

Exam 1 Comments

*It takes a really bad school to ruin a good student
and
a really fantastic school to rescue a bad student.*

General Comments

- ❑ Write your answers in a technical/formal style.
- ❑ Avoid the use of imprecise and non-professional wording and language as computer science is an exact science and we have to learn to communicate in a professional way.
- ❑ Present all key elements as grading is based on how many key elements are answered properly.
- ❑ Justify your answer. For example, if you claim there is a race condition, then show it with execution sequences.
- ❑ ***I do not do grade inflation.***

Problem 1(a)

- ❑ Modern CPUs have two execution modes: the **user** mode and the **supervisor** (or system, kernel, privileged) mode, controlled by a **mode bit**.
- ❑ The OS runs in the supervisor mode and all user programs run in the user mode. Some instructions that may do harm to the OS (e.g., I/O and CPU mode change) are privileged instructions, which, for most cases, can only be used in the supervisor mode.
- ❑ When execution switches to the OS (resp., a user program), execution mode is changed to the supervisor (resp., user) mode.

Problem 1(b)

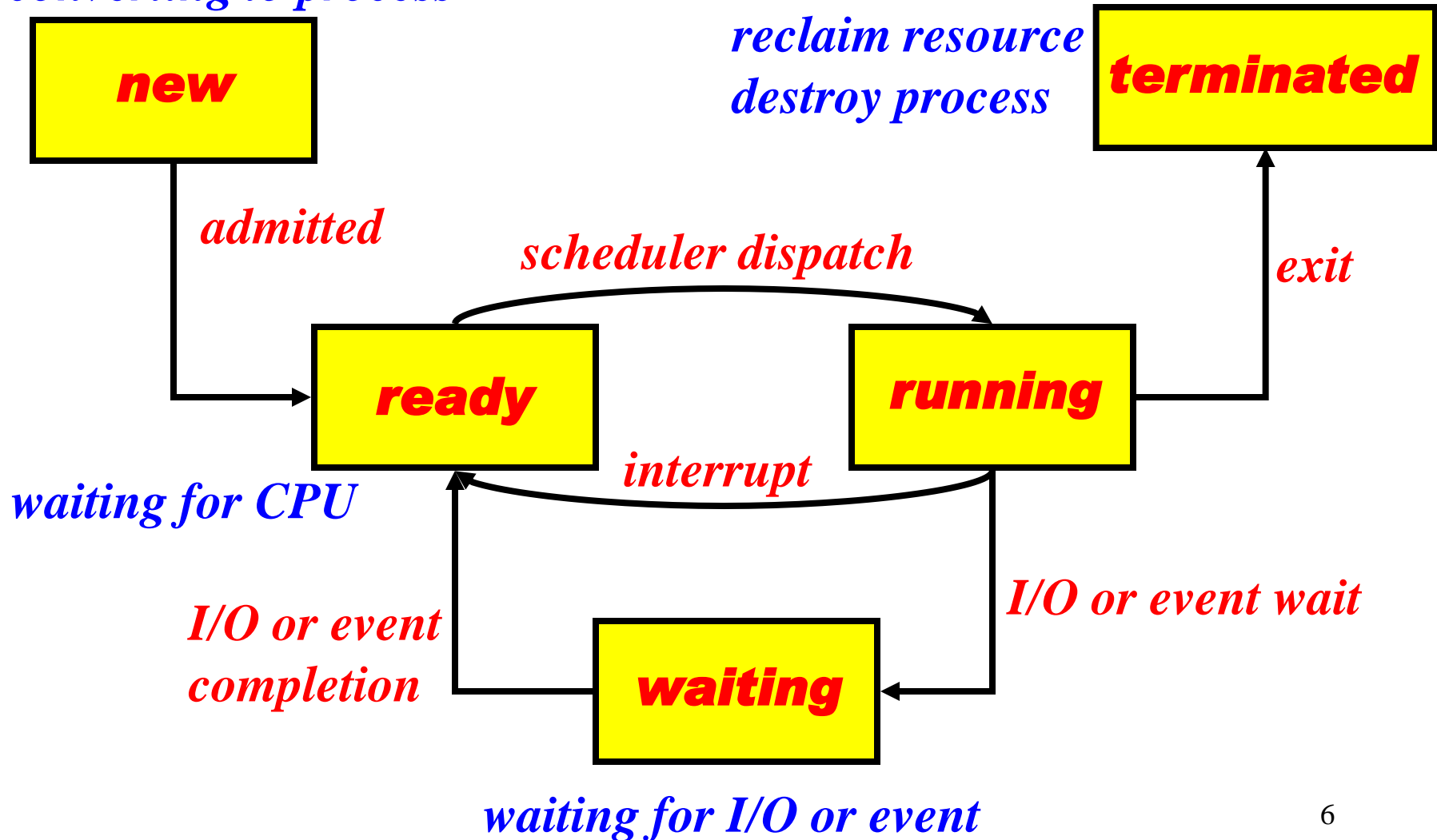
- ❑ An ***interrupt*** is an event that requires OS's attention. It may be generated by hardware (e.g., I/O completion and timer) or software (e.g., system call and division by 0).
- ❑ Interrupts generated by software (e.g., division by 0 and system call) are traps.
- ❑ Don't forget mode switch.
- ❑ Interrupts are ***not signals*** and are ***not called***. Signals have a different meaning in operating systems.

