Forfattere:
13.07-89 Rane Eriksen BVL193
05.01-90 Christian Nielsen BNF287
14.06-79 Sebastian O. Jensen GJX653
Hold 1

Datalogisk Institut

Københavns Universitet

Operativsystemer og Multiprogrammering

G-Assignment 3

2. marts 2015

Indhold

1 Task 1: A thread-safe unbounded queue

implementation

In this section we will explain how we implemented our thread-safe unbounded queue.

We chose to ignore the example provided by the assignment and wrote our own dynamic array.

These are the most essential parts of our program and is a snippet from our queue.h file

```c
typedef struct {
    void *item;
    pthread_mutex_t lock;
} node_t;

typedef struct {
    node_t *array;
    int max_size;
    pthread_mutex_t update_array_lock;
    pthread_cond_t update_condition;
    pthread_mutex_t users_lock;
    pthread_cond_t users_condition;
    int users;
    int is_updating;
} list_t;
```

As we can see in the snippet we have an array inside the list_t struct which contains every node_t struct that is being inserted in our implementation by multiple threads.

We have decided to keep track of how many threads are currently writing or reading from the array, to do this we use two pthread_mutex_t, two pthread_cond_t variables and two int variables to keep track of when we may expand our array and when we are allowed to either read or write in our array.

In our implementation we have four basic functions

- void queue_init_list_t(list_t *list)
  Initializing the queue is expected to be done in a single thread since it does not take into account that multiple might want to initialize the same list which would reset the list and cause a memory leak if any of the entries in the array are filled with allocated data.

- void queue_put_list_t(list_t *list, void *item)
  In this function we are inserting an item in to the list array object, we do this by iterating through every entry in the array untill we acquice a mutex lock, ince acquired we check if the item is NULL, if it is, we keep looking in the array.
  Because the array must be dynamic we want to have a way to handle expansion of the array but we also want to handle when we are allowed to expand the array and when we are allowed to insert into the array.

```c
void queue_put_list_t(list_t *list, void *item) {
    pthread_mutex_lock(&list->update_array_lock);
    while (list->is_updating != 0) {
        pthread_cond_wait(&list->update_condition, &list->
            update_array_lock);
    }
    pthread_mutex_unlock(&list->update_array_lock);
    ...
}
```

In this code snippet we are waiting for the up_updating to be 0, once it is we may write into the array, otherwise we are waiting until the condition is signaled to resume.

```c
 1 void queue_put_list_t(list_t *list, void *item) {
 2     ...
 3     pthread_mutex_lock(&list->users_lock);
 4     list->users++;
 5     pthread_mutex_unlock(&list->users_lock);
 6     ...
 7     pthread_mutex_lock(&list->users_lock);
 8     list->users--;
 9     pthread_cond_signal(&list->users_condition);
10     pthread_mutex_unlock(&list->users_lock);
11     ...
12 }
```

in these sections of code we are increasing and decreasing how many are inserting into the array.
on line 9 we see that we are signaling a thread that is listening to users_condition variable that a user has exited, this is how we keep track of when it is allowed to expand the array - When users is 0.
We will see how these variables are used in queue_expand_list_t.

```
void queue_put_list_t(list_t *list, void *item) {
    ...
      for (int i = 0; i < max_size; i++) {
        if (pthread_mutex_trylock(&list->array[i].lock) ==
            0) {
          if (list->array[i].item == NULL) {
            list->array[i].item = item;
            found = 1;
            pthread_mutex_unlock(&list->array[i].lock);
            break;
          }
          pthread_mutex_unlock(&list->array[i].lock);
        }
      }
    ...
}
```

This is our inserting algorithm, we realize that the insertion time is very slow which is also why we are using trylock instead of a regular lock, one reason is speed, it will not block and our reason using this type is, that we do not want to wait since another entry is most likely free to lock, the other reason is that it is very likely that another thread is writing into that very entry we want to lock and so when obtaining the there is a very high risk that the item is not NULL and we will have to check the next again, but with trylock this will be very fast since it's not required to block, it will just skip the entry if it is not able to lock.

