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.

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

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.

6. 012345678 ...
7. 012301234 ...
8. 012345670123 ...
9. 01234567891011 ...
10. 123456789 ...

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.

11. 0
12. 1000
13. 999
14. 998
15. 1002

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.

16. Running, Running, Running, Ready, Running, Running, Running, Ready
17. Embryo, Ready, Ready, Ready, Ready, Ready
18. Running, Running, Blocked, Blocked, Blocked, Running
19. Running, Running, Blocked, Blocked, Blocked, Ready, Running
20. Embryo, Running, Blocked, Running, Zombie, Running

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.

21. ab
22. abb
23. bab
24. bba
25. a

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.

26. ab
27. abb
28. bab
29. bba
30. a

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.

31. 123
32. 12
33. 2
34. 23
35. 3

Problem VIII: 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 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.

36. 123
37. 12
38. 2
39. 23
40. 3

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.

41. A=10, B=10, C=10
42. A=20, B=20, C=20
43. A=5, B=10, C=15
44. A=20, B=30, C=40
45. A=30, B=1, C=1

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
47. Round Robin
48. STCF (Shortest Time to Completion First)
49. Multi-level Feedback Queue
50. Lottery Scheduling

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
52. MLFQ starves long running jobs
53. MLFQ uses different length time slices for jobs
54. MLFQ uses round robin
55. MLFQ forgets what it has learned about running jobs sometimes

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.

56. 0
57. 1000
58. 10000
59. 10050
60. 10100

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.

61. 0
62. 1000
63. 10000
64. 10050
65. 10100

Problem XIV: Segmentation is a generalization of base-and-bounds. Which possible advantages does seg- mentation 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.

66. Faster translation
67. Less physical memory waste
68. Better sharing of code in memory
69. More hardware support needed to implement it
70. More OS issues to handle, such as compaction

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.
71. 0x1d (decimal: 29)
72. 0x7b (decimal: 123)
 73. 0x10 (decimal: 16)
74. 0x5a (decimal: 90)
75. 0x0a (decimal: 10)

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.

76. 0x34 (decimal: 52)
77. 0x44 (decimal: 68)
78. 0x57 (decimal: 87)
79. 0x18 (decimal: 24)
80. 0x46 (decimal: 70)

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.

81. 0, 100, 200, 1, 101, 201, ... (repeats in this pattern)
82. 0, 100, 200, 300, 0, 100, 200, 300, ... (repeats)
83. 0, 1000, 2000, 3000, 4000, 0, 1000, 2000, 3000, 4000, ... (repeats)
84. 0,1,2,3,4,5,6,0,1,2,3,4,5,6,... (repeats)
85. 300, 200, 100, 0, 300, 200, 100, 0, ... (repeats)

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.
87. The OPT (optimal) policy always performs at least as well 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.
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.
90. Random replacement is always worse than LRU replacement.

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.

91. 7
92. 9
93. 0
94. 4
95. 9

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.
96. 1
97. 2
98. 3
99. 4
100. 5

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.

101. ticket=0 and turn=0
102. ticket=0 and turn=1
103. ticket=1 and turn=0
104. ticket=16 and turn=5
105. ticket=1000 and turn=999

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.
106. 0
107. 1
108. 2
109. 3
110. 4

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;
  *int_ptr = calloc(1, sizeof(int)); // allocates space for 1 int
  **int_ptr = 10;

 return NULL;
}

// calloc: also zeroes memory

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.

111. The code prints out 0
112. The code prints out 10
113. The code prints out 100
114. The code crashes
115. The code hangs forever

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.

116. abcde
117. edcba
118. cccde
119. eeeee
120. aaaaa

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

```
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.

121. abcde
122. edcba
123. cccde
124. eeeee
125. aaaaa

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.

126. T1 and T2 remain blocked inside allocate()
127. T1 becomes unblocked, gets 100 bytes allocated, and returns from allocate()
128. T2 becomes unblocked, gets 1000 bytes allocated, and returns from allocate()
129. T3 becomes blocked inside free()
130. T1, T2, and T3 become deadlocked

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
132. If a semaphore is initialized to 1, it can be used as a lock
133. Semaphores can be initialized to values higher than 1
134. A single lock and condition variable can be used in tandem to implement a semaphore
135. Calling sem post() may block, depending on the current value of the semaphore

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.

136. The semaphore full must be initialized to 0
137. The semaphore full must be initialized to 1
138. The semaphore empty must be initialized to 1
139. The semaphore empty can be initialized to 1
140. The semaphore mutex must be initialized to 1

Problem XXIX: One way to avoid deadlock is to schedule threads carefully. Assume the following charac- teristics 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.

141. T1 runs to completion, then T2 to completion, then T3 runs
142. T1 and T2 run concurrently to completion, then T3 runs
143. T1, T2, and T3 run concurrently
144. T3 runs to completion, then T1 and T2 run concurrently
145. T1 and T3 run concurrently to completion, then T2 runs

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.

146. A multi-level page table may use more pages than a linear page table
147. It's easier to allocate pages of the page table in a multi-level table (as compared to a linear page table)
148. Multi-level page table lookups take longer than linear page table lookups
149. With larger virtual address spaces, usually more levels are used
150. TLBs are useful in making multi-level page tables even smaller