3

List and Queue Manipulation

Linked list processing is fundamental in operating systems – it seems to pervade every component. This chapter introduces the set of procedures that form the backbone of linked list manipulation in the PC-Xinu process manager. The routines described here are used to maintain queues ordered by time of insertion and queues ordered by priority. They perform such actions as: inserting an item at the tail of a list, inserting an item in an ordered list, removing an item at the head of a list, and allocating a new list.

The linked list routines in this chapter provide a good introduction to the programming language C and to the programming conventions used throughout this book. Starting with these routines is especially helpful because they deal with a familiar subject, and because only one process executes them at a given time. Thus, the reader can think of the code as being part of a sequential program – there is no need to worry about interference from other processes executing concurrently.

3.1 Linked Lists Of Processes

The process manager deals with objects called *processes*, moving them to and from various lists frequently. The items actually stored in these lists are small, nonnegative integers called *process identifiers* (or *process ids*); we will use the terms ‘‘process,’’ ‘‘process identifier,’’ and ‘‘process id’’ interchangeably throughout this chapter. Constant *NPROC* gives the range of process identifiers. If it helps, assume *NPROC* is 30 and the items to be stored are integers between 0 and 29.

An early design called for many process lists, each with its own data structure. Some of the lists were first-in-first-out (FIFO) queues, others were ordered by key. Some were singly-linked, while others had to be doubly-linked. After the requirements

were formulated, it became obvious that centralizing the linked-list processing into a single data structure would eliminate many special cases in the code.
To accommodate all these cases, we have chosen a representation in which: all lists are doubly-linked (each node points to its predecessor as well as its successor), each node contains a key (even though key values are not used in FIFO lists), and each list has both a head and tail. In essence, they all have the form shown in Figure 3.1.

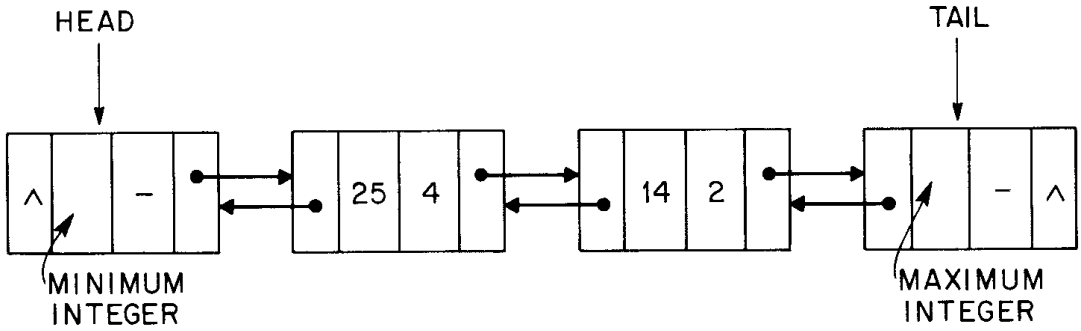

Figure 3.1 A doubly-linked list containing 4 (key=25) and 2 (key=14).

The key field in the head node contains the minimum possible integer; the key field in the tail contains the maximum possible integer. As expected, the successor of the tail and the predecessor of the head are null. When a list is empty, the successor of the head is the tail, and the predecessor of the tail is the head.
The diagram above is only a logical one. In practice, memory requirements have been reduced by storing the data field implicitly. Such optimization is only possible because of the following property:

A process appears on at most one list at any time.

To understand how data elements can be stored implicitly, look at Figure 3.2. It shows an array called the *Q* structure, each entry of which has three fields: a key field, a next field, and a previous field. Positions 0 through *NPRO*C-1 correspond to the integer process ids that are stored in the list; positions *NPROC* and higher are used for heads and tails of lists. To place item *i* on a list, the node with index *i* is linked into the list. It is only possible to reserve a node for each item because the range of values is small (typically, *NPRO*C=30), and no item ever appears on more than one list simultaneously. A closer look at the code should make the operations clear.

