CS-537: Midterm (Spring 2018)
Mission Impossible

Please Read All Questions Carefully!

There are 16 numbered pages, 30 Problems, and 150 answers to fill in.

Directions

With the operating system, all is possible, or is it?

In this exam, we'll explore what is possible and what isn't, in touching on topics as heady as concurrency and as detailed as page-based memory virtualization.

Most of the questions ask simply: is this outcome possible, or not? A few veer from this basic structure, because nothing is perfect in life, even 537 exams.

Your mission, should you choose to accept it: Answer all of the questions below correctly and without fear. Fill in only A or B for each question (not both!). And, most importantly, do so in pencil.

(and the famous Mission Impossible theme song starts .... now!)

Important stuff:

- Fill in your **name** and **student ID** carefully on the answer sheet.
- Fill in 0 for Special Code A if you are a graduate student, 1 for Special Code A if you are an undergraduate (this is just for data analysis, it doesn't affect your grade).
- Fill out A or B (but not both!) for each of 150 questions.
- Color each oval for A or B completely; don't use a checkmark, box around the oval, or other weird things that only a desperate student can think of.
- Fill in the oval with pencil, not pen.
- If you skip a question, be careful and make sure to fill in the correct bubbles! Pay careful attention to numbering.

Problem I: 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" as per project 2a (the wish shell):

```
wish> 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.

When two processes run in parallel, their outputs can be arbitrarily interleaved. Thus, any mix of 'a' and 'b' is possible.

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

Problem II: 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.

The program 'reset' is running in a separate process. It's updating its own 'value', which has nothing to do with the 'value' in increment. Thus, the question is basically asking, "what are legal outputs of the 'increment' process?" Below are such answers.

6. 012345678 ... **A. Possible**
7. 012301234 ... **B. Not Possible** (value is reset, but how?)
8. 012345670123 ... **B. Not Possible** (value is reset, but how?)
9. 01234567891011 ... **A. Possible** (value not reset; it's just "9 10 11" squished together)
10. 123456789 ... **B. Not Possible** (increment starts at 0)

Problem III: 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.

In this code sequence, there is a race on the decrement of counter. If the threads perform the decrement one after the other, the "right" result will occur: 998. However, if the decrements take place concurrently, each will read 1000 as the value of the counter, decrement it to 999, and then write 999 back to the counter, thus losing one decrement. All other outputs should not be possible.

11. 0 **B. Not Possible**
12. 1000 **B. Not Possible**
13. 999 **A. Possible** (race on counter; if both read before decrement...)
14. 998 **A. Possible** (race, but one decrements before the other)
15. 1002 **B. Not Possible**

Problem IV: 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?

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

To answer: Fill in **A** for possible, **B** for not possible.

Embryo transitions to Ready; Ready to Running; Running to either Blocked (if an I/O is issued, or if waiting on a condition variable/semaphore, etc.) or back to Ready (if descheduled) or to Zombie if exited; Blocked back to Ready (if I/O completes, or signal on condition variable takes place, etc.); there is no transition out of Zombie. Thus, the first case below is possible if a process has been running, then gets descheduled, the runs again, and is descheduled again; the second case represents a process being created and then waiting to run; the third case cannot happen because a process must be Ready before it runs again; the fourth case is possible because a process runs, then blocks on I/O, then unblocks (becomes Ready), then runs; the last case is not possible for many reasons, one of which there is no state ever beyond Zombie.

16. Running, Running, Running, Ready, Running, Running, Running, Ready **A. Possible**
17. Embryo, Ready, Ready, Ready, Ready, Ready **A. Possible**
18. Running, Running, Blocked, Blocked, Blocked, Running **B. Not Possible**
19. Running, Running, Blocked, Blocked, Blocked, Ready, Running **A. Possible**
20. Embryo, Running, Blocked, Running, Zombie, Running **B. Not Possible**

Problem V: 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.

