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

CMSC216: Practice Final Exam B SOLUTION

Spring 2024
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
```

```c
// #include "headers.h"
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <wait.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
      break;          // immediately
    }
  }
  exit(0);
}
```

Problem 2 (10 pts): Nearby is the output of `pmap` showing page table virtual memory mapping information for a running program called `memory_parts`. Answer the following questions about this output.

(A) Certain addresses of memory are marked with the annotation `r-x`. Explain what this means and what kind of information you would expect to find in those addresses.
SOLUTION: The annotation means "read and execute" with no write permission. Typically this is a page of memory that would contain program text: executable instructions that should not be changed but can be fed to the processor to run the program. Examples are in the ***`memory_parts`*** *program itself for its* ***`main()`*** *and in the shared library* ***`libc`*** *which has instructions for* ***`printf()`*** *and the like.*

(B) Why does `pmap` only show a limited number of virtual addresses? What would happen if the program attempted to access an address not listed in the output? Example: address `0x00` is not in the listing.
SOLUTION: The page table only contains mapped pages for program. These mapped addresses are what is shown. The large number of other addresses are unmapped. Attempting to access these unmapped addresses will result in errors such as ***`segmentation faults`****; this usually causes the program to be immediately terminated.*

```
> pmap 7986
7986:   ./memory_parts
00005579a4abd000      4K r-x-- memory_parts
00005579a4cbd000      4K r---- memory_parts
00005579a4cbe000      4K rw--- memory_parts
00005579a4cbf000      4K rw---   [ anon ]
00005579a53aa000    132K rw---   [ heap ]
00007f441f2e1000   1720K r-x-- libc-2.26.so
00007f441f48f000   2044K ----- libc-2.26.so
00007f441f68e000     16K r---- libc-2.26.so
00007f441f692000      8K rw--- libc-2.26.so
00007f441f694000     16K rw---   [ anon ]
00007f441f698000    148K r-x-- ld-2.26.so
00007f441f88f000      8K rw---   [ anon ]
00007f441f8bb000      4K r---- gettysburg.txt
00007f441f8bc000      4K r---- ld-2.26.so
00007f441f8bd000      4K rw--- ld-2.26.so
00007f441f8be000      4K rw---   [ anon ]
00007fff96ae1000    132K rw---   [ stack ]
00007fff96b48000     12K r----   [ anon ]
00007fff96b4b000      8K r-x--   [ anon ]
 total             4276K
```

Problem 3 (10 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 and will improve speed.

Problem 4 (5 pts): To further optimize the `subcol_opt()` function, a common strategy is to utilize multiple threads. Describe briefly how this might be done. Include in your answer.

(A) How the work to be done is divided among threads
(B) How changes to shared data will be coordinated to ensure safety.

SOLUTION: Set up a "worker" function where each thread would select some rows to work on. Working across rows as in the optimized version will continue to run fast. (A) Each thread subtracts elements from `vec` *away from corresponding elements in its assigned rows of the matrix: 2 threads operation on a matrix with 100 rows would have thread 0 subtract off of rows 0-49 and thread 1 from 50-99. A main thread could pass logical thread ID numbers, thread counts, and pointers to the matrix/vector via a "context" struct. (B) Even though the Matrix is shared, threads will NOT need to coordinate with a mutex in this case as they are each changing different elements in the matrix: thread 0 and thread 1 will never alter the same elements.*

Problem 5 (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();
  ...;
}
```