Common Problems

- ❑ Interrupts are not machine instructions, not signals, not functions/procedures.
- ❑ Signals have a different meaning in OS.
- ❑ Interrupts, machine instructions, threads, processes are **NOT** called.
- ❑ OS does not call an interrupt. Except for system calls and a few others, interrupts are **not** called to happen.
- ❑ Many answered this question by stating the result of an interrupt rather than talking about an interrupt itself.

Problem 2(a)

converting to process

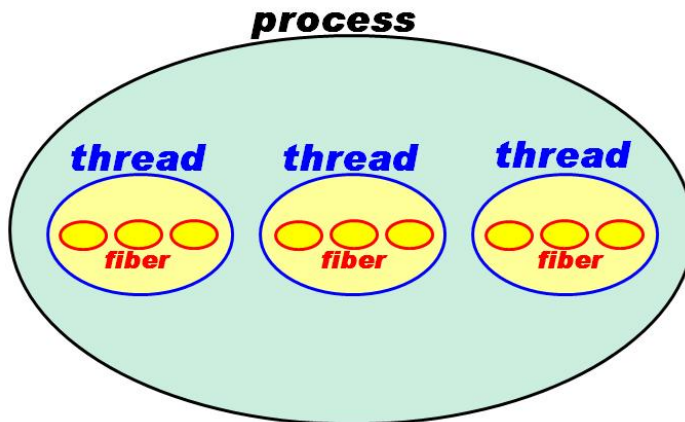


Problem 2(b)

- ❑ The ***context*** of a process is the environment for that process to run properly.
- ❑ This includes process ID, process state, registers, memory areas, a program counter, files, scheduling priority, etc.
- ❑ The sequence of actions are:
 - Control switches back to the OS. Mode switch may be needed.
 - The outgoing process is suspended and its context saved.
 - The context of the incoming process is loaded.
 - Resume its execution. Mode switch may be needed.

Problem 3(a): 1/3

- A process is the execution of a program; a thread (i.e., lightweight process) is a unit of CPU that is created by a process; and a fiber is a lightweight thread, created by a thread.
- A comparison can be based on hierarchy, resource usage and scheduling.
- **Hierarchy**: Processes create thread, and threads create fibers.



A process contains the threads it creates, and a thread contains the fibers it creates.

Problem 3(a): 2/3

Resource Sharing:

- A process acquires all resources to run properly.
- All peer threads created within a process share with each other (i.e., files, memory, etc.). A thread has a program counter, a register set, and a stack, and shares the resources acquired by the containing with other peer threads.
- A fiber also has a program counter, a subset of registers, and a stack. A fiber shares all resources acquired by the containing threads with other fibers.

