Problem 1: A program's main function is as follows:

```
int main(int argc, char *argv[]) {
      char *str = argv[1];
      while (1)
            printf("%s", str);
      return 0;
}
```

Two processes, both running instances of this program, are currently running (you can assume nothing else of relevance is, except perhaps the shell itself). The programs were invoked as follows, assuming a "parallel command":

```
Shell> main a && main b
```

Below are possible (or impossible?) screen captures of some of the output from the beginning of the run of the programs. Which of the following are possible?
To answer: Fill in A for possible, B for not possible.

1. abababab ... A
2. aaaaaaaa ... **A**
3. bbbbbbbb ... **A**
4. aaaabbbb ... **A**
5. bbbbaaaa ... A

Problem 2: Here is source code for another program, called increment.c:

```
int value = 0;
int main(int argc, char *argv[]) {
      while (1) {
            printf("%d", value);
            value++;
      }
      return 0;
}
```

While increment.c is running, another program, reset.c, is run once as a separate process. Here is the source code of reset.c:

```
int value;
int main(int argc, char *argv[]) {
      value = 0;
      return 0;
}
```

Which of the following are possible outputs of the increment process? To answer: Fill in A for possible, B for not possible.

6. 012345678 ... **A**
7. 012301234 ... **B**
8. 012345670123 ... **B**
9. 01234567891011 ... **A**
10. 123456789 ... **B**

Problem 3: Which of the following are more like policies, and which are more like mechanisms?

To answer: Fill in A for policy, B for mechanism.

11. The timer interrupt **B**
12. How long a time quantum should be **A**
13. Saving the register state of a process **B**
14. Continuing to run the current process when a disk I/O interrupt occurs **A**
15. The address space **B**

Problem 4: A concurrent program (with multiple threads) looks like this:

```
volatile int counter = 1000;
void *worker(void *arg) {
      Counter--;
      return NULL;
}

int main(int argc, char *argv[]) {
      pthread_t p1, p2;
      pthread_create(&p1, NULL, worker, NULL);
      pthread_create(&p2, NULL, worker, NULL);
      pthread_join(p1, NULL);
      pthread_join(p2, NULL);
      printf("%d\n", counter); return 0;
}
```

Assuming pthread create() and pthread join() all work as expected (i.e., they don't return an error), which outputs are possible? To answer: Fill in A for possible, B for not possible.

16. 0 **B**
17. 1000 **B**
18. 999 **A**
19. 998 **A**
20. 1002 **B**

Problem 5: Processes exist in a number of different states. We've focused upon a few (Running, Ready, and Blocked) but real systems have slightly more. For example, xv6 also has an Embryo state (used when the process is being created), and a Zombie state (used when the process has exited but its parent hasn't yet called wait() on it).

Assuming you start observing the states of a given process at some point in time (not necessarily from its creation, but perhaps including that), which process states could you possibly observe? To answer: Fill in A for possible, B for not possible.

Note: once you start observing the process, you will see ALL states it is in, until you stop sampling.

21. Running, Running, Running, Ready, Running, Running, Running, Ready **A**
22. Embryo, Ready, Ready, Ready, Ready, Ready **A**
23. Running, Running, Blocked, Blocked, Blocked, Running **B**
24. Running, Running, Blocked, Blocked, Blocked, Ready, Running **A**
25. Embryo, Running, Blocked, Running, Zombie, Running **B**

Problem 6: The following code is shown to you:

```
int main(int argc, char *argv[]) {
      printf("a");
      fork();
      printf("b");  return
      0;
      }
```

Assuming fork() succeeds and printf() prints its outputs immediately (no buffering occurs), what are possible outputs of this program? To answer: Fill in A for possible, B for not possible.

26. ab **B**
27. abb **A**
28. bab **B**
29. bba **B**
30. a **B**

Problem 7: Assuming fork() might fail (by returning an error code and not creating a new process) and printf() prints its outputs immediately (no buffering occurs), what are possible outputs of the same program as above? To answer: Fill in A for possible, B for not possible.

31. ab **A**
32. abb **A**
33. bab **B**
34. bba **B**
35. a **B**

Problem 8: Here is even more code to look at. Assume the program /bin/true, when it runs, never prints anything and just returns 0 in all cases.

```
int main(int argc, char *argv[]) {
     int rc = fork();
     if (rc == 0) {
          char *my_argv[] = { "/bin/true", NULL };
          execv(my_argv[0], my_argv);
          printf("1");
     } else if (rc > 0) {
          wait(NULL);
          printf("2");
     } else {
          printf("3");
     }
     return 0;
}
```

Assuming all system calls succeed and printf() prints its outputs immediately (no buffering occurs), what outputs are possible?
To answer: Fill in A for possible, B for not possible.

