Module IV

Lists Of Processes And List Functions

Queues And Lists

- Keeping track of processes is fundamental throughout an operating system
- Various forms are needed
  - FIFO queues of processes waiting for I/O devices
  - Lists of processes kept in priority order for scheduling
  - Lists of events ordered by the time they will occur
- Operations required
  - Insert a process onto a list
  - Extract the “next” process from a list
  - Delete an arbitrary process (e.g., if the process is killed)

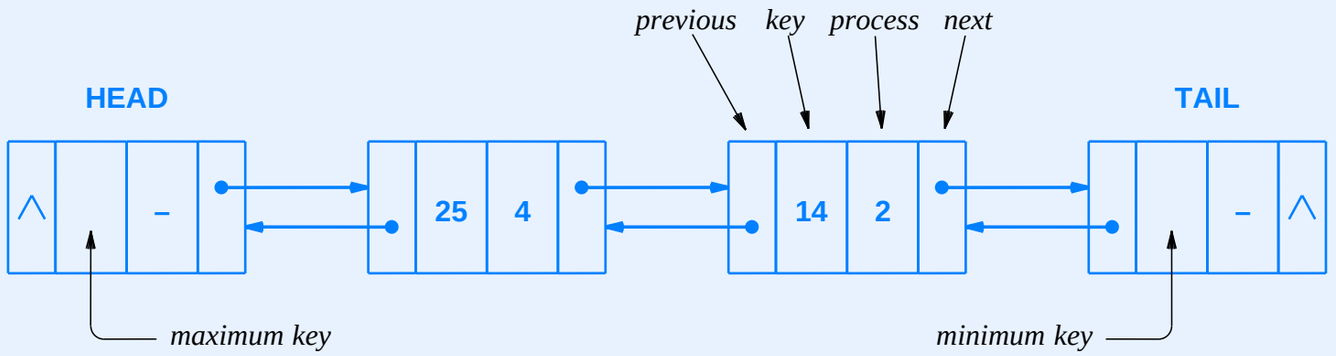
Lists And Queues In Xinu

- Important ideas
  - A process is known by an integer *process ID*
  - A list of processes really consists of a list of process IDs
- A single data structure can be used to store many types of process lists

Unified List Storage in Xinu

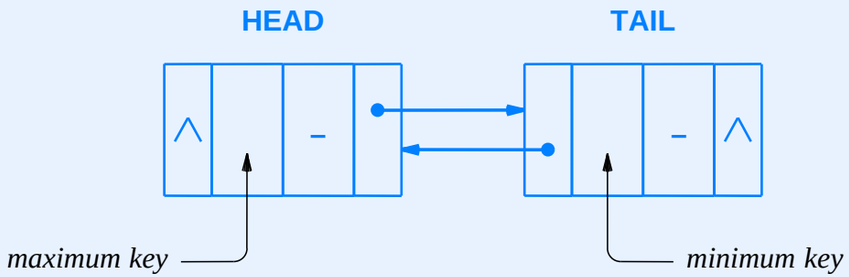
- All lists are doubly-linked, which means a node points to its predecessor and successor
- Each node stores a *key* as well as a process ID, even though the key is not used in a FIFO list
- Each list has a *head* and *tail*; the head and tail nodes have the same shape as other nodes
- Non-FIFO lists are always ordered in descending order according to the key values
- The key value in a head node is the maximum integer used as a key, and the key value in the tail node is the minimum integer used as a key

Conceptual List Structure



- The example list contains two processes, 2 and 4
- Process 4 has key 25
- Process 2 has key 14

Pointers In An Empty List



- In an empty list, the head and tail nodes are linked
- Having a head and tail eliminates special cases for insertion and deletion

