Contribution

1. future.h Pratik Patel
2. xsh_prodcons.c Pratik Patel
3. future_cons.c Anand Nahar
4. future_prod.c Anand Nahar
5. future_alloc.c Anand Nahar
6. future_get.c Anand Nahar
7. future_set.c Pratik Patel
8. future_free.c Pratik Patel
9. fut_queue.h Anand Nahar
10. fut_queue.c Pratik Patel

Functions

future.h

Contains the declaration for MACROS, struct future and other functions.

Code

```
#ifndef _FUTURE_H_
#define _FUTURE_H_

#include <xinu.h>
#include <fut_queue.h>

/* define states */
#define FUTURE_EMPTY          0
#define FUTURE_WAITING        1
#define FUTURE_VALID          2

/* modes of operation for future*/
#define FUTURE_EXCLUSIVE  1
#define FUTURE_SHARED     2
#define FUTURE_QUEUE      3

typedef struct futent
{
  int *value;
  int flag;
  int state;
  pid32 pid;
  fut_queue *set_queue;
  fut_queue *get_queue;
}future;

extern int count;
```

```
/* Interface for system call */
future* future_alloc(int future_flags);
syscall future_free(future*);
syscall future_get(future*, int*);
syscall future_set(future*, int*);

int future_cons(future *fut);
int future_prod(future *fut);

#endif /* _FUTURE_H_ */
```

xsh_prodcons.c

Contains the declaration of future variable and thread creation logic.

Code

```
#include <prodcons.h>
#include <future.h>

int n=0;                //Definition for global variable 'n'
/*Now global variable n will be on Heap so it is accessible all the processes i.e.
consume and produce*/

sid32 produced,consumed;
future *f1,*f2,*f3,*f_exclusive,*f_shared,*f_queue;
int count;

shellcmd xsh_prodcons(int nargs, char *args[])
{
  //Argument verifications and validations

  int count;            //local varible to hold count
  int flag_sem=1;
  n=0;
  count=0;

  if (nargs == 2 && strncmp(args[1], "--help", 7) == 0)
  {
    printf("\nHELP");
    printf("\n\tProducer consumer problem");
    printf("\n\tUsage prodcons [number] -- default 2000");
    printf("\n\n\tFuture Implementation");
    printf("\n\tUsage prodcons -f");
    printf("\n\n\t--help\tdisplay this help and exit\n");
    return 0;
```

```
  }

  if(nargs>2)
  {
    fprintf(stderr,"\n%s: many Arguments...!!!",args[0]);
    fprintf(stderr,"\nUsage prodcons [number]");
    return 1;
  }
  else if(nargs==2)
  {
    if(strncmp(args[1],"-f",2)==0 && strlen(args[1])==2)
      flag_sem=0;
    else    //check args[1] if present assign value to count
    {
      count=atoi(args[1]);
      if(count<=0)
      {
        printf("\nPlease enter a valid value.",count);
        printf("\nUsage:");
        printf("\n\tprodcons [number] -- default 2000");
        printf("\n\tprodcons -f\n");
        return 1;
      }
    }
  }
  else
    count=2000;

  if(flag_sem)
  {
    produced = semcreate(0);
    consumed = semcreate(1);

    //create the process producer and consumer and put them in ready queue.
    //Look at the definations of function create and resume in exinu/system
folder for reference.
    resume( create(producer, 1024, 20, "producer", 1, count) );
    resume( create(consumer, 1024, 20, "consumer", 1, count) );
  }
  else
  {/*
    f1 = future_alloc(FUTURE_EXCLUSIVE);
    f2 = future_alloc(FUTURE_EXCLUSIVE);
    f3 = future_alloc(FUTURE_EXCLUSIVE);

    if(f1)
```

```
    {
      resume( create(future_cons, 1024, 20, "fcons1", 1, f1) );
      resume( create(future_prod, 1024, 20, "fprod1", 1, f1) );
    }
    else
      printf("\nError creating future f1");

    if(f2)
    {
      resume( create(future_cons, 1024, 20, "fcons2", 1, f2) );
      resume( create(future_prod, 1024, 20, "fprod2", 1, f2) );
    }
    else
      printf("\nError creating future f2");

    if(f3)
    {
      resume( create(future_cons, 1024, 20, "fcons3", 1, f3) );
      resume( create(future_prod, 1024, 20, "fprod3", 1, f3) );
    }
    else
      printf("\nError creating future f3");
*/
    f_exclusive = future_alloc(FUTURE_EXCLUSIVE);
    f_shared = future_alloc(FUTURE_SHARED);
    f_queue = future_alloc(FUTURE_QUEUE);

// Test FUTURE_EXCLUSIVE
    if(f_exclusive)
    {
      resume( create(future_cons, 1024, 20, "fcons1", 1, f_exclusive) );
      resume( create(future_prod, 1024, 20, "fprod1", 1, f_exclusive) );
    }
    else
      printf("\nError creating future f_exclusive");


// Test FUTURE_SHARED
    if(f_shared)
    {
      resume( create(future_cons, 1024, 20, "fcons2", 1, f_shared) );
      resume( create(future_cons, 1024, 20, "fcons3", 1, f_shared) );
      resume( create(future_cons, 1024, 20, "fcons4", 1, f_shared) );
      resume( create(future_cons, 1024, 20, "fcons5", 1, f_shared) );
      resume( create(future_prod, 1024, 20, "fprod2", 1, f_shared) );
    }
```

```
    else
      printf("\nError creating future f_shared");

// Test FUTURE_QUEUE
    if(f_queue)
    {
      resume( create(future_cons, 1024, 20, "fcons6", 1, f_queue) );
      resume( create(future_cons, 1024, 20, "fcons7", 1, f_queue) );
      resume( create(future_cons, 1024, 20, "fcons7", 1, f_queue) );
      resume( create(future_cons, 1024, 20, "fcons7", 1, f_queue) );
      resume( create(future_prod, 1024, 20, "fprod3", 1, f_queue) );
      resume( create(future_prod, 1024, 20, "fprod4", 1, f_queue) );
      resume( create(future_prod, 1024, 20, "fprod5", 1, f_queue) );
      resume( create(future_prod, 1024, 20, "fprod6", 1, f_queue) );
    }
    else
      printf("\nError creating future f_queue");
  }
  return 0;
}
```

