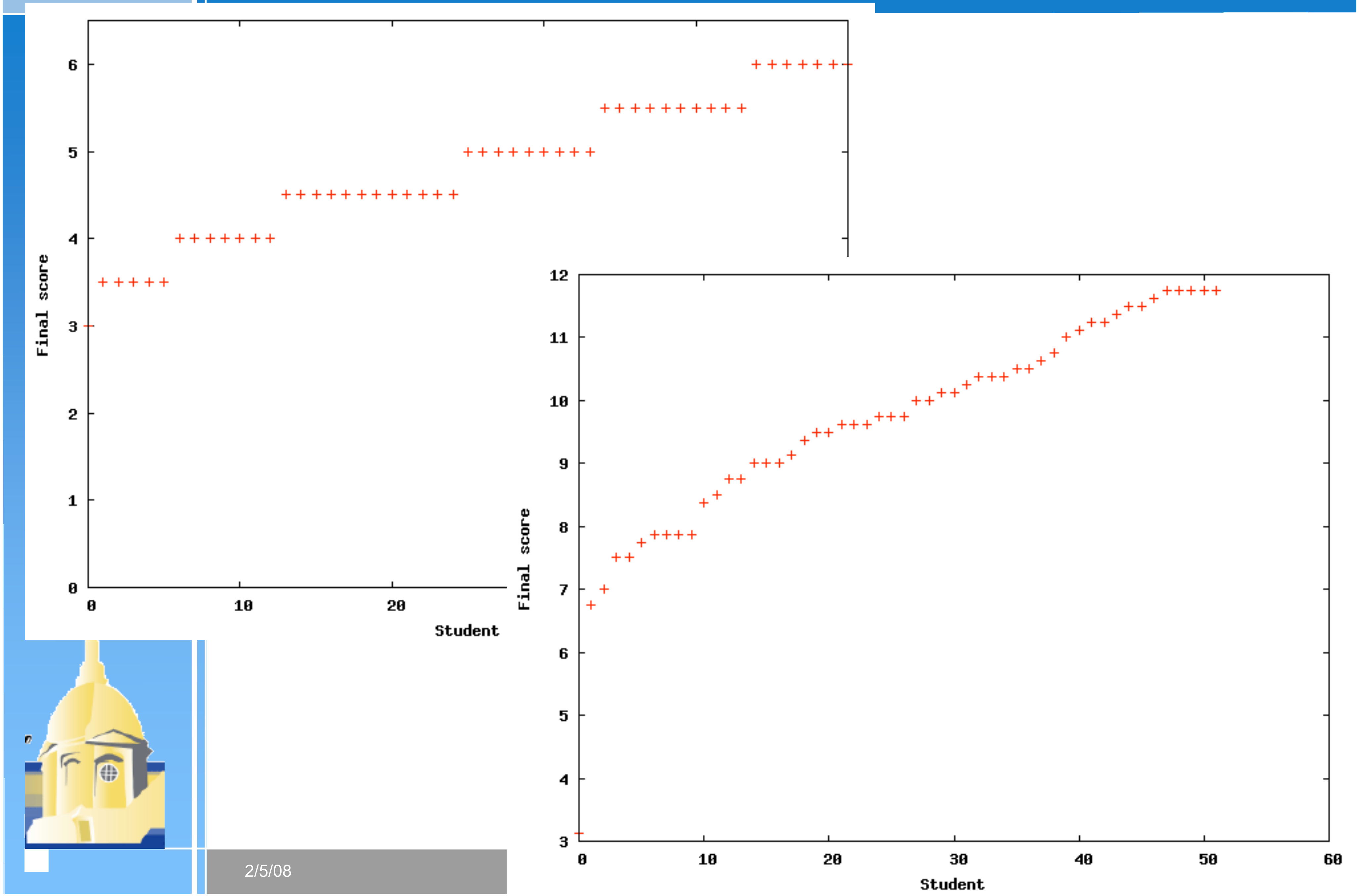
Grades



6
5
4
3
2
1
0
Final score
0
10
20
Student
12
11
10
9
8
7
6
5
4
3
Final score
0
10
20
30
40
50
60
Student

So far…

- Scheduling algorithms: FCFS, SJF, Priority, RR …
- What about: LFJ, FCLS, random?