fork() creates a new process (when it works, as it does here) which is a copy of the existing process. In this case, because printf() takes place immediately, 'a' prints in the child, and then each the parent and child print 'b'. Thus, 'abb' is the output. It is interesting to think about what would happen if 'a' is buffered by the parent before printing ... (but outside the scope of this question).

21. ab **B. Not Possible**
22. abb **A. Possible**
23. bab **B. Not Possible**
24. bba **B. Not Possible**
25. a **B. Not Possible**

Problem VI: 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.

Here, if fork() fails, the parent will still complete, thus printing 'a' and 'b', and thus adding one more option to the choices below.

26. ab **A. Possible**
27. abb **A. Possible**
28. bab **B. Not Possible**
29. bba **B. Not Possible**
30. a **B. Not Possible**

Problem VII: 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.

The common case is that the child (rc==0) case exec's /bin/true; it will thus never print 1. The parent (rc>0) prints 2 after waiting for the child to complete.

31. 123 **B. Not Possible**
32. 12 **B. Not Possible**
33. 2 **A. Possible**
34. 23 **B. Not Possible**
35. 3 **B. Not Possible**

Problem VIII: 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.

As before, everything could work, and thus only '2' get printed again. If the fork() succeeds but the execv() fails, the child will return from execv(), the child will print '1'. Thus, '12' is possible (the '2' only prints after the '1' because of the wait() - note that '21' is possible here if the wait() fails, but not an option below. If fork() fails (rc<0), just '3' is printed. Any output with '2' and '3' is not possible, because that means the fork() must have succeeded and failed at the same time.

36. 123 **B. Not Possible**
37. 12 **A. Possible**
38. 2 **A. Possible**
39. 23 **B. Not Possible**
40. 3 **A. Possible**

Problem IX: 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.

Turnaround time is the time from job arrival to job completion. Average turnaround is just the computed average of the turnaround times of a number of jobs. We now just compute the turnaround for each of the jobs below, and then see if the average is between 10 and 20, inclusive. With FIFO and A arriving before B arriving before C, we know that in each case, A runs before B runs before C.

41. A=10, B=10, C=10 *A's turnaround: 10-0=10, B: 20-5=15; C: 30-10=20. Avg: 15* **A. Possible**
42. A=20, B=20, C=20 *A: 20-0=20; B: 40-5=35; C: 60-10=50; Avg: 35* **B. Not Possible**
43. A=5, B=10, C=15 *A: 5-0=5; B: 15-5=10; C: 30-10=20; Avg: 35/3=11.67* **A. Possible**
44. A=20, B=30, C=40 *A: 20-0=20; B: 50-5=45; C: 90-10=80; Avg: 48.33* **B. Not Possible**
45. A=30, B=1, C=1 *A: 30-0=30; B=31-5=26; C=32-10=22; Avg: 26.* **B. Not Possible** (should have set A=22 or so; oh well)

Problem X: 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.

46. FIFO **B. Not Possible** *FIFO would run to completion, the schedule above does not*
47. Round Robin **A. Possible** *Switch jobs in RR order every 10 time units.*
48. STCF (Shortest Time to Completion First) **B. Not Possible** *Not possible because A is run to completion before B, which is shorter.*
49. Multi-level Feedback Queue **A. Possible** *Can act like RR, so is possible. Could have stayed on same queue, or switch queues.*
50. Lottery Scheduling **A. Possible** *With lottery (random), anything is possible.*

Problem XI: 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.

51. MLFQ learns things about running jobs **A. Possible/True** *By moving jobs down queues, it learns that they are long running (for example).*
52. MLFQ starves long running jobs **B. Not Possible/False** *By bumping priority on occasion, MLFQ avoids starvation.*
53. MLFQ uses different length time slices for jobs **A. Possible** *True, sometimes, across different queues.*
54. MLFQ uses round robin **A. Possible** *True, within a given level.*
55. MLFQ forgets what it has learned about running jobs sometimes **A. Possible** *True, with priority bump to top priority, all jobs look the same now, and all is forgotten about them.*

Problem XII: 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
```

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.

1KB virtual address space means 1024 bytes can be accessed by program, from 0 ... 1023. Base-and-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:

56. 0 **B. Not Possible**
57. 1000 **B. Not Possible**
58. 10000 **A. Possible**
59. 10050 **A. Possible**
60. 10100 **B. Not Possible**

Problem XIII: Assuming the same set-up 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.

Answer explained above.