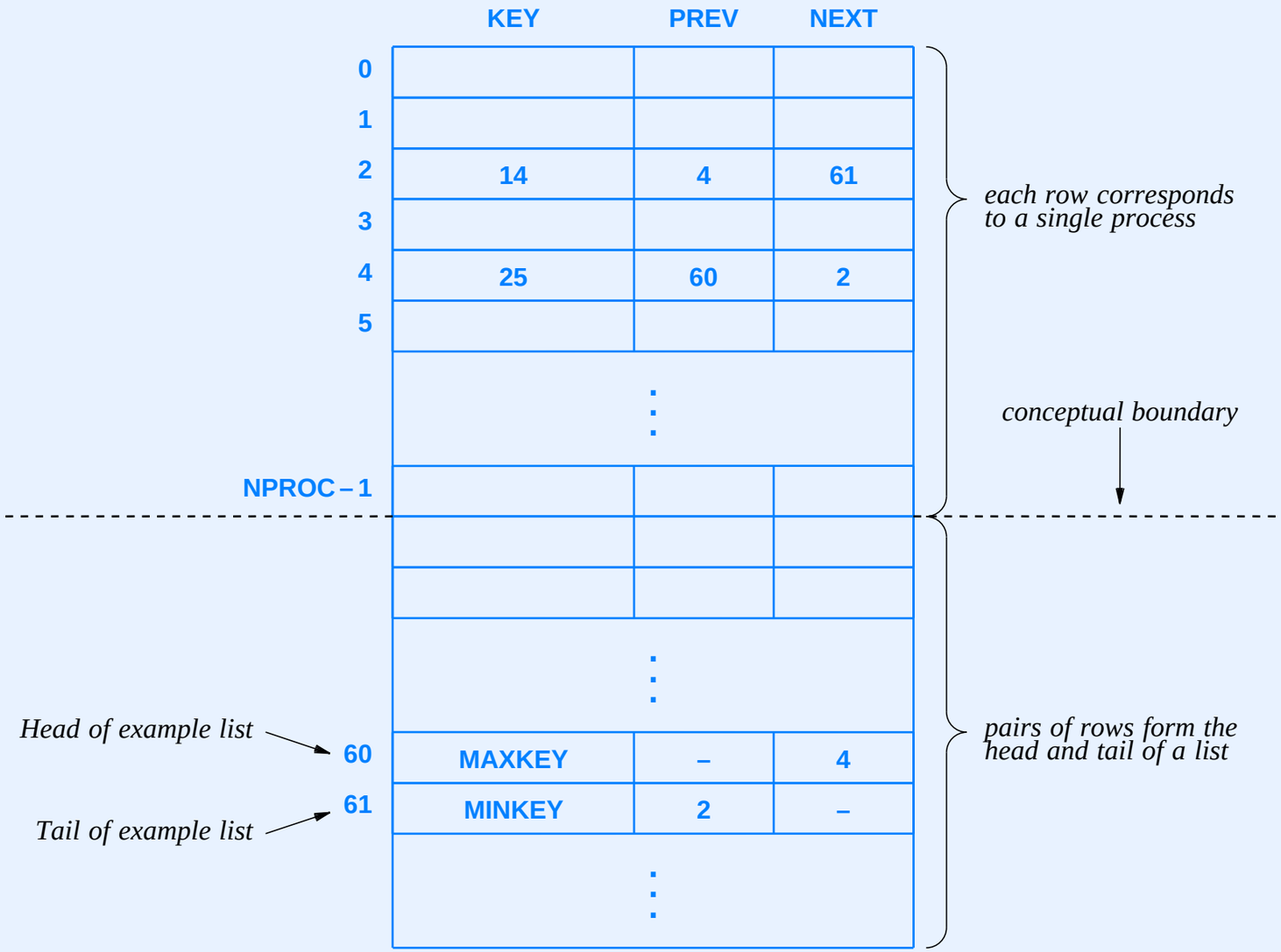
Reducing The List Size

- Pointers can mean a large memory footprint, especially on a 64-bit computer
- Important concept: a process can appear on at most one list at any time
- Xinu uses two clever techniques to reduce the size of lists
  - Relative pointers instead of memory addresses
  - An implicit data structure

Xinu List Optimizations

- Lists are stored in an array
  - Each item in the array stores one node of the list
  - Relative pointers: the array index is used to identify a node instead of an address
- Implicit data structure
  - Let *NPROC* be the number of processes in the system
  - Assign process IDs 0 through *NPROC – 1*
  - Let i^{th} element of the array correspond to process *i*, for $0 \leq i < NPROC$
  - Store heads and tails in same array at positions *NPROC* and higher

An Illustration Of An Array Holding The Xinu List Structure



Implementation

- A single array is used to hold all lists of processes
  - The array is global and available throughout the entire OS
  - The array is named *queuetab*
- Functions are available to manipulate lists
  - Include tests, such as *isempty*, as well as insertion and deletion operations
  - For efficiency, functions are implemented with inline macros when possible
- Example code shown after a discussion of types

Definitions From queue.h (Part 1)

```
/* 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[];
```

Definitions From queue.h (Part 2)

```
/* 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)

/* Inline to check queue id assumes interrupts are disabled */

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

Code For Insertion And Deletion From A FIFO Queue (Part 1)

```
/* 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           */
        )
{
        qid16   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;
}
```

Code For Insertion And Deletion From A FIFO Queue (Part 2)

```
/*------------------------------------------------------------------------
 *  dequeue  -  Remove and return the first process on a list
 *------------------------------------------------------------------------
 */
pid32   dequeue(
          qid16         q               /* ID of 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;
}
```

Code For Insertion In An Ordered List (Part 1)

```
/* 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 */
        )
{
        qid16   curr;                   /* Runs through items in a queue*/
        qid16   prev;                   /* Holds previous node index    */

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

        curr = firstid(q);
        while (queuetab[curr].qkey >= key) {
                curr = queuetab[curr].qnext;
        }
```

Code For Insertion In An Ordered List (Part 2)

```
        /* 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;
}
```

Accessing An Item In A List (Part 1)

```
/* 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);
}
```

Accessing An Item In A List (Part 2)

```
/*------------------------------------------------------------------------
 *  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);
}
```

Accessing An Item In A List (Part 3)

```
/*------------------------------------------------------------------------
 *  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;
}
```

Allocating A New List

```
/* excerpt from newqueue.c */

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;
        }

        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;
}
```

Summary

- An operating system supplies a set of services
- System calls provide interface between OS and application
- Concurrency is fundamental concept
  - Between I/O devices and processor
  - Between multiple computations
- A process is OS abstraction for concurrency; it does not appear in the code
- A process differs from program or function
- You will learn how to design and implement system software that supports concurrent processing

Summary
(continued)

- An OS has well-understood internal components
- Complexity arises from interactions among components
- A multilevel approach helps organize system structure
- OS design involves inventing policies and mechanisms that enforce overall goals
- Xinu includes a compact list structure that uses relative pointers and an implicit data structure to reduce size
- Xinu type names specify both purpose and data size

Questions?