Problem 3(a): 3/3

□ **Scheduling**:

- **Processes** are scheduled by the CPU scheduler.
- **Threads**:
 - ✓ **User-level threads** are scheduled by a thread scheduler built into a thread library in a user space.
 - **Kernel threads** are scheduled by the kernel thread scheduler in the kernel.
- **Fibers** are scheduled by a co-operative policy. This is usually done using a statement like **YIELD**.

Problem 4(a): 1/9

- ❑ A **race condition** is a situation in which **more than one** processes or threads access a **shared** resource **concurrently**, and the result depends on the **order of execution**.
- ❑ Use instruction level execution sequences for your examples.
- ❑ You must show concurrent sharing in your execution sequences.
- ❑ It takes **two** execution sequences to justify the existence of a race condition, because **you need to show the results depend on the order of execution**.

Problem 4(a): 2/9

int x, a = 4, b = 5;		
exe seq 1	Process 1	Process 2
x = 5	x = a	
		x = b
exe seq 2		x = b
x = 4	x = a	

This is not a valid example to show the existence of a race condition because variable `x` is not shared concurrently.

Problem 4(a): 3/9

```
int Count = 10;
```

Higher-level language statements
are not atomic

Process 1

Process 2



Count = 9, 10 or 11?

Only say `Count++` and `Count--` would cause a race condition is inaccurate because the “sharing” and “concurrent access” conditions are not addressed.

Problem 4(a): 4/9

```
int  Count = 10;
```

Process 1

```
  ⋮  
LOAD  Reg, Count  
ADD   #1  
STORE Reg, Count  
  ⋮
```

Process 2

```
  ⋮  
LOAD  Reg, Count  
SUB   #1  
STORE Reg, Count  
  ⋮
```

The problem is that the execution flow may be switched in the middle. **Possible answers are 9, 10 or 11.**
Show two execution sequences.

Problem 4(a): 5/9

Process 1			Process 2		
Inst	Reg	Memory	Inst	Reg	Memory
LOAD	10	10			
			LOAD	10	10
			SUB	9	10
ADD	11	10			
STORE	11	11	<i>overwrites the previous value 11</i>		
			STORE	9	9

Problem 4(a): 6/9

Process 1			Process 2		
Inst	Reg	Memory	Inst	Reg	Memory
LOAD	10	10			
ADD	11	10			
			LOAD	10	10
			SUB	9	10
			STORE	9	9
STORE	11	11	<i>overwrites the previous value 9</i>		

Problem 4(a): 7/9

- ***You should use instruction level interleaving to demonstrate the existence of race conditions***, because
 - a) higher-level language statements are not atomic and can be switched in the middle of execution
 - b) instruction level interleaving can show clearly the “sharing” of a resource among processes and threads.

Problem 4(a): 8/9

```
int a[3] = { 3, 4, 5};
```

Process 1

```
a[1] = a[0] + a[1];
```

Process 2

```
a[2] = a[1] + a[2];
```

Execution Sequence 1

Process 1	Process 2	Array a[]
$a[1] = a[0] + a[1]$		{ 3, 7, 5 }
	$a[2] = a[1] + a[2]$	{ 3, 7, 12 }

There is no concurrent sharing, not a valid example for a race condition.

Execution Sequence 2

Process 1	Process 2	Array a[]
	$a[2] = a[1] + a[2]$	{ 3, 4, 9 }
$a[1] = a[0] + a[1]$		{ 3, 7, 9 }