- void *queue_get_list_t(list_t *list)
  The algorithm is exactly the same as putting an item into the array except that we are taking an item from the array and setting it's value to NULL on success, this way we free up the entry for the next item to be inserted.

- void queue_expand_list_t(list_t *list, void *item)
  This function handles the expansion of a list and is only called by void queue_put_list_t when it fails to insert an item in that it assumes that the array must be full since it failed to insert.

```
 1 void queue_expand_list_t(list_t *list, void *item) {
 2     int i_update = 0;
 3     pthread_mutex_lock(&list->update_array_lock);
 4     if (list->is_updating == 0) {
 5         list->is_updating = 1;
 6         i_update = 1;
 7     }
 8     pthread_mutex_unlock(&list->update_array_lock);
 9     if (i_update == 1) {
10       pthread_mutex_lock(&list->users_lock);
11       while (list->users > 0) {
12           pthread_cond_wait(&list->users_condition, &list->
                 users_lock);
13       }
14       pthread_mutex_unlock(&list->users_lock);
15       ...
16       list->is_updating = 0;
17       pthread_cond_broadcast(&list->update_condition);
18     }
19     queue_put_list_t(list, item);
20 }
```

In this snippet we see how we make sure that only one thread expands the array to avoid undefined behaviour.
The first thread to acquire the lock will set the is_updating variable to 1 and also set a local variable i_update to 1.
Any other thread getting a lock will do nothing and just skip the entire function and retry inserting the item as we see on line 13.

The most important part to notice is on line 10 through17.
Because we have set the is_updating to 1, any thread trying to either get or insert an item into the array is told to wait for is_updating to turn to 0 which means that we need not hold a lock while updating the array because only one thread is guaranteed to access that particular area of code at a time and only when the array appears to be empty.
Once the array has been expanded to twice its size we simply set the is_updating to 0 and tell every thread waiting for that condition to be true that they may continue from where they stopped, we use broadcast in this case because one or more thread may be waiting for the condition and signal would only tell one thread to continue.

Tests

2 Task 2: Userland semaphores for Buenos

Where we made changes

In this section we will only focus on where we made changes in order to implement our solution to userland semaphores.
We will explain what we implemented and why later, this section is only to create a simple overview of what once was and now is.

init/main.c

This section initializes the userland semaphores along with every other initialization in init/main.c

```
kwrite("Initializing userland semaphores\n");
userland_semaphore_init();
```

proc/syscall.h

We added this line because in order to follow the standards of the current buenos

```
#define SYSCALL_SEM_CLOSE  0x303
```

proc/syscall.c

Add something to this section!

tests/lib.h

In order to avoid name clashes of function in different areas of the buenos we renamed a few functions in tests/lib.c. The changes are as follows:
'putc' was changed to 'putc_userland'
'strlen' was changed to 'strlen_userland'

The functions below are included in this header file and are explained in detail in the next section

```
usr_sem_t* syscall_sem_open(char const* name, int value);

int syscall_sem_p(usr_sem_t* handle);
int syscall_sem_v(usr_sem_t* handle);

int syscall_sem_destroy(usr_sem_t* handle);
```

tests/lib.c

In order to avoid name clashes of function in different areas of the buenos we renamed a few functions in tests/lib.c. The changes are as follows:
'putc' was changed to 'putc_userland'
'strlen' was changed to 'strlen_userland'
In this file we also added four functions that are visible to any program programmed to run in the buenos OS and acts as a link between the operating system and user program.