61. 0 **A. Possible**
62. 1000 **B. Not Possible**
63. 10000 **B. Not Possible**
64. 10050 **B. Not Possible**
65. 10100 **B. Not Possible**

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

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

66. Faster translation **B. Not Possible** *Translation with Segmentation is not faster.*
67. Less physical memory waste **A. Possible** *Less waste is possible, because space between a stack and heap need not be allocated.*
68. Better sharing of code in memory **A. Possible** *Code segments, marked read only, can be shared.*
69. More hardware support needed to implement it **B. Not Possible** *More hardware support is needed, BUT, this is not an advantage.*
70. More OS issues to handle, such as compaction **B. Not Possible** *More OS issues must be handled, BUT, this is not an advantage.*

Problem XV: 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!)

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.

With the above address space, virtual addresses 0 ... 19 are valid (the first 20 bytes of the AS, as described by the segment 0 base/limit pair), and virtual addresses 127 ... 108 are valid too (the last 20 bytes of the AS. It doesn't matter where they map to in physical space for this question. Thus:

71. 0x1d (decimal: 29) **B. Not Possible**
72. 0x7b (decimal: 123) **A. Possible**
73. 0x10 (decimal: 16) **A. Possible**
74. 0x5a (decimal: 90) **B. Not Possible**
75. 0x0a (decimal: 10) **A. Possible**

Problem XVI: 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
```

Which of the following virtual addresses are **valid**?

To answer: Fill in **A** for valid virtual accesses, **B** for non-valid accesses.

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.

76. 0x34 (decimal: 52) **B. Not Valid** *VPN=3*
77. 0x44 (decimal: 68) **A. Valid** *VPN=4*
78. 0x57 (decimal: 87) **A. Valid** *VPN=5*
79. 0x18 (decimal: 24) **B. Not Valid** *VPN=1*
80. 0x46 (decimal: 70) **A. Valid** *VPN=4*

Problem XVII: 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.

81. 0, 100, 200, 1, 101, 201, ... (repeats in this pattern) **A. Speed up** *Three pages accessed repeatedly; first miss, miss, miss, but then hit, hit hit, etc.*
82. 0, 100, 200, 300, 0, 100, 200, 300, ... (repeats) **A. Speed up** *4 pages being accessed repeatedly, misses the first time, then hits.*
83. 0, 1000, 2000, 3000, 4000, 0, 1000, 2000, 3000, 4000, ... (repeats) **B. No Speedup** *5 pages being accessed in a cyclical pattern; result is that each access is a TLB miss.*
84. 0, 1, 2, 3, 4, 5, 6, 0, 1, 2, 3, 4, 5, 6, ... (repeats) **A. Speed up** *Just one page being accessed repeatedly, so mostly hits (except first access).*
85. 300, 200, 100, 0, 300, 200, 100, 0, ... (repeats) **A. Speed up** *4 pages accessed repeatedly, hence mostly hits.*

Problem XVIII: Which of the following statements are true statements about various page-replacement policies?
To answer: Fill in **A** for true statements, **B** for false ones.

86. The LRU policy always outperforms the FIFO policy. **B. False** *LRU has some bad cases, thus it's not guaranteed to be better than FIFO.*
87. The OPT (optimal) policy always performs at least as well as LRU. **A. True** *OPT cannot be outperformed (in terms of hit rate); thus it must be at least as good as LRU.*
88. A bigger cache's hit percentage is always greater than or equal to a smaller cache's hit percentage, if they are using the same replacement policy. **B. False** *See Belady's anomaly.*
89. A bigger cache's hit percentage is always greater than or equal to a smaller cache's hit percentage, if they are using the LRU replacement policy. **A. True** *True because a smaller LRU cache is just a subset of a bigger LRU cache; therefore, can't get a higher hit rate from a smaller cache when LRU is the policy (might be the same though).*
90. Random replacement is always worse than LRU replacement. **B. False** *Random could be better, just depends on the luck of the draw.*

Problem XIX: Assume a memory that can hold 4 pages, and an LRU replacement policy. The first four references to memory are to pages 6, 7, 7, 9.

Assuming the next five accesses are to pages 7, 9, 0, 4, 9, which of those will hit in memory? (and which will miss?)

To answer: Fill in **A** for cache hits, **B** for misses.

You just have to run a simulation here. Perhaps with a homework simulator?