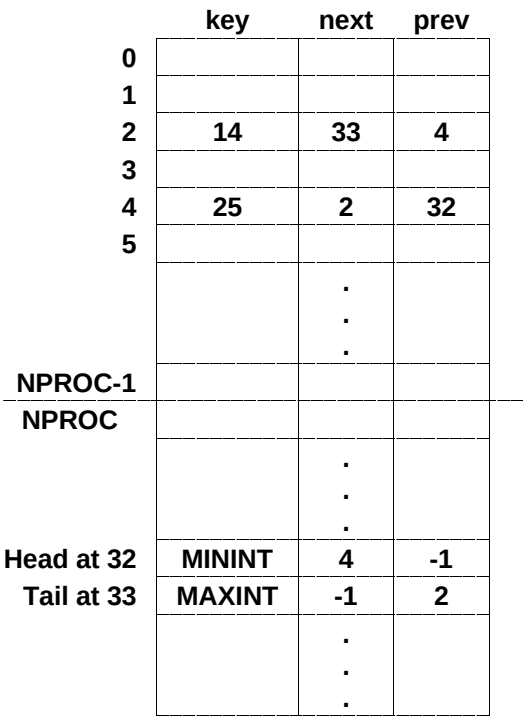

Figure 3.2 The list from Figure 3.1 stored in the q array.

3.2 Implementation Of The Q Structure

In C, the *Q* structure pictured above is *q*, an array of *qent* structures. File *q.h* contains the declarations of both *q* and *qent*:

```
/* q.h - firstid, firstkey, isempty, lastkey, nonempty */

/* q structure declarations, constants, and inline procedures           */

#define NQENT           NPROC + NSEM + NSEM + 4 /* for ready & sleep    */

struct qent {                   /* one for each process plus two for    */
                                /* each list                            */
        int   qkey;             /* key on which the queue is ordered    */
        int   qnext;            /* pointer to next process or tail      */
        int   qprev;            /* pointer to previous process or head  */
};

extern  struct  qent q[];
extern  int     nextqueue;

/* inline list manipulation procedures */

#define isempty(list)   (q[(list)].qnext >= NPROC)
#define nonempty(list)  (q[(list)].qnext < NPROC)
#define firstkey(list)  (q[q[(list)].qnext].qkey)
#define lastkey(tail)   (q[q[(tail)].qprev].qkey)
#define firstid(list)   (q[(list)].qnext)

#define EMPTY   -1              /* equivalent of null pointer           */
```

Each *q* entry either corresponds to the head of a list, the tail of a list, or an item to be placed on a list. For now, remember that items stored on lists are process id integers in the range 0 to *NPROC*-1. The implicit assumption throughout the code is that *q*[0] through *q*[NPROC-1] correspond to these process ids, while *q*[NPROC] through *q*[NQENT-1] correspond to the heads or tails of lists.

Symbolic constant *NQENT* defines the number of entries in the *q* array; the value ‘‘NPROC+NSEM+NSEM+4’’ will allocate enough room in the *q* structure for *NPROC* processes as well as head and tail pointers for *NSEM* semaphore queues, a ready list, and a sleep list.

The contents of entries in the *q* array are defined by structure *qent*. (This file contains only a declaration of the shape of elements in the *q* array; we will see the definition of its contents in Chapter 13.) Field *qnext* points forward, *qprev* points backward, and *qkey* contains an integer key for the node. When the forward and backward pointers do not contain valid indexes, they are assigned *EMPTY*.

3.2.1 In-Line Q Functions

The functions *isempty* and *nonempty* are predicates (Boolean functions) that test whether a list is empty or not, given the index of its head as an argument. *Isempty* determines whether a list is empty by checking to see if the first node on the list is a process or the list tail; *nonempty* makes the opposite test. Remember that an item is a process if and only if its index is less than *NPROC*.

The other in-line functions should also be easy to understand. Functions *firstkey*, *lastkey*, and *firstid* return the key of the first process on a list, the key of the last process on a list, or the *q* index of the first process on a list. Usually, these functions are applied to nonempty lists, but they do not abort even if the list is empty because *qkey* is always initialized.

3.2.2 FIFO Queue Manipulation

To produce a FIFO queue, items are inserted at the tail of a list and removed at the head of a list. Procedures *enqueue* and *dequeue*, found in file *queue.c*, perform the FIFO operations in PC-Xinu. The code is straight-forward, once you understand how pointers operate. Variables *tptr* and *mptr* are pointers to *qent* structures. The first two executable statements in *enqueue* assign these pointers the addresses of the *q* entries corresponding to the tail of the list and the process to be inserted. Once the address of an element has been recorded, individual fields of the structure are referenced using the ‘‘–>’’ operator. The point of recording addresses in pointers is efficiency – it avoids recomputing array subscripts again and again.