Problem 4(a): 9/9

```
int Count = 10;
```

Process 1

```
LOAD  Reg, Count
ADD    #1
STORE Reg, Count
```

Process 2

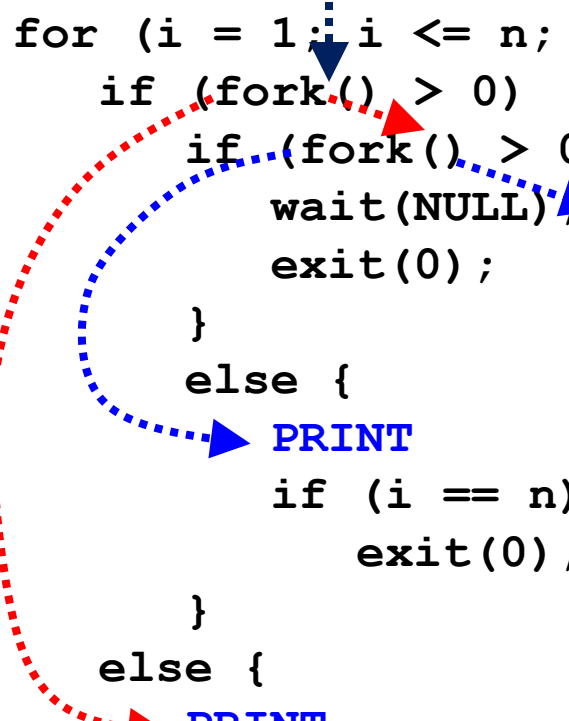
```
LOAD  Reg, Count
SUB    #1
STORE Reg, Count
```

Process 1	Process 2	Memory
LOAD Reg, Count		10
	LOAD Reg, Count	10
	SUB #1	10
ADD #1		10
STORE Reg, Count		11
	STORE Reg, Count	9

variable
count is
shared
concurrently
here

Problem 5(a): 1/2

```
printf("main()'s PID = %ld\n\n", getpid());
for (i = 1; i <= n; i++)    // for each level
    if (fork() > 0)         // parent forks the 1st child
        if (fork() > 0) {   // parent forks the 2nd child
            wait(NULL);     // parent waits
            exit(0);        // and exit
        }
        else {              // the 2nd child process
            PRINT            // print the needed info
            if (i == n)      // if it is a leaf node
                exit(0);    // exit w/o coming back to fork
        }
    else {                  // the 1st child process
        PRINT              // print the needed info
        if (i == n)        // if it is a level node
            exit(0);      // exit w/o coming back to fork
    }
}
```



Problem 5(a): 2/2

```
printf("main()'s PID = %ld\n\n", getpid());
newChild = 2;                // total number of children
while (newChild != 0) {      // parent loops twice
    if (fork() == 0) {        // create a child process
        printf("PID = %d PPID = %d\n", getpid(), getppid());
        depth++;              // depth level increases by 1
        if (depth == n)       // if it is a leaf,
            break;             // get out of the loop
        else
            newChild = 2;      // if it is not a leaf,
    }                          // loops back to create 2 children
    else
        newChild--;           // parent needs 1 more child
}
wait(NULL);                  // the parent waits twice
wait(NULL);                  // because it has 2 children
exit(1);
```

Problem 5(b): 1/3

- Statement interleaving (3, 5 and 6):

Process 1	Process 2	x in memory
	x = 2*x	0
x += 2		2
x++		3

Process 1	Process 2	x in memory
x += 2		2
	x = 2*x	4
x++		5

Process 1	Process 2	x in memory
x += 2		2
x++		3
	x = 2*x	6

Problem 5(b): 2/3

- Instruction interleaving (0, 1 and 4):

Process 1	Process 2	x in memory
	LOAD x	0
	MUL #2	0
x += 2		2
x++		3
	SAVE x	0

Process 1	Process 2	x in memory
	LOAD x	0
	MUL #2	0
x += 2		2
	SAVE x	0
x++		1

Process 1	Process 2	x in memory
x += 2		2
	LOAD x	2
	MUL #2	2
x++		3
	SAVE x	4

Problem 5(b): 3/3

- **The result of 2 is impossible.**
- Note that $x += 2$ and $x = 2 * x$ always produce even numbers, including 0.
- Hence, **Process 1** cannot produce 2 because it has $x++$ making an even number an odd one.
- If 2 is produced by **Process 2**, it is the result of $x = 2 * x$, which means $x = 1$ before $x = 2 * x$. This is impossible because the result of $x += 2$ is an even number.