36. 123 **B**
37. 12 **B**
38. 2 **A**
39. 23 **B**
40. 3 **B**

Problem 9: Same code snippet as in the last problem, but new question: assuming any of the system calls above might fail (by not doing what is expected, and returning an error code), what outputs are possible? Again assume that printf() prints its outputs immediately (no buffering occurs). To answer: Fill in A for possible, B for not possible.

41. 123 **B**
42. 12 **A – child execv fails, printf(1) then parent prints 2**
43. 2 **A – same as 8**
44. 23 **B**
45. 3 **A**

Problem 10: Assume, for the following jobs, a FIFO scheduler and only one CPU. Each job has a "required" runtime, which means the job needs that many time units on the CPU to complete.

Job A arrives at time=0, required runtime=X time units; Job B arrives at time=5, required runtime=Y time units; Job C arrives at time=10, required runtime=Z time units.

Assuming an average turnaround time between 10 and 20 time units (inclusive), which of the following run times for A, B, and C are possible?
To answer: Fill in A for possible, B for not possible.
46. A=10, B=10, C=10 **A**
47. A=20, B=20, C=20 **B**
48. A=5, B=10, C=15 **A**
49. A=20, B=30, C=40 **B**
50. A=30, B=1, C=1 **B**

Problem 11: Assume the following schedule for a set of three jobs, A, B, and C:

A runs first (for 10 time units) but is not yet done
B runs next (for 10 time units) but is not yet done
C runs next (for 10 time units) and runs to completion
A runs to completion (for 10 time units)
B runs to completion (for 5 time units)

Which scheduling disciplines could allow this schedule to occur? To answer: Fill in A for possible, B for not possible.

51. FIFO **B**
52. Round Robin **A**
53. STCF (Shortest Time to Completion First) **A**
54. Multi-level Feedback Queue **A**
55. Lottery Scheduling **A**

Problem 12: The Multi-level Feedback Queue (MLFQ) is a fancy scheduler that does lots of things. Which of the following things could you possibly say (correctly!) about the MLFQ approach?
To answer: Fill in A for things that are true about MLFQ, B for things that are not true about MLFQ.

56. MLFQ learns things about running jobs **A**
57. MLFQ starves long running jobs **B**
58. MLFQ uses different length time slices for jobs **A**
59. MLFQ uses round robin **A**
60. MLFQ forgets what it has learned about running jobs sometimes **A**

Problem 13: The simplest technique for virtualizing memory is known as dynamic relocation, or "base- and-bounds". Assuming the following system characteristics:

- a 1KB virtual address space
- a base register set to 10000
- a bounds register set to 100

1KB virtual address space means 1024 bytes can be accessed by program, from 0 … 1023. Baseand-bounds places this address space into physical memory at physicall address 10,000 (as above). However, only the first 100 virtual addresses are legal (0 … 99), which translate to physical addresses 10,000 … 10,099. Thus:
Which of the following physical memory locations can be legally accessed by the running program?
To answer: Fill in A for legally accessible locations, B for locations not legally accessible by this program.

61. 0 **B**
62. 1000 **B**
63. 10000 **A**
64. 10050 **A**
65. 10100 **B**

Problem 14: Assuming the same setup as above (1 KB virtual address space, base=10000, bounds=100), which of the following virtual addresses can be legally accessed by the running program? (i.e., which are valid?) To answer: Fill in A for valid virtual addresses, B for not valid ones.

66. 0 **A**
67. 1000 **B**
68. 10000 **B**
69. 10050 **B**
70. 10100 **B**

Problem 15: Segmentation is a generalization of base-and-bounds. Which possible advantages does segmentation have as compared to base-and-bounds?

Segmentation is a slight generalization of base-and-bounds. It has a few base/bound register pairs per process, instead of just one.

To answer: Fill in A for cases where the statement is true about segmentation and (as a result) segmentation has a clear advantage over base-and-bounds, B otherwise.

71. Faster translation **B**
72. Less physical memory waste **A**
73. Better sharing of code in memory **A**
74. More hardware support needed to implement it **B**
75. More OS issues to handle, such as compaction **B**

Problem 16: Assume the following in a simple segmentation system that supports two segments: one (positive growing) for code and a heap, and one (negative growing) for a stack:

- Virtual address space size 128 bytes (small!)
- Physical memory size 512 (small!)

FIRST 20 OF 128 and LAST 20 are VALID

Segment register information:
Segment 0 base (grows positive) : 0
Segment 0 limit: 20 (decimal)
Segment 1 base (grows negative) : 0x200 (decimal 512)
Segment 1 limit: 20 (decimal)

Which of the following are valid virtual memory accesses?
To answer: Fill in A for valid virtual accesses, B for non-valid accesses.

76. 0x1d (decimal: 29) **B**
77. 0x7b (decimal: 123) **A**
78. 0x10 (decimal: 16) **A**
79. 0x5a (decimal: 90) **B**
80. 0x0a (decimal: 10) **A**

Problem 17: In a simple page-based virtual memory, with a linear page table, assume the following:

- virtual address space size is 128 bytes (small!)
- physical memory size of 1024 bytes (small!)
- page size of 16 bytes

The format of the page table: The high-order (leftmost) bit is the VALID bit. If the bit is 1, the rest of the entry is the PFN. If the bit is 0, the page is not valid.

