list->data);
        return buf;
}

void example()
{
        list_t *head;

        // .. creation of list ...

        // don't forget save and free retval at the end
        char *val = item_data(head);
        printf("%s\n", val);
        free(val);
}
```

2

By default, Linux builds user-space programs as dynamically linked applications. Please describe scenarios where you would change the default behavior to compile your application as a statically linked one.

At first of all statically linked binaries are faster to load. But in my experience we made statically linked binaries as the simplest way of distribution, exactly if an application depends on an unusual library or specific library version.

3

Develop your version of the ls utility. Only the 'ls -l' functionality is needed.

See source file *ls.c* in the attachment.

4

Please explain, at a very high-level, how a Unix OS, such as Linux, implements the break-point feature in a debugger.

The main idea is based on hardware support. There is an instruction (int3 for x86) which handled by a CPU like an exception. Size of this instruction is 1 byte, which allows to substitute any instruction.

So algorithm of insertion a breakpoint is:

1. changing page with required instruction to the read-write mode;
2. substitution of required instruction by a int3 and remembering this association;

When a CPU meets int3 it generates a hardware interrupt which is handled by a kernel. If a process with this instruction is not under debug, it's stopped. In other case a debugger get control.

5

Suppose you debug a Linux application and put a break-point into a function defined by a dynamically-linked library used by the application. Will that break-point stop any other application that makes use of the same library?

No, it won't. Since page's changed by int3 substitution, copy-on-write engine makes unique copy of this page, so only debugged process sees this changes.

6. Please translate in English. Don't metaphrase, just put the idea as you see it into English.

Если пользовательский процесс начал операцию ввода-вывода с устройством, и остановился, ожидая окончания операции в устройстве, то операция в устройстве закончится во время работы какого-то другого процесса в системе. Аппаратное прерывание о завершении операции будет обработано операционной системой. Так как на обработку прерывания требуется некоторое время, то текущий процесс в однопроцессорной системе будет приостановлен. Таким образом, любой процесс в системе непредсказуемо влияет на производительность других процессов независимо от их приоритетов.

If a userspace process initiates i/o operation with a device, it's stops awaiting completion of this operation. As i/o operations happen asynchronously with any process in a system, the i/o operation will complete while the CPU is running another process in the system. The hardware interrupt on the i/o operation completion will be handled by an operating system. Since the interrupt handler requires CPU time, current process in uniprocessor system is going to be stopped. So any process in a system inadvertently affects to other processes performance despite their priorities.