Process 1

$x += 2;$

$x++;$

Process 2

$x = 2 * x;$

Problem 5(c): 1/2

```
int  status[2];    // status of a process
int  turn;          // initialized to either 0 or 1
```

P_0

```
status[0]=COMPETING;
while (status[1]==COMPETING) {
    status[0]=OUT_CS;
    repeat until (turn==0);
    turn = 0;
    status[0] = COMPETING;
}
```

P_1

```
status[1] = COMPETING;
while (status[0]==COMPETING) {
    status[1]=OUT_CS;
    repeat until (turn==0 || turn==1);
    turn = 1;
    status[1] = COMPETING;
```

Before entering while, status[] is COMPETING

When loops back status[] is set to COMPETING

P_0 enters its critical section

if status[0] is COMPETING and status[1] not COMPETING

P_1 enters its critical section

if status[1] is COMPETING and status[0] not COMPETING

If both P_0 and P_1 are in their critical section, status[0] (and status[1]) must be COMPETING and not COMPETING at the same time.

Problem 5(c): 2/2

```
int  status[2];    // status of a process
int  turn;         // initialized to either 0 or 1
```

P_0

```
status[0]=COMPETING;
while (status[1]==COMPETING) {
    status[0]=OUT_CS;
    repeat until (turn==0);
    turn = 0;
    status[0] = COMPETING;
}
```

P_1

```
status[1] = COMPETING;
while (status[0]==COMPETING) {
    status[1]=OUT_CS;
    repeat until (turn==0 || turn==1);
    turn = 1;
    status[1] = COMPETING;
```

turn plays no role here.

REASON:

If a process sets status[] to COMPETING and finds the other status[] being non-COMPETING, this process enters its critical section.

In this case, the process never sets its status[] to OUT_CS and turn to 0 or 1.

Hence, you should not use turn in your argument,

My Expectation

- ***I expected you to receive approximately 70 points as shown below.***

Problem		Possible	Expected	Class Average	Class Median
1	A	10	8	7	8
	B	10	7	6	7
2	A	10	8	7	8
	B	10	7	6	7
3	A	10	7	5	6
4	A	10	7	6	6
5	A	10	7	4	4
	B	15	10	9	9
	C	15	10	4	1
Total		100	71	56	58

