Operating Systems

Xinu – Internals

List and Queue Manipulation

1

Linked Lists in the OS

- Manipulating lists of processes is an important operation in the OS
  - A process's lifecycle consists of moving between, and in, queues and lists
- Xinu implements a unified approach to list management
  - All list management uses this common infrastructure
  - Common functions to create a new list, insert an element at the end of the list, insert or remove from the middle, remove an item from the front
- We can think of this code as being sequential
  - Protected with mutual exclusion as necessary

2

Unified Lists in Xinu

- The process manager handles processes
  - A process moves among the lists frequently
  - At any time, a process is only in one list
- Rather than store all the information about a process, the process manager is free to store only the process ID (PID) or Thread ID (TID) in a list
  - So when we refer to to putting a process on a list, it really means putting the PID there – the process control block need not move
- Unified implementation means that not every subsystem uses all the list features

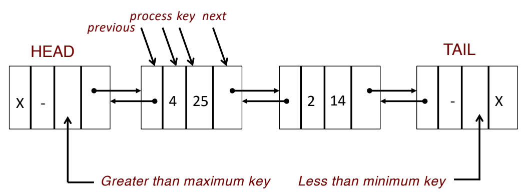
3

List Properties

- All lists are doubly linked – each node points to predecessor and successor
- Each node stores a key as well as a process ID, even though a key is not used in a FIFO list
- Each list has head and tail nodes; the head and tail nodes have the same memory layout as other nodes
- Non-FIFO lists are ordered in descending order; the key in the head node is greater than the maximum valid key value, and the key value in the tail node is less than the minimum valid key

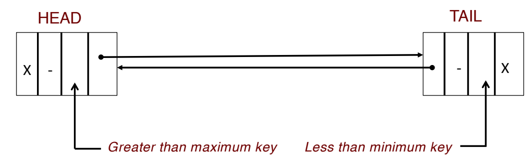
4

A doubly-linked List



5

An Empty List



6

A compact list structure

- Compact memory use is important
  - in general, but in embedded systems in particular
- Two ideas related to process specific representation
- Relative pointers
  - Given that there is some (small) fixed number of processes (NPROC)
  - One might use a pointer in this situation, which is 4 bytes
  - For NPROCS < 62, we only need 6 bits
  - Allocate the nodes in a contiguous array and use the array index as a “pointer”

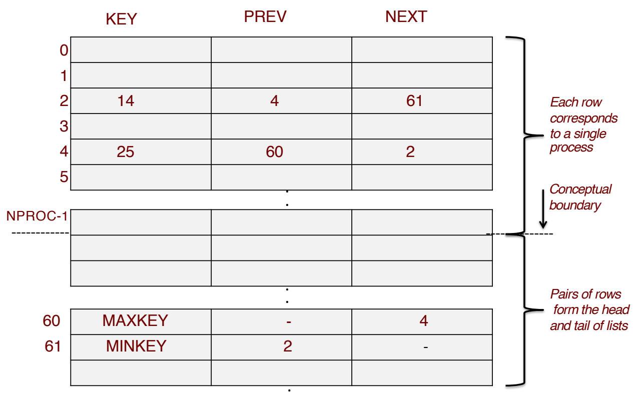
7

A compact list structure

- Implicit data structure – based on the fact that a process can only be in one list, we can use the list position to indicate the ID
- To omit the PID, use an array and use the *ith* element of the array for process *i*
- To put process 3 in a linked list, insert node 3 into the list
- The relative address of a node is the same as the ID of a process being stored
  - List membership is implicit

8

Illustration of implicit identifiers



Notes

- The table contains NQENT entries
- There is an implicit divider at NPROCS-1
- Between queuetab[NPROC] and queuetab[NQENT-1] are the heads and tails of lists
- The default NPROC + 4 + NSEM + NSEM allocates enough space for
  – each process
  – head and tail entries for the ready and sleep lists
  – a head and tail entry for each of the NSEM semaphores in the system
- This can be changed at compile time

Implementation of the Queue Data Structure

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

/* Queue structure declarations, constants, and inline functions   */

/* Default # of queue entries: 1 per process plus 2 for ready list plus*/
/*     2 for sleep list plus 2 per semaphore    */

#ifndef NQENT
#define NQENT  (NPROC + 4 + NSEM + NSEM)
#endif

#define EMPTY   (-1)        /* null value for qnext or qprev index    */
#define MAXKEY  0x7FFFFFFF /* max key that can be stored in queue    */
#define MINKEY  0x80000000 /* min key that can be stored in queue    */
```



```
struct  qentry {       /* one per process plus two per list */
   int32    qkey;      /* key on which the queue is ordered */
   qid16    qnext;     /* index of next process or tail  */
   qid16    qprev;     /* index of previous process or head */
};

extern  struct qentry  queuetab[];

/* Inline queue manipulation functions */

#define queuehead(q)    (q)
#define queuetail(q)    ((q) + 1)
#define firstid(q)  (queuetab[queuehead(q)].qnext)
#define lastid(q)   (queuetab[queuetail(q)].qprev)
#define isempty(q)  (firstid(q) >= NPROC)
#define nonempty(q) (firstid(q) <  NPROC)
#define firstkey(q) (queuetab[firstid(q)].qkey)
#define lastkey(q)  (queuetab[ lastid(q)].qkey)
```

```
/* assumes interrupts are disabled */

/* Inline to check queue id */

#define isbadqid(x) (((int32)(x) < 0) || (int32)(x) >= NQENT-1)

