Q1

(a) Outline what is meant by a zombie process in a Unix like system.
A zombie process is a process that has finished execution but remains on the process table

(b) How does the zombie process ‘disappear’?
zombie processes are removed from the process table when their exit() status is read by the parent process.

(c) How is a zombie process different from an orphaned process?
an orphaned process is a process whose parent no longer exists. zombie processes have always finished execution and may still have a parent whereas orphaned processes have no parent and are still executing.

Q2

In lecture 5, slide 5 we see a web server implemented using threads - the model consists of one dispatcher thread and a variable number of worker threads that service each incoming HTTP request. Assume that each incoming request results in an access to the file system in order to return the requested html file in the HTTP response. The writer of the web server is really keen to utilise user threads as they are very fast and have little overhead.
(a) What is the major shortcoming of user threads in the outlined scenario?
While any thread is blocked on a system call to the file system the entire process will be blocked.

(b) What is the resultant impact on the performance of the web server?
The server will respond slower if it receives requests to close together.

(c) How could our OS be changed to potentially solve this shortcoming (hint, the answer is NOT to use kernel threads!).

Q3

(a) What is busy waiting?
When busy waiting a process repeatedly checks if a condition has been met before continuing execution.

(b) What is the primary advantage of busy waiting?
Busy waiting allows the system to avoid the overhead of a wait queue and context switching to, then back from a different process during the wait.

(c) When is busy waiting a good idea?
When wait times are small or system memory is more limited than cpu cycles.

Q4

Why can’t we use interrupts to enforce mutual exclusion on a multi-cpu system?

In a multi cpu system processes can access resources used by processes on different cpus without interrupts. Thus an additional method of ensuring exclusive control is required in critical sections.

Q5

(a) Consider a system with 12 instances of a particular resource type and four processes: P0, P1, P2 and P3. The table below shows the maximum needs and the currently allocated resource instances for each process.

Process	Maximum	Allocated	needed
P0	9	5	4
P1	4	1	3
P2	8	2	6
P3	5	1	4
free		3	

State if the system is in a safe state and justify your answer.

Process	needed
P0	4
P1	3
P2	6
P3	4
Free	3

1. P1 is the only process that doesn’t exceed the available resources so it runs leaving 4 free.
2. P0 or P3 could run, therefore P0 executes leaving 9 free.
3. P2 or P3 could run. P2 executes leaving 11 free.
4. P3 executes leaving all 12 resources free.

All the processes can run so the system is in a safe state.

Q6

Write a pseudo code implementation of semaphores P() and V() using the atomic swap instruction instead of turning off interrupts. Your solution should avoid busy waiting as much as possible (you can spin on the swap but not the semaphore) - descriptive text/comments in your code to outline the operations are perfectly acceptable. Hint: look at the code in class for swap and for the pintos Semaphores.

```
void p(struct semaphore *sema){
   ASSERT (sema != NULL);

   bool key = true;

   while (sema->value == 0)
         {
          while(key){
                swap(lock, key);
                list_push_back (&sema->waiters, &thread_current ()->elem);
                thread_block ();
                key = false;
          }
         }
   sema->value--;
}

void v(struct semaphore *sema){
   ASSERT (sema != NULL);

   bool key = true;

   while(key){
         swap(lock, key);
         if (!list_empty (&sema->waiters))
                thread_unblock (list_entry (list_pop_front (&sema->waiters), struct thread,
elem));
         sema->value++;
         lock = false;
   }
}
```