```
prompt> ./paging-policy.py -c -p LRU -s 5 -C 4
// priming the cache ...
Access: 6  MISS LRU ->          [6] <- MRU Replaced:- [Hits:0 Misses:1]
Access: 7  MISS LRU ->       [6, 7] <- MRU Replaced:- [Hits:0 Misses:2]
Access: 7  HIT  LRU ->       [6, 7] <- MRU Replaced:- [Hits:1 Misses:2]
Access: 9  MISS LRU ->    [6, 7, 9] <- MRU Replaced:- [Hits:1 Misses:3]
// now the questions below:
Access: 7  HIT  LRU ->    [6, 9, 7] <- MRU Replaced:- [Hits:2 Misses:3]
Access: 9  HIT  LRU ->    [6, 7, 9] <- MRU Replaced:- [Hits:3 Misses:3]
Access: 0  MISS LRU -> [6, 7, 9, 0] <- MRU Replaced:- [Hits:3 Misses:4]
Access: 4  MISS LRU -> [7, 9, 0, 4] <- MRU Replaced:6 [Hits:3 Misses:5]
Access: 9  HIT  LRU -> [7, 0, 4, 9] <- MRU Replaced:- [Hits:4 Misses:5]
```

91. 7 **A. Hit**
92. 9 **A. Hit**
93. 0 **B. Miss**
94. 4 **B. Miss**
95. 9 **A. Hit**

Problem XX: Assume this attempted implementation of a lock:

```
void init(lock_t *mutex) {
  mutex->flag = 0;  // 0 -> lock is available, 1 -> held
}
void lock(lock_t *mutex) {
  while (mutex->flag == 1) // L1
    ;                      // L2
  mutex->flag = 1;         // L3
}
void unlock(lock_t *mutex) {
  mutex->flag = 0;         // L4
}
```

Assume 5 threads are competing for this lock. How many threads can possibly acquire the lock?
To answer: Fill in **A** for possible, **B** for not possible.

Unfortunately, two interpretations of this question. One is "how many threads can acquire the lock *at the same time*"; the other is "how many threads can acquire the lock *at all*". Fortunately, answer is the same in this case, because the lock is broken; it doesn't use an atomic exchange, and thus has a race condition in its very definition. Multiple threads, calling lock() at the same time, may acquire the lock at the same time! Thus, all five answers are possible.

96. 1 **A. Possible**
97. 2 **A. Possible**
98. 3 **A. Possible**
99. 4 **A. Possible**
100. 5 **A. Possible**

Problem XXI: Here is a ticket lock:

```
typedef struct __lock_t {
    int ticket, turn;
} lock_t;
void lock_init(lock_t *lock) {
    lock->ticket = 0;
    lock->turn   = 0;
}
void lock(lock_t *lock) {
    int myturn = FetchAndAdd(&lock->ticket);
    while (lock->turn != myturn)
        ; // spin
}
void unlock(lock_t *lock) {
    lock->turn = lock->turn + 1;
}
```

Assuming a maximum of 5 threads in the system, and further assuming the ticket lock is used "properly" (i.e., threads acquire and release it as expected), what values of `lock->ticket` and `lock->turn` are possible? (at the same time) **To answer:** Fill in **A** for possible, **B** for not possible.

This question requires us to trace through possible behaviors of the ticket lock, assuming 5 threads running and repeatedly acquiring and releasing the lock. Can the various values of ticket and turn arise?

101. `ticket=0` and `turn=0` **A. Possible** *Before any thread does anything.*
102. `ticket=0` and `turn=1` **B. Not Possible** *`ticket` is incremented before `turn`, thus this should not happen.*
103. `ticket=1` and `turn=0` **A. Possible** *After one thread calls lock(), ticket is now 1, and turn is 0.*
104. `ticket=16` and `turn=5` **B. Not Possible** *With only 5 threads, `ticket` can't get this far ahead of `turn`.*
105. `ticket=1000` and `turn=999` **A. Possible** *As before, `ticket` is at 999, then one thread calls lock(), resuting in this state.*

Problem XXII: Assume the following list insertion code, which inserts into a list pointed to by shared global variable `head`:

```
int List_Insert(int key) {
  node_t *n = malloc(sizeof(node_t));
  if (n == NULL) { return -1; }
  n->key = key;
  n->next = head;
  head = n;
  return 0;
}
```

This code is executed by each of three threads exactly once, without adding any synchronization primitives (such as locks). Assuming `malloc()` is thread-safe (i.e., can be called without worries of data races) and that `malloc()` returns successfully, how long might the list be when these three threads are finished executing? (assume the list was empty to begin)
To answer: Fill in **A** for possible, **B** for not possible.

Assumes list empty at beginning. Because there is a race to include a node into the list, it is possible that a node gets dropped during insert. If we have three threads doing an insert, they call could succeed (they get serialized for some reason), or only one could succeed (with the other two updates lost). In no case can all get lost. Thus:

106. 0 **B. Not Possible**
107. 1 **A. Possible**
108. 2 **A. Possible**
109. 3 **A. Possible**
110. 4 **B. Not Possible**

Problem XXIII: Assume the following code, in which a "background malloc" allocates memory in a thread and initializes it:

```
void *background_malloc(void *arg) {
  int **int_ptr = (int **) arg;      // [typo from int* -> int** corrected here]
  *int_ptr = calloc(1, sizeof(int)); // allocates space for 1 int
  **int_ptr = 10;                    // calloc: also zeroes memory
  return NULL;
}

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

The code unfortunately is buggy. What are the possible outcomes of this code? Assume the calls to `pthread_create()` and `calloc()` succeed, and that a NULL pointer dereference crashes reliably.
To answer: Fill in **A** if possible, **B** for not possible.

This code is racing to update result (in background_malloc()) and access the resulting value in the main thread. Various cases discussed below.

111. The code prints out 0 **A. Possible** *Trickiest case first: This happens if the background thread gets created, calls calloc(), but then the main thread runs again and prints the value of result. At that point, it is zero'd memory (thanks to calloc), and thus 0 is the output.*
112. The code prints out 10 **A. Possible** *This happens if, after the thread is created, the background thread runs to completion, thus setting result to 10 before the printf() in the main thread.*
113. The code prints out 100 **B. Not Possible** *result is never set to anything like 100.*
114. The code crashes **A. Possible** *This happens if the main thread inits result to NULL, creates the background thread, but then executes the next line (dereferencing result) before the background thread ever runs. Because it is NULL, crash!*
115. The code hangs forever **B. Not Possible** *Not possible: the code doesn't have anything that waits for anything.*

Problem XXIV: Here is some more multi-threaded code:

```
void *printer(void *arg) {
  char *p = (char *) arg;
  printf("%c", *p);
  return NULL;
}

int main(int argc, char *argv[]) {
  pthread_t p[5];
  for (int i = 0; i < 5; i++) {
    char *c = malloc(sizeof(char));
    *c = 'a' + i; // hint: 'a' + 1 = 'b', etc.
    pthread_create(&p[i], NULL, printer, (void *) c);
  }
  for (int i = 0; i < 5; i++)
    pthread_join(p[i], NULL);
  return 0;
}
```

Assuming calls to all library routines succeed, which of the following outputs are possible?
To answer: Fill in **A** if possible, **B** for not possible.

Each thread is created, and handed a unique argument with a letter in it: 'a', or 'b', or 'c', or 'd', or 'e'. The thread may run in any order, thus any output with one letter each, but in arbitrary order, is possible.

116. abcde **A. Possible**
117. edcba **A. Possible**
118. cccde **B. Not Possible**
119. eeeee **B. Not Possible**
120. aaaaa **B. Not Possible**

Problem XXV: Assume the same `printer()` function (from above), but this slightly changed `main()`:

```
void *printer(void *arg) {
  char *p = (char *) arg;
  printf("%c", *p);
  return NULL;
}

int main(int argc, char *argv[]) {
  pthread_t p[5];
  for (int i = 0; i < 5; i++) {
    char c = 'a' + i;
    pthread_create(&p[i], NULL, printer, (void *) &c);
  }
  for (int i = 0; i < 5; i++)
    pthread_join(p[i], NULL);
  return 0;
}
```

Assuming calls to all library routines succeed, which of the following outputs are possible?
To answer: Fill in **A** if possible, **B** for not possible.

