Practice Midterm Exam Solutions

1. Short Answer

Part A: Eliminating Inodes

- Advantage: fewer distinct disk blocks would need to be accessed when opening files, which could result in better performance.
- Disadvantage: hard links would be difficult or impossible to implement (it would require duplicating inode information each of the directory entries and maintaining consistency between these copies).

Part B: Log Issues

The problem is that the append operation isn't idempotent: if the operation is done multiple times, it will produce a different result than if it is done only once. This is a problem because it's possible that the operation has already been reflected on disk at the time of a crash, so replaying the log may perform the operation a second time. In general we don't know whether operations have already been performed or not when a crash occurs, so it's important that all operations in the log are idempotent.

Part C: Doubly-Indirect Indexing

- Advantage: supports even larger files than original design.
- Disadvantage: accessing payload for very small files requires many disk accesses

Part D: Free List Designs

- Bitmap advantage: contiguous on disk means access with fewer seeks
- Linked list advantage: minimal additional space needed since we are reusing space in free blocks

Part E: Inode Table

One block of inodes can lead to 2^5 files, each with at least one block. 1 out of every 2^5 blocks must store inodes. That's ~3%. (Arguments involving 16 bytes of **`dirent`** structure are also good, but remember that answer only needed to be approximate.)

Part F: Rename

Find the inode number of the file being moved by iteratively drilling toward it as **`pathname_lookup`** would. Remove file's **`dirent`** from parent directory payload, and then drill toward new parent directory of second argument, creating new **`dirent`**'s along the way as needed. Finally, append new **`dirent`** on behalf of new name, with new name (e.g. **`index-w19.html`**) and existing inode number.

Part G: Crash Recovery Tradeoffs

Generally, if we want to improve durability and consistency, we must sacrifice performance in order to perform additional operations / write more aggressively to avoid data loss. As an example, the block cache may have delayed writes of about 30 seconds to improve performance, but this means we may lose the last

30sec of data. If we wrote immediately, this would improve crash recovery, but at the expense of performance. (Another example is data logging – logging doesn't usually support payload data due to the amount of data that would log, thus impacting performance, but it means that we don't log payload changes so that data may be lost. If we did log payload data, we would have better crash recovery but the logging operations would be much more intensive and affect system performance.)

Part H: Multithreading

Three possibilities (there are more, these are just 3):

50 (if thread 1 runs to completion, then thread 2 runs to completion)

40 (if thread 2 runs to completion, then thread 1 runs to completion)

60 (if thread 1 increments, then thread 2 increments, then thread 1 multiplies, then thread 2 multiplies)

2. Duet

Sample Solution

```
static void duet(int incoming, char *one[], char *two[], int outgoing) {
    pid_t pids[2];
    int fds[2];
    pipe(fds);
    pids[0] = fork();
    if (pids[0] == 0) {
        close(fds[0]);
        close(outgoing);
        dup2(incoming, STDIN_FILENO);
        close(incoming);
        dup2(fds[1], STDOUT_FILENO);
        close(fds[1]);
        execvp(one[0], one);
    }

    close(incoming);
    close(fds[1]);
    pids[1] = fork();
    if (pids[1] == 0) {
        dup2(fds[0], STDIN_FILENO);
        close(fds[0]);
        dup2(outgoing, STDOUT_FILENO);
        close(outgoing);
        execvp(two[0], two);
    }

    close(outgoing);
    close(fds[0]);
    waitpid(pids[0], NULL, 0);
    waitpid(pids[1], NULL, 0);
}
```

3. Concurrent Evaluation

Sample Solution

```
typedef struct threadInfo {
  // Part A
  mutex m;
  condition_variable_any cv;
  size_t count;
} threadInfo;

static void evaluate(Expression& exp, threadInfo& info) {
  // Part C
  exp.evaluate();
  info.m.lock();
  info.count--;
  if (info.count == 0) info.cv.notify_all();
  info.m.unlock();
}

static bool concurrentAnd(const vector<Expression>& expressions) {
  threadInfo info;

  // Part B
  info.count = expressions.size();

  for (size_t i = 0; i < expressions.size(); i++) {
    thread(evaluate, ref(expressions[i]), ref(info));
  }

  // Part D
  info.m.lock();
  while (info.count > 0) {
    info.cv.wait(info.m);
  }
  info.m.unlock();
}
```

4. Multiprocessing

Part A: Possible Outputs

Possible Output 1: 112265

Possible Output 2: 121265

Possible Output 3: 122165

Part B: Close

The **`close(sp.infd)`** within the test program can only indicate the end of input that **`sort`** must process if all other references to it are closed. That doesn't happen unless we close it everywhere it's needed. If we don't, sort will continue waiting for more input forever thinking that more could come.

Part C: Thyme

Sample Solution

```
int main(int argc, char *argv[]) {
      struct timespec start;
      clock_gettime(CLOCK_REALTIME, &start);
      pid_t pid = fork();
      if (pid == 0) execvp(argv[1], argv + 1);
      waitpid(pid, NULL, 0);
      struct timespec finish;
      clock_gettime(CLOCK_REALTIME, &finish);
      print_elapsed_time(&start, &finish);
      return 0;
}
```