Grade Distribution

Problem-Wise

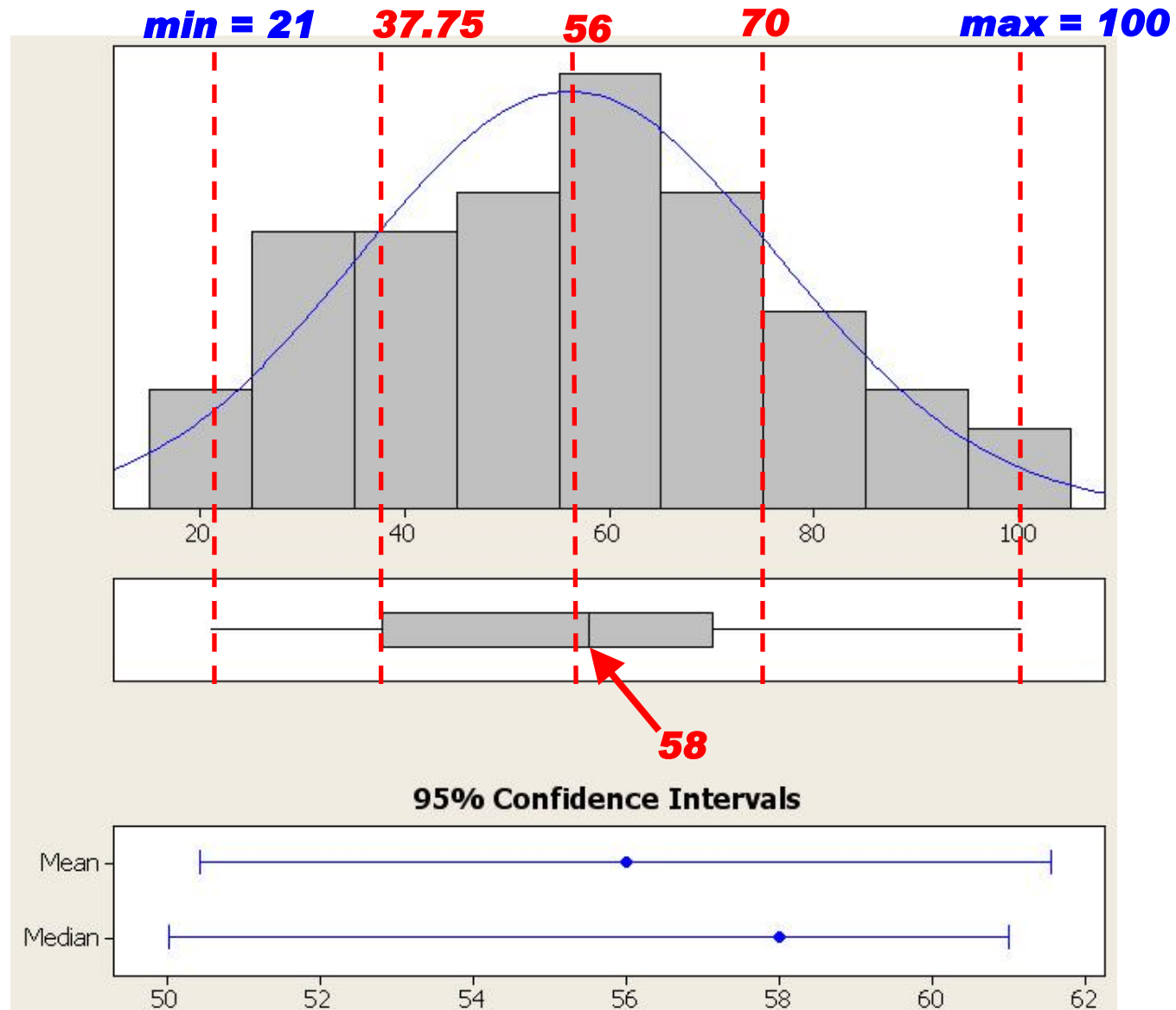
	1a	1b	2a	2b	3	4a	5a	5b	5c	Class
Min	0	0	0	0	0	0	0	0	0	0
Max	10	10	10	10	10	10	10	15	15	100
Median	8	7	8	7	6	6	4	9	1	58
Avg	7	6	7	6	5	6	4	9	4	52
St DEV	3	3	3	3	3	3	3	5	6	20

- Problems 1a, 1b, 2a, 2b, 4a are from our course slides.
- Problem 3a asks you to summarize what you have learned. Everything is on slides.
- Problem 5a tests whether you know `fork()` properly.
- Problem 5b tests whether you can use machine instruction interleaving.
- Problem 5c is a simple problem using prove-by-contradiction.