Operating System Examples

- Windows XP scheduling
- Linux scheduling

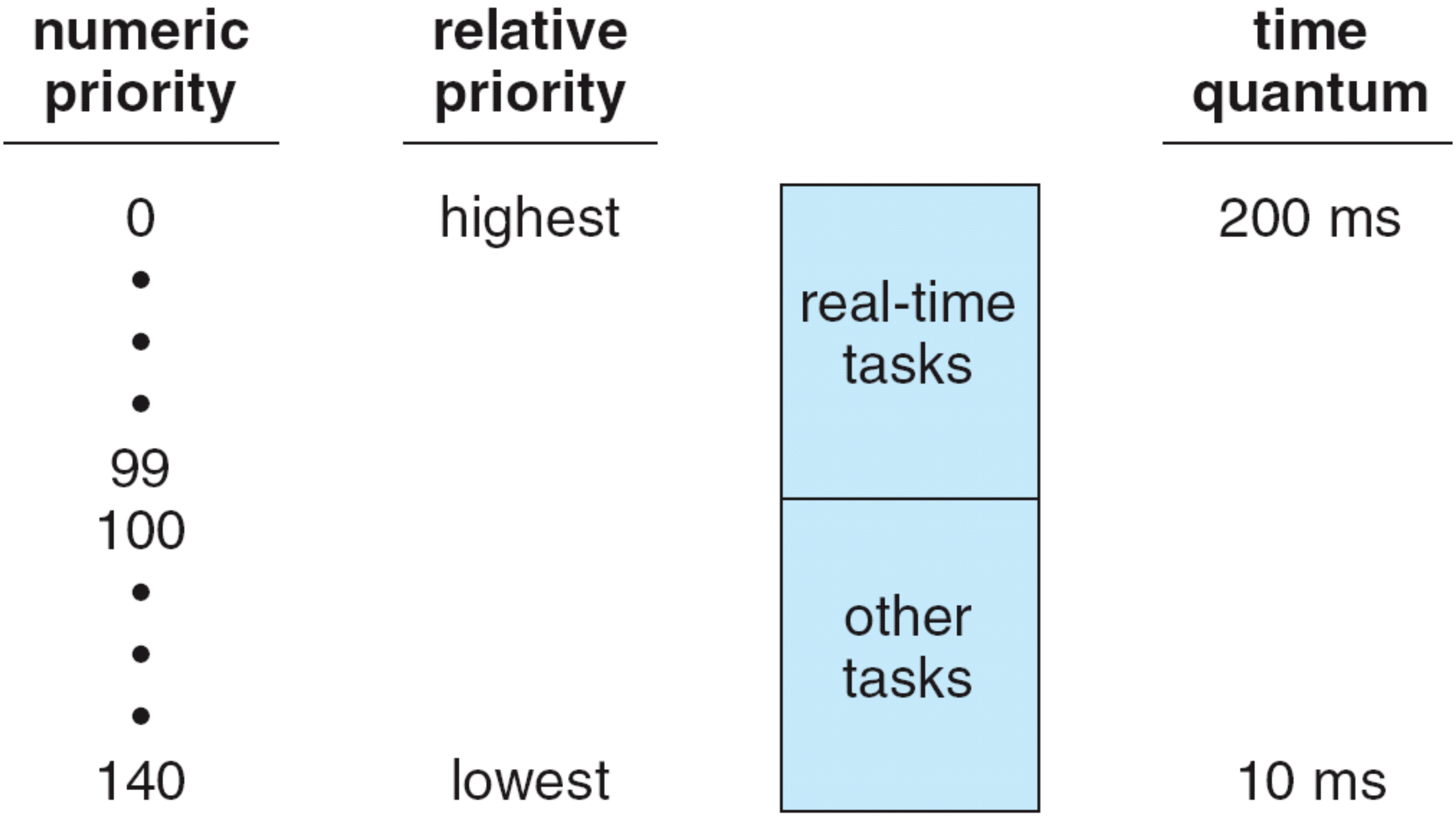
Windows XP Priorities

	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

Linux Scheduling

- Two algorithms: time-sharing and real-time
- Time-sharing
  - Prioritized credit-based – process with most credits is scheduled next
  - Credit subtracted when timer interrupt occurs
  - When credit = 0, another process chosen
  - When all processes have credit = 0, recrediting occurs
    - Based on factors including priority and history
- Real-time
  - Soft real-time
  - Posix.1b compliant – two classes
    - FCFS and RR
    - Highest priority process always runs first

The Relationship Between Priorities and Time-slice length



Module 2: Process Synchronization

- Concurrent access to shared data may result in data inconsistency
  - Multiple threads in a single process
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