future_cons.c

Consumes the values produced by the producer and also free's future.

Code

```
#include <future.h>

int future_cons(future *fut)
{
  int i, status;
  count++;
  status = future_get(fut, &i);
  count--;
  if (status < 1)
  {
    printf("future_get failed\n");
    return -1;
  }
  kprintf("\nConsumer consumed %d", i);

  if(count==0)
    if(!(future_free(fut)))
      return SYSERR;

  return OK;
```

```
}
```

future_prod.c

Responsible for producing the value that would be consumed by the consumer

Code

```
#include <future.h>

int future_prod(future *fut)
{
  int i,status;
  int j;
  j = (int)fut;

  for (i=0; i<1000; i++)
  {
    j += i;
  }
  kprintf("\nProducer produced %d",j);
  status=future_set(fut, &j);
  if (status < 1)
  {
    printf("future_set failed\n");
    return -1;
  }
  return OK;
}
```

future_alloc.c

Allocates memory to future variable and also to value variable inside the future.

Code

```
#include <future.h>

future* future_alloc(int future_flag)
{
  future *f;
  intmask mask;

  mask=disable();

  f=(future *)getmem(sizeof(future));    //allocating memory to new future

  if(f==NULL)
  {
    printf("\nError allocating memory for future variable");
```

```
    restore(mask);
    return NULL;
  }

  f->value=(int *)getmem(sizeof(int));   //allocating to member of struct future

  if(f->value==NULL)
  {
    printf("\nError allocating memory for value in future variable");
    restore(mask);
    return NULL;
  }

  f->set_queue=fut_qcreate();
  if(f->set_queue==NULL)
  {
    restore(mask);
    return NULL;
  }

  f->get_queue=fut_qcreate();
  if(f->get_queue==NULL)
  {
    restore(mask);
    return NULL;
  }

  f->flag=future_flag;       //initializing flag for EXCLUSIVE mode
  f->state=FUTURE_EMPTY;     //initializing state of the variable
  f->pid=-1;  //initializing pid
  *(f->value)=0;
  restore(mask);
  return f;
}
```

future_free.c

Free the memory allocated for the the value and future variables

Code

```
#include <future.h>

syscall future_free(future* f)
{
  intmask mask;
  mask=disable();
```

```
  if(!freemem(f->value,sizeof(int)))
  {
    restore(mask);
    return -1;
  }

//free all waiting nodes also

  if(!freemem(f->set_queue,sizeof(fut_queue)))
  {
    restore(mask);
    return -1;
  }

  if(!freemem(f->get_queue,sizeof(fut_queue)))
  {
    restore(mask);
    return -1;
  }

  if(!freemem(f,sizeof(future)))
  {
    restore(mask);
    return -1;
  }
  restore(mask);
  return 0;
}
```