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

#include <conf.h>
#include <kernel.h>
#include <q.h>

/*------------------------------------------------------------------------
 *  enqueue  --  insert an item at the tail of a list
 *------------------------------------------------------------------------
 */
int enqueue(item, tail)
int item;                               /* item to enqueue on a list    */
int tail;                               /* index in q of list tail      */
{
        struct  qent    *tptr;          /* points to tail entry         */
        struct  qent    *mptr;          /* points to item entry         */

        tptr = &q[tail];
        mptr = &q[item];
        mptr->qnext = tail;
        mptr->qprev = tptr->qprev;
        q[tptr->qprev].qnext = item;
        tptr->qprev = item;
        return(item);
}


/*------------------------------------------------------------------------
 *  dequeue  --  remove an item from a list and return it
 *------------------------------------------------------------------------
 */
int dequeue(item)
int item;
{
        struct  qent    *mptr;          /* pointer to q entry for item  */

        mptr = &q[item];
        q[mptr->qprev].qnext = mptr->qnext;
        q[mptr->qnext].qprev = mptr->qprev;
        return(item);
}
```

File *queue.c* includes three other files: *conf.h*, *kernel.h*, and *q.h*. File *q.h* is needed because procedures *enqueue* and *dequeue* both reference the *q* structure. But why are the

other two included? It turns out that neither are needed by the code contained explicitly in *queue.c*. However, *queue.c* includes file *q.h* which references constants like *NPROC* and *NSEM* that are defined in *kernel.h* and *conf.h*. As a general rule, so many important constants have been collected into these two files that most system routines must include them. We will simply include them for now and postpone looking at them until later.

3.3 Priority Queue Manipulation

The process manager often needs to select from a set of processes one with the highest priority, so the linked list routines must be able to maintain sets of processes that have an associated *priority*. (Priorities in PC-Xinu are easy to understand: for now, think of them as integer values assigned to the processes.) In general, the task of selecting a process with highest priority is performed frequently compared with the tasks of inserting processes in the set and deleting them, so the idea is to design a data structure that makes selection efficient compared to insertion.

A variety of data structures have been devised to store sets when selection by priority is important. Such data structures are called *priority queues*. Although not all ''priority queues'' use a queue, the term accurately describes the PC-Xinu implementation – Xinu priority queues are merely linked lists in which processes are ordered by their priority. The highest priority process can always be found at the tail of the list. Of course, insertion in a priority queue is more expensive than insertion in a FIFO queue because the list must be searched to determine where the new item should be located.

When many items appear in a priority queue, or if the number of insertions is high compared to the number of times items are extracted by priority, using linear lists for priority queues would not be efficient (the exercises discuss this point further). However, in a small system like PC-Xinu, where we expect 2 or 3 elements to be on a given priority queue at any time, simple lists suffice.

The procedures that maintain ordered lists are straightforward. *Insert*, shown below, takes as an argument a process id, an integer giving the head of a list in the *q* structure, and a priority, and inserts the process into its correct position in the list. It uses the *qkey* field of a process' node to store that process' priority. To find the correct location in the list, *insert* searches for an existing element with a key greater than or equal to the key of the element being inserted. During the search, integer *next* moves along the list. The loop must eventually terminate because the key of the tail element contains the largest possible integer. Once the correct location has been found, *insert* changes the necessary pointers to link the new node into the list.

```
/* insert.c - insert */

#include <conf.h>
#include <kernel.h>
#include <q.h>

/*------------------------------------------------------------------------
 *  insert  --  insert a process into a q list in key order
 *------------------------------------------------------------------------
 */
int insert(proc, head, key)
int proc;                               /* process to insert            */
int head;                               /* q index of head of list      */
int key;                                /* key to use for this process  */
{
        int     next;                   /* runs through list            */
        int     prev;

        next = q[head].qnext;
        while (q[next].qkey < key)      /* tail has MAXINT as key       */
                next = q[next].qnext;
        q[proc].qnext = next;
        q[proc].qprev = prev = q[next].qprev;
        q[proc].qkey  = key;
        q[prev].qnext = proc;
        q[next].qprev = proc;
        return(OK);
}
```

Elements can be extracted from a FIFO queue by removing them from the head; they can be extracted from a priority queue at either the head or tail. Procedures *getfirst* and *getlast* provide the operations of removing items from queues. *Getfirst* takes the list head index as an argument, and *getlast* takes the list tail index as an argument. If the list is a priority queue, *getfirst* removes an item with the smallest key and *getlast* removes an item with the largest key. For FIFO queues, *getfirst* removes the oldest item in the list. Both routines return the index of the item removed. These returned values are either process ids or *EMPTY*.