The change here is interesting: now the threads all refer to the same stack location (of variable `c`), which is first set to 'a', then overwritten with 'b', etc. Thus, the order of when the thread runs and what the value of 'c' is at that moment is critical. Details below:

121. abcde **A. Possible** *Possible if, as each thread is created, it is run to completion, thus printing 'a', then 'b', etc.*
122. edcba **A. Possible** *Definitely the trickiest case. First thread is created, and runs, and is about to print 'a'. Just before printing it, it copies the value onto the stack while calling printf() (the dereference of p); however, before it prints, it is interrupted. Then the next thread gets created, and the same thing happens; it is about to print 'b', with a copy of the value placed on the stack calling into printf(), but doesn't yet print it. This continues until the last thread runs and prints 'e'. Then the fourth thread runs and prints 'd', etc.*
123. cccde **A. Possible** *Possible because value of `c` will get set to 'c' after three threads are created (but not yet run); if they then run, you get three prints of 'c'; then `c` is set to 'd' and the thread runs, printing 'd'; finally, `c` is set to 'e', and the thread runs and prints it.*
124. eeeee **A. Possible** *Possible because the value could get set to 'b', then 'c', then 'd', and then finally 'e' before any thread actually runs. In this case, all threads will print 'e'.*
125. aaaaa **B. Not Possible** *No way to get the value 'a' 5 times, because once the second thread is created, the value in the variable `c` must have been overwritten to 'b'.*

Problem XXVI: Assume the following multi-threaded memory allocator, roughly sketched out as follows:

```
int bytes_left = MAX_HEAP_SIZE;

pthread_cond_t c;
pthread_mutex_t m;

void *allocate(int size) {
  pthread_mutex_lock(&m);
  while (bytes_left < size)
    pthread_cond_wait(&c, &m);
  void *ptr = ...; // get mem from internal data structs
  bytes_left -= size;
  pthread_mutex_unlock(&m);
  return ptr;
}

void free(void *ptr, int size) {
  pthread_mutex_lock(&m);
  bytes_left += size;
  pthread_cond_signal(&c);
  pthread_mutex_unlock(&m);
}
```

Assume all of memory is used up (i.e., `bytes_left` is 0). Then:

- One thread (T1) calls `allocate(100)`
- Some time later, a second thread (T2) calls `allocate(1000)`
- Finally, some time later, a third thread (T3) calls `free(200)`

Assuming all calls to thread library functions work as expected, which of the following are possible just after this sequence of events has taken place?
To answer: Fill in **A** if possible, **B** for not possible.

The problem with this code example is that when free() signals a waiting thread, there is no guarantee it wakes "the right" thread, i.e., it may wake a thread that is waiting for too much memory. In this case, we have T1 waiting for 100 bytes, T2 waiting for 1000 bytes, and then T3 freeing only 200 bytes (not enough for T2 to succeed in allocation, but enough for T1). Thus:

126. T1 and T2 remain blocked inside `allocate()` **A. Possible** *Possible, because the signal may wake T2, which then rechecks its condition, and goes back to sleep, potentially forever.*
127. T1 becomes unblocked, gets 100 bytes allocated, and returns from `allocate()` **A. Possible** *If we're luck (instead), T1 gets awoken by T3, and then succeeds in its allocation request.*
128. T2 becomes unblocked, gets 1000 bytes allocated, and returns from `allocate()` **B. Not Possible** *There aren't 1000 bytes free, so this should not happen.*
129. T3 becomes blocked inside `free()` **B. Not Possible** *Not possible (really) because there is nothing to get permanently blocked upon. That said, the lock() acquisition could take a little while ...*
130. T1, T2, and T3 become deadlocked **B. Not Possible** *Only one lock here, so no deadlock can arise.*

Problem XXVII: A Semaphore is a useful synchronization primitive. Which of the following statements are true of semaphores?
To answer: Fill in **A** if true, **B** for not true.

