FUTURES IMPLEMENTATION IN XINU

Introduction:

The design document provides an overview of the futures concept part of the new C++ standard. The futures variable is a placeholder for an eventual output generated by a function called asynchronously.

The asynchronous function can then use a variety of methods to query, wait for, or extract a value from the future variable. These methods may block if the asynchronous operation has not yet provided a value.

Design Considerations

- The program has producers and consumers of data. The producer outputs a value and sets the future variable using future_set() and the consumer consumes data and it queries the future variable using the future_get() as per our implementation.
- If a thread calls future_get() on an empty future, then the calling thread should block and subsequent future_get() calls should fail. If a thread calls future_set() on an empty future, then it becomes full and a subsequent future_set() should fail. Calling future_get() on a full future yields the value and resets the future's state to empty.

Implementation Details

- Header file <future.h> contains function prototype and the structure corresponding to the future variable.

```
struct futent          /*future table entry*/
{
        int data;          /* Data that future should hold */
        char state;   /*state of the future which can be FUT_FREE or FUT_USED*/
        int flag;          /*flag to check if a process is waiting for the future*/
        int tid;           /*process id waiting for the future*/
};
```

- future futalloc() in futcreate.c

    - ➢ The function futalloc () returns the ID of the allocated future to f1.

```
future f1;
f1=futalloc();
```

- syscall future_set(future fut, int value)
    - The function is implemented as a system call and it is used to set the data of the future.
- syscall future_get(future fut, int *value)
    - The function is implemented as a system call and it is used to get the data stored in future.
- syscall future_free(future fut)
    - The function is implemented as a system call to free up the future variable.

Input snapshot

```
int a,b; a=10; b=20;
future f1,f2;
f1=futalloc();
f2=futalloc();
resume(create(consumer,1024,20,"cons1",1,f1));
resume(create(consumer,1024,20,"cons3",1,f1));
resume(create(producer,1024,20,"prod1",2,f1,a));
resume(create(producer,1024,20,"prod2",2,f2,b));
resume(create(producer,1024,20,"prod3",2,f2,b));
resume(create(consumer,1024,20,"cons2",1,f2));
```

Output snapshot

```
Output: Trying to access blocked future: Process cons3 blocked
Calling used future, Process prod3 blocked
The value is 10
The value is 20
```

- cons1 process executes and waits for the future f1 value and is invoked once future f1 value is set.
- cons3 process gets blocked since cons1 process is already waiting for future f1.
- cons2 process executes and waits for future f2 since the value is not set initially and resumes execution once future f2 value is set.
- prod3 process gets blocked since future f2 is already used by process prod2 and is full.

References

Wikipedia, CPP reference, Xinu Approach- Douglas Comer