future_get.c

Consumer calls future_get in order to fetch the value present in future variable

Code

```
#include <future.h>

syscall future_get(future *f, int *value)
{
  pid32 temp_pid;

  if(f->flag==FUTURE_EXCLUSIVE)
  {
    if(f->state==FUTURE_WAITING)
    {
      return SYSERR;
    }
```

```
  if(f->state==FUTURE_EMPTY)
  {
    f->pid=getpid();
    f->state=FUTURE_WAITING;
  }

  while(f->state==FUTURE_WAITING)
  {
    printf("");
  }
  f->state=FUTURE_EMPTY;
}

else if(f->flag==FUTURE_SHARED)
{
  f->pid=getpid();
  if(f->state==FUTURE_EMPTY)
     f->state=FUTURE_WAITING;

  if(f->state==FUTURE_WAITING)
  {
    fut_enqueue(f->get_queue,f->pid);
    suspend(f->pid);
  }
}

else if(f->flag==FUTURE_QUEUE)
{
  if(fut_isempty(f->set_queue))
  {
    f->pid=getpid();
    fut_enqueue(f->get_queue,f->pid);
    suspend(f->pid);
  }
  else
  {
    temp_pid=fut_dequeue(f->set_queue);
    if(temp_pid==-1)
    {
      return -1;
    }
    resume(temp_pid);
  }
}
```

```
  *value=*(f->value);
  return OK;
}
```

future_set.c

Producer calls future_set in order to set the value present in the future variable.

Code

```
#include <future.h>

syscall future_set(future *f, int *value)
{
  intmask mask;
  pid32 temp_pid;

  mask=disable();

  if(f->flag==FUTURE_EXCLUSIVE)
  {
    if(f->state==FUTURE_EMPTY || f->state==FUTURE_WAITING)
    {
      f->state=FUTURE_VALID;
      *(f->value)=*value;
    }
    else
    {
      restore(mask);
      return SYSERR;
    }
  }
  else if(f->flag==FUTURE_SHARED)
  {
    if(f->state==FUTURE_VALID)
    {
      restore(mask);
      return SYSERR;
    }

    *(f->value)=*value;
    if(f->state==FUTURE_EMPTY)
      f->state=FUTURE_VALID;

    if(f->state==FUTURE_WAITING)
    {
      f->state=FUTURE_VALID;
```

```
      while(!fut_isempty(f->get_queue))
      {
        temp_pid=fut_dequeue(f->get_queue);
        if(temp_pid==-1)
        {
          restore(mask);
          return -1;
        }
        resume(temp_pid);
      }//while end
    }//waiting end
  }//shared end

  else if(f->flag==FUTURE_QUEUE)
  {
    if(fut_isempty(f->get_queue))
    {
      f->pid=getpid();
      fut_enqueue(f->set_queue,f->pid);
      suspend(f->pid);
      *(f->value)=*value;
    }
    else
    {
      *(f->value)=*value;
      temp_pid=fut_dequeue(f->get_queue);
      if(temp_pid==-1)
      {
        restore(mask);
        return -1;
      }
      resume(temp_pid);
    }
  }

  restore(mask);
  return OK;
}
```

fut_queue.h

Declarations and function prototypes for queue data structure.

Code

```
/* Queue structure declarations, functions*/
```

```
#ifndef _FUT_QUEUE_H_
#define _FUT_QUEUE_H_

#include <xinu.h>

typedef struct fut_qnode
{
  pid32       key;            /* Key on which the queue is ordered       */
  qid16       *next;
}fut_qnode;

typedef struct fut_queue
{
  fut_qnode *front,*rear;
}fut_queue;

/*queue manipulation functions */
fut_queue *fut_qcreate();
fut_qnode* newNode(pid32 pid);
int fut_enqueue(fut_queue *q, pid32 pid);
pid32 fut_dequeue(fut_queue *q);
int fut_isempty(fut_queue *q);

#endif
```

fut_queue.c

Implementation of working logic and enqueue,dequeue operations for queue.

Code

```
/* fut_queue.c - enqueue, dequeue */

#include <fut_queue.h>

/*queue manipulation functions */

fut_queue *fut_qcreate()
{
  fut_queue *q=(fut_queue *)getmem(sizeof(fut_queue));
  if(q==NULL)
    return NULL;
  q->front=NULL;
  q->rear=NULL;
  return q;
}

fut_qnode* newNode(pid32 pid)
{
  fut_qnode *qnode=(fut_qnode *)getmem(sizeof(fut_qnode));
  if(qnode==NULL)
    return NULL;
  qnode->next=NULL;
  qnode->key=pid;
  return qnode;
}

int fut_enqueue(fut_queue *q, pid32 pid)
{
  fut_qnode *temp=newNode(pid);
  if(temp==NULL)
    return -1;

  if(q->rear==NULL)
  {
    q->front=q->rear=temp;
    return 0;
  }

  q->rear->next=temp;
  q->rear=temp;
  return 0;
}

pid32 fut_dequeue(fut_queue *q)
{
  pid32 pid;
```

```
  fut_qnode *temp;

  if(q->front==NULL)
    return -1;

  pid=q->front->key;
  temp=q->front;
  q->front=q->front->next;

  if(!freemem(temp,sizeof(fut_qnode)))
    return -1;

  if(q->front==NULL)
    q->rear=NULL;

  return pid;
}

int fut_isempty(fut_queue *q)
{
  if(q->front==NULL)
    return 1;
  return 0;
}
```