Bivas Maiti
bmaiti@iu.edu

Assignment Futures 1

In this assignment, we implement futures in XINU. We have 3 modes for futures-FUTURE_EXCLUSIVE, FUTURE_SHARED and FUTURE_QUEUE.

In our implementation, futures are struct objects, defined as below:

```
typedef struct {
  int value;
  future_state_t state;
  future_mode_t mode;
  pid32 pid;
  future_item *set_queue;
  future_item *get_queue;
} future_t;
```

Get_queue and Set_queue are linked lists, whose HEAD values are stored in the future. Below is the definition of the linked list items.

```
typedef struct{
        pid32 pid;
        struct future_item *fqnext;
} future_item;
```

Here we are only storing the pid of the process which is waiting in the queue, and pointer for the next item in the queue.

Implementation of FUTURE_EXCLUSIVE

Implementation of this mode is done via linked lists for get_queue and set_queue. Here we can have only 1 item in get_queue, and none in set_queue.

Allocation of the Future

Allocation is done by the function future_alloc, which takes a mode, and allocates space for a new future item to be stored. It sets the state of the future as FUTURE_EMPTY, and the get_queue and set_queue values to NULL.

There are 2 functions future_get and future_set which are used to store and retrieve values to and from a future.

future_get(future_t* f,int* address)

Here we take the pointer of the future t as an argument, and address to which the values needs to be stored.

Here we check if the future is empty. If it is, we enqueue the process in the get_queue, then change the status to FUTURE_WAITING, and suspend it.

If it is ready, we simply retrieve the value from the future and store it in the address from the argument and change the status back to FUTURE_EMPTY.

Once it gets woken up(resumed) by the future_set function, we retrieve the value from future and store it in the address (from the argument)

If the state is already waiting, we can't do anything, since in FUTURE_EXCLUSIVE there can't be multiple get processes waiting for a value. So we return SYSERR.

future_set(future_t* f,int value)

Here we take the pointer of the future as and the value to put in the future as the arguments.

If the state of the future is empty, we simply store the value, change the status to FUTURE_READY and return.

If it is waiting, we store the value and resume the process from get_queue which was suspended waiting for the value we stored. There can only be 1 such process in FUTURE_EXCLUSIVE mode. So, if the status of the future is already in READY state, and future set is called, it will throw a SYSERR.

Implementation of FUTURE_SHARED

Implementation of this mode is done via linked lists for get_queue and set_queue. Here we can have multiple item in get_queue, and only 1 in set_queue.

Allocation of the Future

Allocation is done by the function future_alloc, which takes a mode, and allocates space for a new future item to be stored. It sets the state of the future as FUTURE_EMPTY, and the get_queue and set_queue values to NULL.

There are 2 functions future_get and future_set which are used to store and retrieve values to and from a future.

future_get(future_t* f,int* address)

Here we take the pointer of the future t as an argument, and address to which the values needs to be stored.

First we check if the future is empty. If it is, we enqueue the process in the get_queue and suspend it. We then change the status to FUTURE_WAITING.

If it is ready, we simply retrieve the value from the future and store it in the address from the argument and change the status to FUTURE_EMPTY.

If the state is already waiting, we traverse through the get_queue queue and store the process at the end of the queue. Then we suspend it.

Once it gets woken up(resumed) by the future_set function, we retrieve the value from future and store it in the address(from the argument)

future_set(future_t* f,int value)

Here we take the pointer of the future as and the value to put in the future as the arguments.

If the state of the future is empty, we simply store the value, change the status to FUTURE_READY and return.

If it is waiting, we store the value and resume all the waiting processes from get_queue by traversing the queue, which were suspended waiting for the value we just stored. There can be multiple waiting processes in the FUTURE_SHARED mode. After all the processes are resumed, we change back the status to FUTURE_EMPTY.

If the status of the future is already in READY state, and future set is called, it will throw a SYSERR.

Implementation of FUTURE_QUEUE

Implementation of this mode is done via linked lists for get_queue and set_queue. Here we can have multiple items in both get_queue and set_queue.

Allocation of the Future

Allocation is done by the function future_alloc, which takes a mode, and allocates space for a new future item to be stored. It sets the state of the future as FUTURE_EMPTY, and the get_queue and set_queue values to NULL.

There are 2 functions future_get and future_set which are used to store and retrieve values to and from a future.

future_get(future_t* f,int* address)

Here we take the pointer of the future t as an argument, and address to which the values needs to be stored.

First we check if the future is empty. If it is, we enqueue the process in the get_queue and suspend it. We then change the status to FUTURE_WAITING.

If it is FUTURE_READY, we retrieve the value from the future and store it in the address from the argument. Then we check if the set_queue is empty. If the set_queue contains processes waiting, we resume the first process from it and dequeue the process from set_queue. If there are no more processes waiting, we change the status to FUTURE_EMPTY.

If the state is already waiting, we traverse through get_queue and store the process at the end of the queue. Then we suspend it.

Once it gets woken up(resumed) by the future_set function, we retrieve the value from future and store it in the address (from the argument)

Bivas Maiti
bmaiti@iu.edu

future_set(future_t* f,int value)

Here we take the pointer of the future as and the value to put in the future as the arguments.

If the state of the future is empty, we simply store the value and return.

If it is waiting, we store the value and resume the first process from it and dequeue the process from get_queue. If there are no more processes waiting, we change the status to FUTURE_EMPTY.

If the status is FUTURE_READY, we traverse through the set_queue queue and store the process at the end of the queue. Then we suspend it.

Free allocated memory

The function future_free is used to free the futures. It will check if there are any waiting processes in either of the queues, then resume the processes and free the queues. After that it frees the future object altogether.

Key points in the implementation (for all modes):

- Function future_set resumes processeses from get_queue
- Function future_get resumes processeses from set_queue
- FUTURE_READY means there can be 0 to n processes in the set_queue but none in the get_queue.
- FUTURE_WAITING means there can be 1 to n processes in the get_queue but none in the set_queue.
- FUTURE_EMPTY means there aren't any processes waiting in either queue.
- Interrupts are disabled before each implementation and restored in each function exit. (get and set)