The four functions are as follows:

A system call to this function enables a user program to create a semaphore or access a semaphore that is already created. How this function works is explained in subsection proc/semaphore.c

```
usr_sem_t* syscall_sem_open(char const* name, int value) {
  return (usr_sem_t*)_syscall(SYSCALL_SEM_OPEN,
                              (uint32_t)name, (uint32_t)value, 0);
}
```

A system call to this function enables a user program or process to block itself untill another program or process releases the block. How this function works is explained in subsection proc/semaphore.c

```
int syscall_sem_p(usr_sem_t* handle) {
  return (int)_syscall(SYSCALL_SEM_PROCURE,
                       (uint32_t)handle, 0, 0);
}
```

A system call to this function enables a user program or process to release another program or process that has blocked itself by the calling syscall_sem_p. How this function works is explained in subsection proc/semaphore.c

```
int syscall_sem_v(usr_sem_t* handle) {
  return (int)_syscall(SYSCALL_SEM_VACATE,
                       (uint32_t)handle, 0, 0);
}
```

A system call to this function enables a user program or process to close a semaphore if no other program or process is blocked by it. How this function works is explained in subsection proc/semaphore.c

```
int syscall_sem_destroy(usr_sem_t* handle) {
  return (int)_syscall(SYSCALL_SEM_CLOSE,
                       (uint32_t)handle, 0, 0);
}
```

proc/semaphore.h

We took what was already implemented in kernel/semaphore.h and modified the existing code to serve the needs to hold a name as identifier for the semaphore and to bring a semaphore to buenos userland which is not related to the semaphores used by buenos kernel.
Instead of having creator point to a thread we made it point to the process which created the semaphore - the creator id is strictly used to see if the semaphore is in use or if it's available.

The maximum length of the semaphore is currentl 256, but this is only due to execute or tests that needs the name to be minimum 129 characters long. The maximum length of the name of the semaphore should not need to be longer than 128 or 64 characters long because we estimate that developers would rather keep short and precise names rather than long names in the programming code.

```
#ifndef BUENOS_USERLAND_SEMAPHORE_H
#define BUENOS_USERLAND_SEMAPHORE_H

#define MAX_SEMAPHORES 128
#define MAX_SEMAPHORE_NAME 256

#define FAILURE -1

#include "kernel/spinlock.h"
#include "proc/process.h"

typedef struct {
    char name[MAX_SEMAPHORE_NAME];
    spinlock_t slock;
    int value;
    process_id_t creator;
} usr_sem_t;

void userland_semaphore_init(void);
usr_sem_t *userland_semaphore_create(char const* name, int value);
int userland_semaphore_P(usr_sem_t *sem);
int userland_semaphore_V(usr_sem_t *sem);
int userland_semaphore_destroy(usr_sem_t *sem);

#endif /* BUENOS_USERLAND_SEMAPHORE_H */
```

proc/semaphore.c

In this section we will only include the most essential parts that differ from the kernel semaphore in an attempt to keep this report short and precise.

As stated in the previous subsection we took the semaphore from the kernel and modified it to fulfill the need to hold a name.

The initializer function does the exact same thing as the kernel version except that it sets the first character of the name to the null terminating string.

‘userland_semaphore_create‘ is split up in two functions and the name may be misleading as name, but it is divided in to two functions:
‘find_existing_semaphore‘ and ‘create_fresh_semaphore‘ are the two functions that actually defines ‘userland_semaphore_create‘, the decision between the two functions are based on the value that is passed to ‘userland_semaphore_create‘. If the value is negative ‘find_existing_semaphore‘ is called and if the value is equal to or higher than zero ‘create_fresh_semaphore‘ is called.

‘create_fresh_semaphore‘ does as titled, creates a new semaphore if one is available however it needs to check every entry in the table to make sure that no other semaphore by the name given already exists, the algorithm therefore runs in $\Theta(n)$ in worst case which makes it expensive, the reason it’s expensive compared to the one in the kernel which also runs $\Theta(n)$ in worst case is, that the one we have implemented must keep checking for existing semaphores with the name provided even though it has determined that a semaphore is available.