/* Queue function prototypes */

pid32   getfirst(qid16);
pid32   getlast(qid16);
pid32   getitem(pid32);
pid32   enqueue(pid32, qid16);
pid32   dequeue(qid16);
status  insert(pid32, qid16, int);
status  insertd(pid32, qid16, int);
qid16   newqueue(void);
```

13

Basic Functions to Extract A Process From A List

```
/* getitem.c - getfirst, getlast, getitem */
#include <xinu.h>
/*------------------------------------------------------------------------ *
  getfirst  -  Remove a process from the front of a queue
 *------------------------------------------------------------------------ */
pid32   getfirst(
     qid16      q    /* ID of queue from which to  */
   )                 /* remove a process (assumed  */
                     /* valid with no check)       */
{
   pid32    head;

   if (isempty(q)) {
    return EMPTY;
   }

   head = queuehead(q);
   return getitem(queuetab[head].qnext);
}
```

14

```
/*------------------------------------------------------------------------
 *  getlast  -  Remove a process from end of queue
 *------------------------------------------------------------------------
 */
pid32   getlast(
     qid16      q    /* ID of queue from which to  */
   )                 /* remove a process (assumed  */
                     /* valid with no check)       */
{
   pid32 tail;

   if (isempty(q)) {
    return EMPTY;
   }

   tail = queuetail(q);
   return getitem(queuetab[tail].qprev);
}
```

15

```
/*------------------------------------------------------------------------
 *  getitem  -  Remove a process from an arbitrary point in a queue
 *------------------------------------------------------------------------
 */
pid32   getitem(
     pid32      pid     /* ID of process to remove*/
   )
{
   pid32    prev, next;

   next = queuetab[pid].qnext;/* following node in list */
   prev = queuetab[pid].qprev;/* previous node in list  */
   queuetab[prev].qnext = next;
   queuetab[next].qprev = prev;
   return pid;
}
```

16

FIFO Queue Manipulation

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

#include <xinu.h>

struct qentry  queuetab[NQENT];   /* table of process queues*/

/*------------------------------------------------------------------
 *  enqueue  -  Insert a process at the tail of a queue
 *------------------------------------------------------------------
 */
pid32   enqueue(
     pid32     pid,    /* ID of process to insert*/
     qid16     q       /* ID of queue to use     */
   )
```

17

```
{
   int  tail, prev;    /* tail & previous node indexes   */

   if (isbadqid(q) || isbadpid(pid)) {
    return SYSERR;
   }

   tail = queuetail(q);
   prev = queuetab[tail].qprev;

   queuetab[pid].qnext  = tail;   /* insert just before tail node   */
   queuetab[pid].qprev  = prev;
   queuetab[prev].qnext = pid;
   queuetab[tail].qprev = pid;
   return pid;
}
```

18

```
/*------------------------------------------------------------------
 *  dequeue  -  Remove and return the first process on a list
 *------------------------------------------------------------------
 */
pid32   dequeue(
     qid16     q       /* ID queue to use     */
   )
{
   pid32    pid;           /* ID of process removed   */

   if (isbadqid(q)) {
    return SYSERR;
   } else if (isempty(q)) {
    return EMPTY;
   }

   pid = getfirst(q);
   queuetab[pid].qprev = EMPTY;
   queuetab[pid].qnext = EMPTY;
   return pid;
}
```

19

Manipulation of Priority Queues

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

#include <xinu.h>

/*------------------------------------------------------------------
 *  insert  -  Insert a process into a queue in descending key order
 *------------------------------------------------------------------
 */
status  insert(
     pid32     pid,    /* ID of process to insert*/
     qid16     q,      /* ID of queue to use     */
     int32     key     /* key for the inserted process   */
   )
```

20

```
{
   int16    curr;          /* runs through items in a queue*/
   int16    prev;          /* holds previous node index  */

   if (isbadqid(q) || isbadpid(pid)) {
    return SYSERR;
   }

   curr = firstid(q);
   while (queuetab[curr].qkey >= key) {
       curr = queuetab[curr].qnext;
   }
/* insert process between curr node and previous node */

   prev = queuetab[curr].qprev;   /* get index of previous node */
   queuetab[pid].qnext = curr;
   queuetab[pid].qprev = prev;
   queuetab[pid].qkey = key;
   queuetab[prev].qnext = pid;
   queuetab[curr].qprev = pid;
   return OK;
}
```

21

List Initialization

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

#include <xinu.h>

/*------------------------------------------------------------------------
 *  newqueue  -  Allocate and initialize a queue in the global queue
 *               table
 *------------------------------------------------------------------------
 */
qid16   newqueue(void)
{
   static qid16    nextqid=NPROC;/* next list in queuetab to use    */
   qid16       q;      /* ID of allocated queue   */

   q = nextqid;
   if (q > NQENT) {        /* check for table overflow   */
    return SYSERR;
   }
```

22

```
nextqid += 2;         /* increment index for next call*/

   /* initialize head and tail nodes to form an empty queue */

   queuetab[queuehead(q)].qnext = queuetail(q);
   queuetab[queuehead(q)].qprev = EMPTY;
   queuetab[queuehead(q)].qkey  = MAXKEY;
   queuetab[queuetail(q)].qnext = EMPTY;
   queuetab[queuetail(q)].qprev = queuehead(q);
   queuetab[queuetail(q)].qkey  = MINKEY;
   return q;
}
```

23