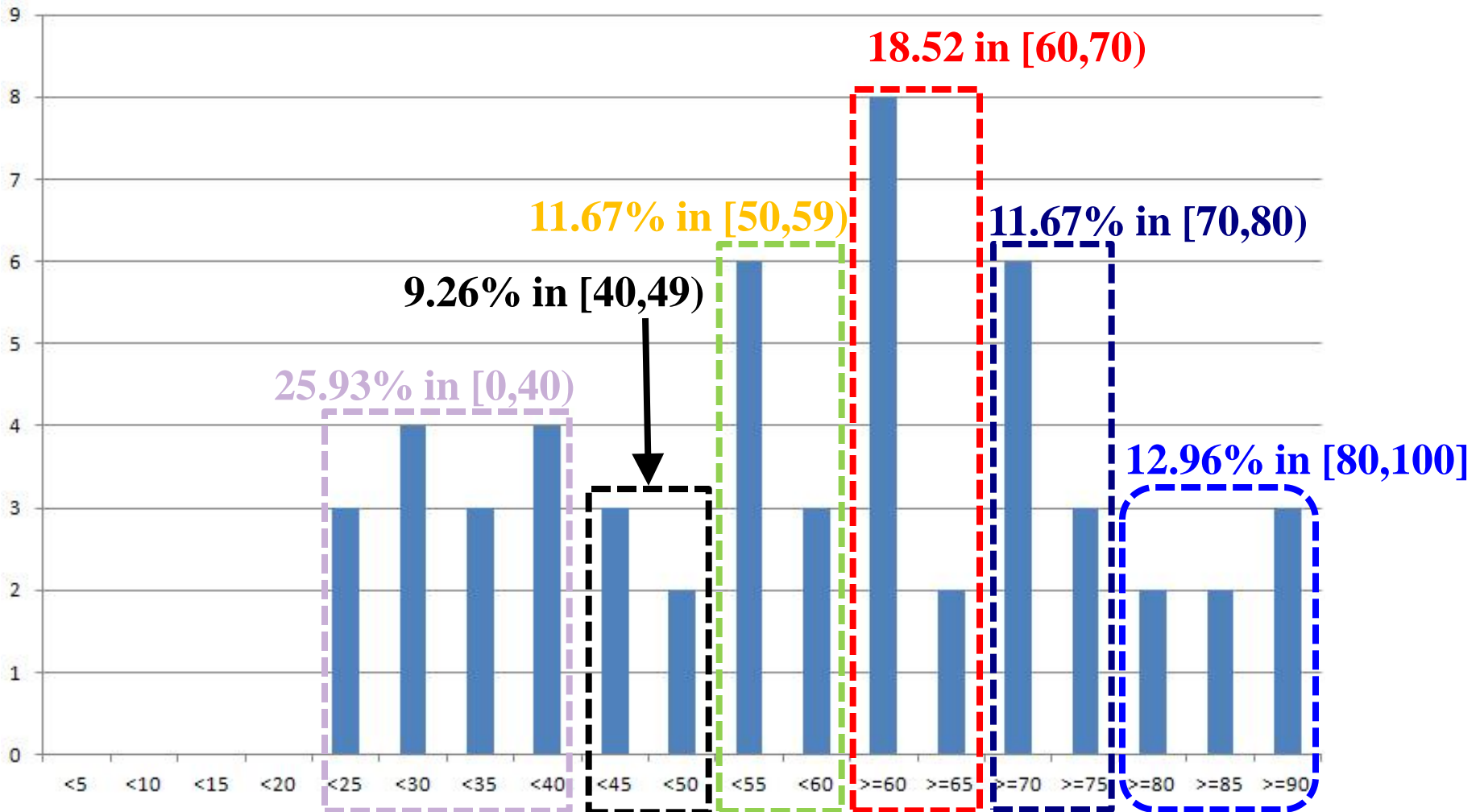
Class Performance

Semester	Average	Standard Dev
2011 Fall	71 best	18
2012 Spring	71	18
2012 Fall	58	14
2013 Spring	61	13
2013 Fall	68	16
2014 Spring	60	19
2014 Fall	57	14
2015 Spring	54	17
2015 Fall	70	17
2016 Spring	55	16
2016 Fall	54	21
2017 Spring	63	16
2017 Fall	53 worst	16
2018 Spring	61	24
2018 Fall	57	19
2019 Spring	56	20