A way to make it faster would be to index the names in alphatecially order which would drastically speed up the function however indexing might be time consuming and there’s always the case where a programmer always starts the name of the semaphores with a prefix which would then affect the indexing of the names in a worse way, also there is the consideration of the cost when actually indexing the names which could also be expensive.

If the function created a semaphore the pointer to the semaphore is returned else NULL is returned.

'find_existing_semaphore' also does as titled, it searches for a semaphore with the given name the running time is also $\Theta(n)$ however it has only one condition, which makes it faster in most cases.

If the semaphore exists the pointer to the semaphore is returned otherwise NULL is returned

'userland_semaphore_P' the only difference from this function and the function used by the kernel is, that we do not trust programmers, we therefore test for an initialized semaphore. If the semaphore 'creator' value is -1 the function returns -1 to indicate an error otherwise 0.
There's a possibility that the function simply does not exist in memory, however TLB exception should handle this case.

'userland_semaphore_V' differs from the kernel version in the same way the 'find_existing_semaphore' differs from kernel.
The return value is the exact same as well under the same circumstances.

Lastly comes 'userland_semaphore_destroy' which enables a program to close a semaphore thus making it available to other programs, it is very important that a program or process closes every semaphore created in order to prevent a memoryleak since buenos will not destroy these as the process is terminated in any way.

A semaphore can be used across different processes which is why we did not implement termination for every semaphore once its process terminates - This is left to the programmer.

Tests

In order to test our implementation we have created a rather large test program consisting of three programs:
semaphore_test.c
semaphore_test2.c and

semaphore_test3.c.

Before we discuss our tests we want to make clear what we want to test.

Regarding creating or reopening a semaphore we want to test that we get the correct return values based on the input we give - We want to test that when we call ‘userland_semaphore_create‘ with a given name and a value that is at least 0 or higher that wea given a new semaphore and in case of errors that we get NULL.
The same test goes for when the value is negative.

We also want to test that when ‘userland_semaphore_P‘ the program or process calling the function is blocked, if the value is less than 0.
Next to this we want to test that calling ‘userland_semaphore_V‘ results in unblocking one program or process that was blocked by the semaphore.

Finally we want to test that a program can close a semaphore if no process is blocked by it and because it is possible to close a semaphore and make it available again we want to test that obtaining every semaphore possible by creating 128 semaphores which is our limit by implementation, closing 128 should then free 128 semaphores again.

‘semaphore_test.c‘ is the initprog in our testcase and is responsible for creating new process that have their own thread - This is needed to test if a semaphore is blocked correctly because a process cannot block and unblock itself since that would defeat the purpose.
‘semaphore_test.c‘ starts by creating a semaphore it then creates 15 new processes(‘semaphore_test2.c‘) that will use the semaphore created and block themselves.
To test that these are blocked a third program is started(‘semaphore_test3.c.‘).
This will try to close the semaphore that ‘semaphore_test2.c‘ is blocked by, the test is, that is should not be allowed to close the semaphore before every process(‘semaphore_test2.c‘) has left the sleep_queue.
For every time the while loop in ‘semaphore_test3.c‘ runs, ‘semaphore_test3.c‘ calls ‘userland_semaphore_V‘ and decreases the value by one and unblocks one process(‘semaphore_test2.c‘).
Finally it exits and ‘semaphore_test.c‘ continues to the next test.

The next test tries to create 129 semaphores which should result in a failure once it tries to create number 129, the rest should be created with success.
Once the 129 has been created with or without failure, we test that they can

also be destroyed by destroying the 128 semaphores we have just created, we loop through 129 entries, but the number 129 semaphore does not exist thus it results in an error which is intended.
Finally we try to create 128 semaphores as before to test that all was closed with success, we then close them immediatly after since we stated previously that it was important to close them for the sake of the buenos system.

In case of failure in our test the system will halt.

In our enviroment the tests run through every test with succcess.