131. Each semaphore has an integer value **A. True** *By definition, each semaphore has a value.*
132. If a semaphore is initialized to 1, it can be used as a lock **A. True** *This is called a binary semaphore.*
133. Semaphores can be initialized to values higher than 1 **A. True** *This is useful in some cases as described in the book.*
134. A single lock and condition variable can be used in tandem to implement a semaphore **A. True** *This is true, along with a state variable to track the value of the semaphore.*
135. Calling `sem_post()` may block, depending on the current value of the semaphore **B. False** *Only sem_wait() blocks, sem_post() just does its work and returns.*

Problem XXVIII: Here is the classic semaphore version of the producer/consumer problem:

```
void *producer(void *arg) {  // core of producer
  for (i = 0; i < num; i++) {
    sem_wait(&empty);
    sem_wait(&mutex);
    put(i);
    sem_post(&mutex);
    sem_post(&full);
  }
}

void *consumer(void *arg) {  // core of consumer
  while (!done) {
    sem_wait(&full);
    sem_wait(&mutex);
    int tmp = get(i);
    sem_post(&mutex);
    sem_post(&empty);
    // do something with tmp ...
  }
}
```

For the following statements about this working solution, which statements are true, and which are not?
To answer: Fill in **A** if true, **B** for not true.

This just reviews how the producer/consumer solution works, when semaphores are used.

136. The semaphore `full` must be initialized to 0 **A. True** *Full tracks how many full buffers there are; at the beginning, zero*
137. The semaphore `full` must be initialized to 1 **B. False** *As above, therefore this is wrong.*
138. The semaphore `empty` must be initialized to 1 **B. False** *Empty tracks how many empty buffers there are; if there are more than one, it should be initialized to the number of empty buffers. Thus, false.*
139. The semaphore `empty` can be initialized to 1 **A. True** *If there is only a size=1 buffer, it is OK to initialize empty this way.*
140. The semaphore `mutex` must be initialized to 1 **A. True** *Mutex provides mutual exclusion, and thus must be set to 1 at the beginning to work properly.*

Problem XXIX: One way to avoid deadlock is to schedule threads carefully. Assume the following characteristics of threads T1, T2, and T3:

- T1 (at some point) acquires and releases locks L1, L2
- T2 (at some point) acquires and releases locks L1, L3
- T3 (at some point) acquires and releases locks L3, L1, and L4

For which schedules below is deadlock possible?
To answer: Fill in **A** if deadlock is possible, **B** for not possible.

The key to this problem: do threads that CAN deadlock ever run at the same time? If so, they might deadlock. If not, it isn't. From the above, only T2 and T3 can deadlock, because they each grab two locks (L1 and L3) in some order.

141. T1 runs to completion, then T2 to completion, then T3 runs **B. Not Possible** *T2 and T3 do not run at the same time.*
142. T1 and T2 run concurrently to completion, then T3 runs **B. Not Possible** *T2 and T3 do not run at the same time.*
143. T1, T2, and T3 run concurrently **A. Possible** *T2 and T3 run at the same time.*
144. T3 runs to completion, then T1 and T2 run concurrently **B. Not Possible** *T2 and T3 do not run at the same time.*
145. T1 and T3 run concurrently to completion, then T2 runs **B. Not Possible** *T2 and T3 do not run at the same time.*

Problem XXX: The multi-level page table is something that cannot be avoided. No matter what you do, there it is, bringing joy and horror to us all. In this last question, you'll get your chance at a question about this foreboding structure. Fortunately, you don't have to perform a translation. Instead, just answer these true/false questions about the multi-level page table.
To answer: Fill in **A** if true, **B** for not true.

Answers below. You must be glad you didn't have to do a multi-level translation, no?

146. A multi-level page table may use more pages than a linear page table **A. True** *If a byte is valid on each page of the address space, the multi-level structure adds the overhead of multiple levels atop the linear structure.*
147. It's easier to allocate pages of the page table in a multi-level table (as compared to a linear page table) **A. True** *The linear table must be allocated contiguously; the multi-level page table is chopped into chunks. Thus, it is easier to allocate the multi-level page table.*
148. Multi-level page table lookups take longer than linear page table lookups **A. True** *Generally, true, because of the multiple levels.*
149. With larger virtual address spaces, usually more levels are used **A. True** *Generally true, if you want to fit each level into page-sized chunks.*
150. TLBs are useful in making multi-level page tables even smaller **B. False** *TLBs make translation faster, not page tables smaller.*