Boxplot



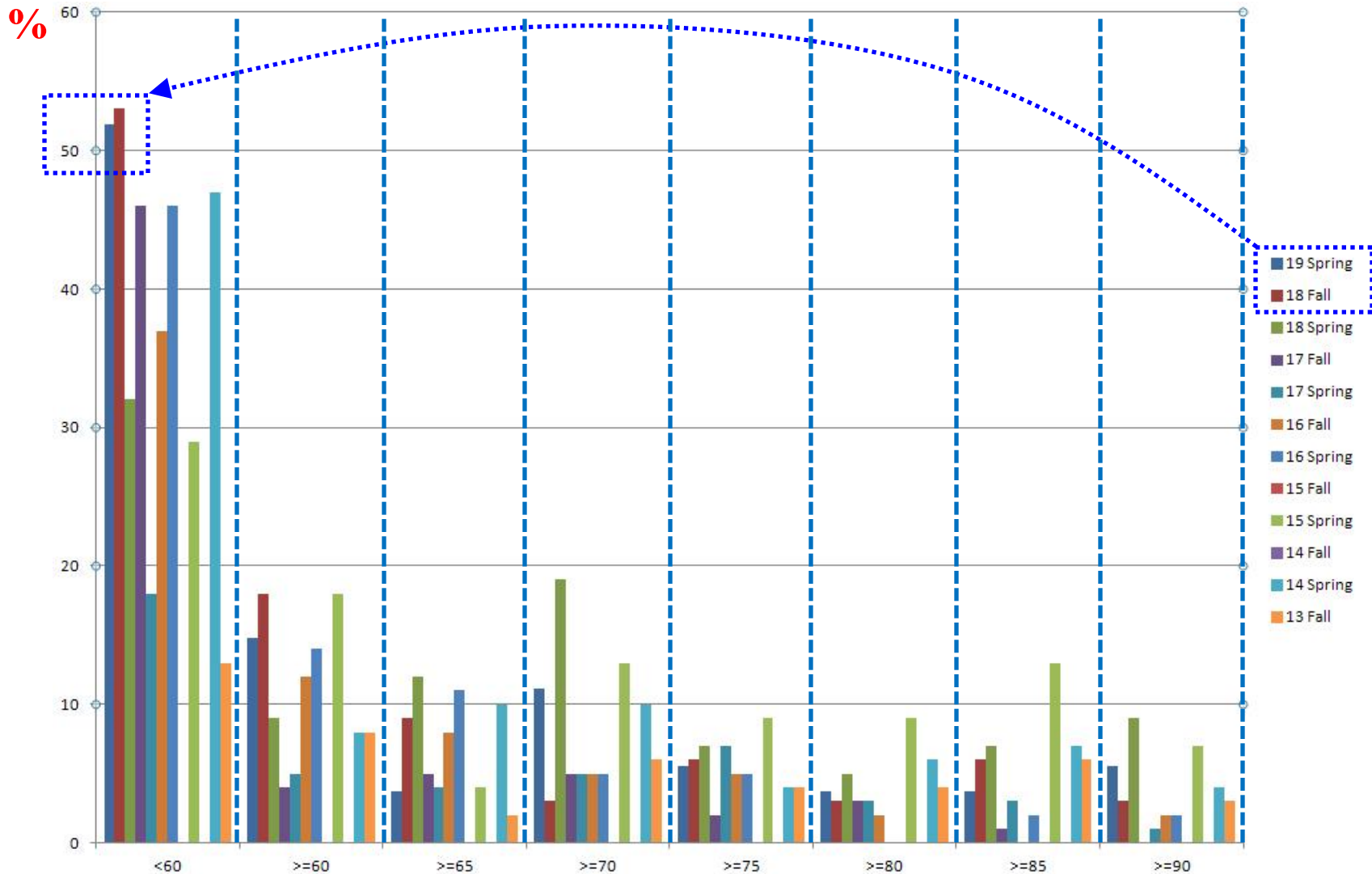
Grade Distribution (2019 Spring)



Grade Distribution (Comparison)

	≥ 80	≥ 70	≥ 60	≥ 50	≥ 40	< 40
2019 Spring	12.96%	11.67%	18.52%	11.67%	9.26%	25.93%
2018 Fall	12%	9%	27%	20%	17%	16%
2017 Fall	6%	11%	14%	32%	17%	14%

Compared with Other Semesters



Compared with recent semesters, the performance of this class is not very good
We have rather high ≤ 60 rate; but we have very high in $[80,100]$

My Findings

- ❑ Many of you did not study the slides carefully. Even the easiest problems were answered poorly/incorrectly.
- ❑ Some just provide an answer or value without elaboration. I am not supposed to finish your answer for you. Whenever a justification and/or elaboration is needed, please do it. ***Use correct wording.***
- ❑ It is still not too late; however, Exam 2 is usually the most difficult one.
- ❑ Please study harder, ask questions, and make sure you understand the subjects.
- ❑ Your grade is proportional to the quality of your answers, and is ***not*** proportional to the time you spent!
- ❑ ***I do not do grade inflation.***

*It takes a really bad school to ruin a good student
and
a really fantastic school to rescue a bad student.*

Dennis J. Frailey

The End