Name: **SOLUTION** UID#: DirectoryID:

CMSC216: Practice Exam 3A SOLUTION

Spring 2026, University of Maryland

Exam period: 20 minutes
Points available: 40

Problem 1 (10 pts): Examine the code to the right and describe what you expect its output to be. Explain why or why not you would expect to see any specific ordering in the output of the program.

SOLUTION: The processes will fork out in a "line" or "list" rather than branching in a tree. This is because each parent process falls into the `if()` *consequence and will break out of the loop so will have only a single child. The order of output will start with the last child in the last iteration and proceed backwards to the original parent due to the placement of* `wait()`*. Example:*

```
> a.out
iter 4, 13374 from 13373
iter 3, 13373 from 13372
iter 2, 13372 from 13371
iter 1, 13371 from 13370
iter 0, 13370 from 26018
```

```
#include "headers.h"
int main(){
  for(int i=0; i<5; i++){
    pid_t p = fork();
    if(p  != 0){
      wait(NULL);
      printf("iter %d, %d from %d\n",
             i,getpid(),getppid());
      fflush(stdout); // output now
      break;          // quit loop
    }
  }
  exit(0);
}
```

Problem 2 (10 pts): Separa Tememory is trying to understand how child and parent processes interact and is baffled by the behavior of the nearby `childchange.c` program. It seems to her that on changing the `life` variable, it should print a different value later on. Explain to Separa:

(A) Why the program prints what it does
(B) How parent and child processes can affect one another.

SOLUTION: The parent and child processes are created with separate memory images. The child process will get a copy of the parents memory [excepting the return value of `fork()`*] but these values are distinct and disconnected from the parent values. Changes made by the child to its memory do not affect the parent memory and vice versa. In short, (A) the parent prints* `42` *because the parent makes no changes to the* `life` *variable and the changes made by the child process to its* `life` *variable do not affect the parent and (B) cannot affect the parent due to processes defaulting to having their own memory images.*

```
// childchange.c
#include "headers.h"
int main(){
  int life = 42;
  pid_t pid = fork();
  if(pid == 0){
    life--;
    return 0;
  }
  else{
    wait(NULL);
  }
  printf("life: %d\n",
         life);
  return 0;
}
// >> gcc childchange.c
// >> ./a.out
// life: 42
//   Why not 41??
```

Problem 3 (15 pts): Nearby is a matrix/vector function which performs poorly. Create a new version of this function that **optimizes the memory access pattern**. Show your code and give a brief description of why the changes you made should improve performance.

```
int subcol_BASE(matrix_t mat, vector_t vec) {
  for(int j=0; j<mat.cols; j++){
    for(int i=0; i<mat.rows; i++){
      int elij = MGET(mat,i,j);
      int veci = VGET(vec,i);
      elij -= veci;
      MSET(mat,i,j,elij);
    }
  }
  return 0;
}
```

```
//////// SOLUTION ////////
int subcol_opt(matrix_t mat,
               vector_t vec)
{
  if(mat.rows != vec.len){
    printf("mat.rows (%ld) != vec.len (%ld)\n",
           mat.rows,vec.len);
    return 1;
  }
// Loop over rows
  for(int i=0; i<mat.rows; i++){
    // subtract same vec el each time
    int veci = VGET(vec,i);
    // across row
    for(int j=0; j<mat.cols; j++){
      int elij = MGET(mat,i,j);
      elij -= veci;
      MSET(mat,i,j,elij);
    } // end INNER LOOP across row
  } // end OUTER LOOP over rows
  return 0;
}
```

WHY CHANGES IMPROVE PERFORMANCE:
SOLUTION: The new version favors cache by visiting matrix elements across rows rather than down columns. This eliminates the memory stride that comes from traversing down columns. This will improve speed as adjacent row elements are adjacent in memory so will come together into cache which will allow faster operations on those memory elements.

Problem 4 (5 pts): Consider the code sample nearby which prints logging messages to either the screen or a log file as dictated by the `USE_LOGFILE` variable. Describe how one could eliminate the conditional `if/else` and all the `fprintf()` calls using **I/O redirection system calls** within the program.

SOLUTION: If `USE_LOGFILE` *is true, do the falling. Call* `dup()` *system call to duplicate* `STDOUT_FILENNO` *then* `dup2(logfd,STDOUT_FILENO)` *so that anything that is printed to the screen instead goes into the log file. One would need to* `open()` *the log file to get a File Descriptor for it and possible restore Standard Output later, but this line of attack would mean only unconditional* `printf()` *calls are needed.*

```
{
  if(USE_LOGFILE){
    fprintf(logfile,"Updating DB\n");
  }
  else{
    printf("Updating DB\n");
  }
  update_db();
  if(USE_LOGFILE){
    fprintf(logfile,"Syncing files\n");
  }
  else{
    printf("Syncing files\n");
  }
  file_sync();
  ...;
}
```