Background

- Suppose that we wanted to provide a solution to the consumer-producer problem that fills all the buffers. We can do so by having an integer *count* that keeps track of the number of full buffers. Initially, *count* is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Producer/Consumer

```
Producer: while (true) {
        /* produce an item and put in nextProduced */
                    while (count == BUFFER_SIZE)
                              ; // do nothing
                    buffer [in] = nextProduced;
                    in = (in + 1) % BUFFER_SIZE;
                    count++;
        }
Consumer:while (1) {
                    while (count == 0)
                              ; // do nothing
                    nextConsumed =  buffer[out];
                    out = (out + 1) % BUFFER_SIZE;
                    count--;
                    /*  consume the item in nextConsumed */
        }
```

Race Condition

- *count*++ could be implemented as

```
register1 = count
register1 = register1 + 1
count = register1
```

- *count*-- could be implemented as

```
register2 = count
register2 = register2 - 1
count = register2
```

- Consider this execution interleaving with “count = 5” initially:

  T0: producer execute register1 = count {register1 = 5}
  T1: producer execute register1 = register1 + 1 {register1 = 6}
  T2: consumer execute register2 = count {register2 = 5}
  T3: consumer execute register2 = register2 - 1 {register2 = 4}
  T4: producer execute count = register1 {count = 6 }
  T5: consumer execute count = register2 {count = 4}

After concurrent execution, count can be 4, 5 or 6

Critical section

- Segment of code where threads are updating common variables is called a critical section
- Solution is to force only one thread inside the critical section at any one time
- Define a section before critical section, called *entry section* and a section at the end called *end section*. We can implement mechanisms in the entry section that ensures that only one thread is inside the critical section. End section can then tell someone in entry section to continue.

Solution to Critical-Section Problem

- Solution must satisfy three requirements:
  1. Mutual Exclusion - If process Pi is executing in its critical section, then no other processes can be executing in their critical sections
  2. Progress - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then only those processes that are not executing in their remainder section can participate in the decision on which will enter its critical section next and this selection cannot be postponed indefinitely
  3. Bounded Waiting - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
  - Assume that each process executes at a nonzero speed
  - No assumption concerning relative speed of the N processes

Classic s/w soln: Peterson’s Solution

- Restricted to two processes
- Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted (not true for modern processors)
- The two threads share two variables:
  - int turn;
  - Boolean flag[2]
- The variable turn indicates whose turn it is to enter the critical section.
- The flag array is used to indicate if a process is ready to enter the critical section. flag[i] = true implies that process P_i is ready!

Algorithm for Process P_i

```
do {
    flag[i] = TRUE;
    turn = j;
    while ( flag[j] && turn == j);
        CRITICAL SECTION
    flag[i] = FALSE;
         REMAINDER SECTION
} while (TRUE);
```

1) Mutual exclusion because only way thread enter critical section when flag[j] == FALSE or turn == TRUE
2) Only way to enter section is by flipping flag[] inside loop
3) turn = j allows the other thread to make progress

Synchronization Hardware

- Many systems provide hardware support for critical section code
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Have to wait for disable to propagate to all processors
    - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
    - Atomic = non-interruptable
  - Either test memory word and set value
  - Or swap contents of two memory words

Solution using TestAndSet

- Definition of TestAndSet:

```
boolean TestAndSet (boolean *target) {
          boolean rv = *target;
          *target = TRUE;
          return rv:
     }
```

- Shared boolean variable *lock*., initialized to false.
- Solution:

```
     do {
        while ( TestAndSet (&lock ))
                  ;   /* do nothing
            //    critical section
        lock = FALSE;
            //      remainder section
      } while ( TRUE);
```

Solution using Swap

- Definition of Swap:

```
void Swap (boolean *a, boolean *b) {
        boolean temp = *a;
        *a = *b;
        *b = temp:
 }
```

- Shared Boolean variable *lock* initialized to FALSE; Each process has a local Boolean variable *key*.
- Solution:

```
 do {
        key = TRUE;
         while ( key == TRUE)
                Swap (&lock, &key );
                  //    critical section
          lock = FALSE;
                //      remainder section
       } while ( TRUE);
```

Solution with TestAndSet and bounded wait

- boolean waiting[n]; boolean lock; initialized to false

Pi can enter critical section iff waiting[i] == false or key == false

```
do {
        waiting[i] = TRUE;
        key = TRUE;
        while (waiting[i] && key)
                key = TestAndSet (&lock);
        waiting[i] = FALSE;
        // critical section
        j = (i + 1) % n;
        while ((j != i) && !waiting[j])
                j = (j + 1) % n;
        if (j == i)
                lock = FALSE;
        else
                waiting[j] = FALSE;
        // remainder section
} while (TRUE);
```