```
/* getitem.c - getfirst, getlast */

#include <conf.h>
#include <kernel.h>
#include <q.h>
```

```
/*------------------------------------------------------------------------
 *  getfirst  --  remove and return the first process on a list
 *------------------------------------------------------------------------
 */
int     getfirst(head)
        int     head;                   /* q index of head of list      */
{
        int     proc;                   /* first process on the list    */

        if ((proc=q[head].qnext) < NPROC)
                return( dequeue(proc) );
        else
                return(EMPTY);
}



/*------------------------------------------------------------------------
 *  getlast  --  remove and return the last process from a list
 *------------------------------------------------------------------------
 */
int     getlast(tail)
        int     tail;                   /* q index of tail of list      */
{
        int     proc;                   /* last process on the list     */

        if ((proc=q[tail].qprev) < NPROC)
                return( dequeue(proc) );
        else
                return(EMPTY);
}
```

3.4 List Initialization

The procedures described so far all assume that even though the lists may be empty, their head and tail nodes have been initialized. We now consider how to create empty lists in the first place. It is appropriate that this material occurs at the end of this chapter because it brings up an important point about the design process:

Initialization is the final step in design.

This may sound strange because it is not possible to postpone thinking about initialization

altogether, but the point is simple: design the data structures needed to keep the system running first and then figure out how to initialize them. Partitioning the ‘‘steady state’’ part of the system from the ‘‘transient state’’ part helps avoid the temptation of sacrificing good design for easy initialization.

Initialization of entries in the *q* structure is performed on demand as entries are needed. Running programs call *newqueue* to create a new list. *Newqueue* allocates a pair of adjacent positions in the *q* array to use as head and tail nodes; it initializes the list to empty by pointing the successor of the head to the tail and the predecessor of the tail to the head. Other pointers are assigned the value *EMPTY*. When it initializes the head and tail, *newqueue* also sets the key fields to the smallest and largest possible integers, respectively, so the head and tail can be used with an ordered list. Finally, *newqueue* returns the index of the list head to its caller.

```
/* newqueue.c - newqueue */

#include <conf.h>
#include <kernel.h>
#include <q.h>

/*------------------------------------------------------------------------
 *  newqueue  --  initialize a new list in the q structure
 *------------------------------------------------------------------------
 */
int     newqueue()
{
        struct  qent    *hptr;          /* address of new list head     */
        struct  qent    *tptr;          /* address of new list tail     */
        int     hindex, tindex;         /* head and tail indexes        */

        hptr = &q[ hindex=nextqueue++ ];/* nextqueue is global variable */
        tptr = &q[ tindex=nextqueue++ ];/*  giving next used q pos.     */
        hptr->qnext = tindex;
        hptr->qprev = EMPTY;
        hptr->qkey  = MININT;
        tptr->qnext = EMPTY;
        tptr->qprev = hindex;
        tptr->qkey  = MAXINT;
        return(hindex);
}
```

3.5 Summary

The code in this chapter described linked-list manipulation in the process manager. Linked lists of processes are kept in a single data structure, the *q* array. Primitive operations for manipulating the lists of processes can produce FIFO queues or priority queues. All lists have the same format: they are doubly-linked, each has both a head and tail, and each node has an integer key field. Keys are used when the list is a priority queue; they are ignored if the list is a FIFO queue.

FOR FURTHER STUDY

Knuth [1968] describes linked-list manipulation in detail. Good algorithms for priority queues and related data structures can be found in Aho et. al. [1974]. Wirth [1976] contains examples in Pascal. Habermann [1976] explains the use of a priority queue in operating systems.

EXERCISES

3.1 Write test drivers to exercise the procedures introduced in this chapter. You will need to initialize the queue structure by setting *nextqueue* to NPROC and making NSEM+2 calls to *newqueue*.

3.2 Duplicate the procedures in this chapter to work with singly-linked lists. How much storage does the second set of routines require? How much CPU time do they save?

3.3 Implement procedures to manipulate lists using pointers instead of subscripts into an array of structures. How much storage/time do they save? Comment on the complexity of routines like *isempty* when implemented with pointers.

3.4 Does *insert* work correctly for all possible keys? If not, for which does it fail?

3.5 Larger systems sometimes use a data structure known as a *heap* to contain a priority queue. What is a heap? Will its use be more or less expensive than an ordered, doubly-linked list when the list size is between 1 and 3?

3.6 Finding the address of an item in an array may require multiplication. If the compiler you are using converts multiplication by a power of 2 into a shift, try padding the size of a *qent* to a power of 2 bytes. How much faster does it make the routines? Investigate the relative speed of multiplication and shifting on the 8088.

3.7 Modify *insert* to use pointers instead of subscripting. Is it faster? Larger?

3.8 Rewrite all the list manipulation routines so they reference a list by a single integer, *k*, and assume that *q[k]* is the list head and *q[k+1]* is the list tail. Are they faster or slower?

3.9 Modify *newqueue* to check for an error caused by allocating more than *NQENT* entries.