1. `git clone` can be used to clone the Pintos repository to a local working directory
2. `strcpy()` does not have a buffer bounds check and can cause an overflow of the destination buffer. Additionally it does not have well-defined behaviour for overlapping memory including the more likely case of `source==destination`.
3. Pintos is a uniprocessing threading system; that is execution only occurs on the primary logical CPU (`thread 0 core 0 processor 0`), any others remain in the Wait-For-Startup-IPI state.

   Kernel Threads are pre-emptible so Thread scheduling is achieved by periodically yielding this single chain of execution between threads based on timer interrupts.

   At Boot the main call graph runs:
   - `thread_start()` – initialise the threading system
   - `thread_create()` – create the idle thread
     - `thread_block()` – permanently yield this thread
       - `schedule()` – schedule the next thread

   Whenever `schedule()` is called:
   - get `next_thread_to_run()` – simply return head of ready list
   - `switch_threads()` – save previous thread's state and switch to new thread
   - `thread_schedule_tail()` – cleanup previous thread if necessary

   Once `schedule()` returns execution continues through the thread body itself.

   If a thread's body exits, `thread_exit()` is called:
   - set thread as dying so it's cleaned up
   - `schedule()` – will clean up this thread and switch to another

   Periodic Interrupt timer calls:
   - `thread_tick()` – updates tick counts
   - if thread must be pre-empted (based on ticks) then `thread_yield()`:
     - `list_push_back()` – current thread moved to end of ready list
     - `schedule()` – will switch to another thread

4. Reproducibility is the ability to consistently induce a certain execution state on separate executions of a given program with the same input. It is weaker than complete determinism in that we are requiring the system's execution of the program to be deterministic only with respect to reaching this certain execution state of interest.

   In Pintos this means that given a certain reached program state of interest (contents of memory and registers) from a given input yielding a certain output, all subsequent executions on that same input reach the same program state and yield the same output. In particular this means that stack traces will also be equivalent (to a degree taking into account randomised algorithms etc.). There is still non-determinism between sets of intermediate steps yielding the same final state arising from e.g. equivalent paths through a non-deterministic algorithm or CPU microcode execution branching.

   In the case of debugging we are usually interested in reproducing certain erroneous behaviour which is isomorphic to reaching the associated erroneous state. If the property of reproducibility does not

hold then we have no guarantees whether given the same input we can ever reach this state and thus reproduce the behaviour, hampering debugging.

Furthermore there may be race conditions and other bugs related to non-determinism that will only be evident in conditions not encountered while debugging or testing e.g. race conditions that only occur on a particular production system. Hence given lack of reproducibility, the passing of a test can make no guarantees about whether whatever property the test was designed to establish will actually consistently hold on the system.

5. Print an unsigned 64bit integer:
   `unsigned long long x=1234; printf("%llu", x);`
6. Locks are actually implemented using the Semaphore implementation which works by disabling pre-emption then forcing a `thread_block()` and `thread_unblock()` on the appropriate threads. Locks are a Semaphore with an initial value of 1 but with the added restrictions that only the thread owning the lock can release it.
7. The `struct thread` cannot exceed cannot exceed the size of the 4kB page it is stored in minus the size of three kernel stack frames (the thread stack depth when the body is initially executing). This limit is smaller depending on whether the thread itself makes function calls and how deep they are. If the stack overflows most likely the assertion `is_thread()` in `thread_current()` will fail because the magic number at the start of the `struct thread` will have been overwritten. This is far less computationally expensive than actually tracking the stack size and means that stack overflow is detected causing the correct fatal error in all but specifically contrived situations.
8. The test execution Pintos standard output is piped to `alarm-multiple.out` in `src/threads/build/tests/threads/` with error output in `alarm-multiple.err` (stderr from Pintos and/or errors from QEMU). The actual result of the Perl test program on this output is printed when the Makefile is run and stored in `alarm-multiple.result` (unless there was an error with the Perl test program itself).
9. Given the following function:
```
static bool bar_is_less (const struct list_elem *a_, const struct list_elem
*b_, void *aux AUX)
{
  const struct foo *a = list_entry (a_, struct foo, e);
  const struct foo *b = list_entry (b_, struct foo, e);

  return a->bar < b->bar;
}
```
   The code to insert element `struct foo` into the ordered `struct list foo_list`:
   `list_insert_ordered(&foo_list, &foo, bar_is_less, NULL);`
10. Function to return a pointer to the `struct foo` in `list foo_list` with `bar` value `x`:
```
struct foo *get_bar_eq (struct list *foo_list, int x) {
  struct list_elem *elem;
  struct foo *cur;

  for (elem = list_begin (foo_list); elem != list_end (foo_list);
       elem = list_next (elem)) {
      cur = list_entry (elem, struct foo, e);
      if (cur->bar == x) return cur;
  }
  return NULL;
}
```