Here are the contents of the page table (from entry 0 down to the max size)
[0] **0x80000034**

[1] 0x00000000

[2] 0x00000000

[3] 0x00000000 [4] **0x8000001e**

[5] **0x80000017**

[6] **0x80000011**

[7] **0x8000002e**

From the page table, we can look for entries that are valid by looking for an 0x8 in the upper-most bits. We can thus see that virtual pages 0, 4, 5, 6, and 7 are valid; the rest are not. We thus need a way to determine, for a given virtual address, which virtual page it is referring to (i.e., what is its VPN?) Each virtual address is 7 bits long (128 byte virtual address space), with 16 byte pages. Thus, the page offset is 4 bits, leaving the top 3 bits for the VPN. Thus, the first hex digit below tells us the VPN and thus whether each access is valid or not

Which of the following virtual addresses are valid?
To answer: Fill in A for valid virtual accesses, B for non-valid accesses.

81. 0x34 (decimal: 52) **B**
82. 0x44 (decimal: 68) **A**
83. 0x57 (decimal: 87) **A**
84. 0x18 (decimal: 24) **B**
85. 0x46 (decimal: 70) **A**

Problem 18: TLBs are a critical part of modern paging systems. Assume the following system:

- page size is 64 bytes
- TLB contains 4 entries
- TLB replacement policy is LRU (least recently used)

Each of the following represents a virtual memory address trace, i.e., a set of virtual memory addresses referenced by a program. In which of the following traces will the TLB possibly help speed up execution?
To answer: Fill in A for cases where the TLB will speed up the program, B for the cases where it won't.

The key ends up being: does having a 4-entry (LRU) TLB improve performance? If any TLB hits occur, the answer is yes, otherwise no. Page size 64 bytes, and we need to use that to figure out how many pages are being accessed, and which ones.

86. 0, 100, 200, 1, 101, 201, ... (repeats in this pattern) **A**
87. 0, 100, 200, 300, 0, 100, 200, 300, ... (repeats) **A**
88. 0, 1000, 2000, 3000, 4000, 0, 1000, 2000, 3000, 4000, ... (repeats) **B**
89. 0, 1, 2, 3, 4, 5, 6, 0, 1, 2, 3, 4, 5, 6, ... (repeats) **A**
90. 300, 200, 100, 0, 300, 200, 100, 0, ... (repeats) **A**

2. For a workload consisting of ten CPU-bound jobs of all the same length (each would run for 10 seconds in a dedicated environment), which policy would result in the **lowest average response time**? **Please circle ONE answer.**

   (a) Round-robin with a 100 millisecond quantum
   (b) Shortest Job First
   (c) Shortest Time to Completion First
   (d) Round-robin with a 1000 nanosecond quantum

Response time is the time from the demand from service until the first service. Shortest-job first and shortest-time-to-completion will make some processes (long ones) wait arbitrarily long, so those can be ruled out. Thus, it is clearly one of the RRs. The one with the shorter time quantum wins (1000 ns is much less than 100 ms).

9. Which of the following will **NOT** guarantee that deadlock is avoided? **Please circle all that apply.**

   (a) Acquire all resources (locks) all at once, atomically
   (b) Use locks sparingly
   (c) Acquire resources (locks) in a fixed order
   (d) Be willing to release a held lock if another lock you want is held, and then try the whole thing over again

Using locks sparingly does not do anything for you, and can still lead to deadlock.

11. For a workload consisting of ten CPU-bound jobs of varying lengths (half run for 1 second, and the other half for ten seconds), which policy would result in the lowest total run time **for the entire workload? Please circle all that apply.**

   (a) Shortest Job First
   (b) Shortest-Time to Completion First
   (c) Round-robin with a 100 millisecond quantum
   (d) Multi-level Feedback Queue

Total time is not altered by scheduling policy, and thus all policies are equivalent. You could argue that SJF is the fastest, because of a lack of context switches. But then you would have to know the parameters of the MLFQ, which were not specified...

Problem 1b[3pts]: What are some of the hardware differences between kernel mode and user mode? Name at least three.

There are many differences. They all involve protection. Here are a few possible answers:
1) *There is a bit in a control register that is different (say 0=kernel, 1=user).*
2) *Hardware access to devices is usually unavailable in user mode.*
3) *Some instructions are available only in kernel mode.*
4) *Modifications to the page tables are only possible in kernel mode.*
5) *The interrupt controller can only be modified in kernel mode.*
6) *Other hardware control registers (such as system time, timer control, etc) are available only in kernel mode.*
7) *Kernel memory is not available to users in user mode.*

Grading Scheme: One point for each of the three differences