CSCI 4210 — Operating Systems
CSCI 6140 — Computer Operating Systems
Sample Final Exam Questions (document version 1.1)

Overview

- The final exam will be on **Monday, December 21, 2015** from 11:30AM-2:30PM (please arrive early).
- The final exam will be 180 minutes; therefore, if you have 50% additional time, you will have 270 minutes (e.g., 11:30AM-4:00PM); if you have 100% additional time, you will have 360 minutes (e.g., 11:30AM-5:30PM).
- The final exam will count as 21% of your final course grade.
- You may bring two double-sided (or four single-sided) 8.5"x11" crib sheets containing anything you would like; crib sheets will not be collected.
- Final exams will **not** be handed back or available per review; however, exams will be graded multiple times to ensure correctness and consistency of grading.
- Final grades will be made available in LMS.
- The final exam will be comprehensive, though the following topics will **not** be on the exam: Network Programming; I/O and disk access topics; and Queuing Theory.

Sample Midterm Exam Questions

1. Given the following C program, what is the **exact** output? If multiple outputs are possible, succinctly describe all possibilities. Assume that all system calls complete successfully. Further, assume that the parent process ID is `168`, with any child processes numbered `768`, `769`, `770`, etc.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{
  int rc;
  printf( "ONE\n" );
  rc = fork();
  printf( "TWO\n" );
  if ( rc == 0 ) { printf( "THREE\n" ); }
  if ( rc > 0 ) { printf( "FOUR\n" ); }
  return EXIT_SUCCESS;
}
```

2. Given the following C program, what is the **exact** output? If multiple outputs are possible, succinctly describe all possibilities. Assume that all system calls complete successfully. Further, assume that the parent process ID is `168`, with any child processes numbered `768`, `769`, `770`, etc.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main()
{
  int x = 150;
  printf( "PARENT: x is %d\n", x );

  printf( "PARENT: forking...\n" );
  pid_t pid = fork();
  printf( "PARENT: forked...\n" );

  if ( pid == 0 )
  {
    printf( "CHILD: happy birthday\n" );
    x *= 2;
    printf( "CHILD: %d\n", x );
  }
  else
  {
    wait( NULL );

    printf( "PARENT: child completed\n" );
    x *= 2;
    printf( "PARENT: %d\n", x );
  }

  return EXIT_SUCCESS;
}
```

How would the output change if the `wait()` system call was removed from the code?

3. Given the following C program, what is the **exact** output? If multiple outputs are possible, succinctly describe all possibilities. Assume that all system calls complete successfully.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{
  printf( "ONE\n" );
  fprintf( stderr, "ERROR: ONE\n" );
  int rc = close( 1 );
  printf( "==> %d\n", rc );

  printf( "TWO\n" );
  fprintf( stderr, "ERROR: TWO\n" );
  rc = dup2( 2, 1 );
  printf( "==> %d\n", rc );

  printf( "THREE\n" );
  fprintf( stderr, "ERROR: THREE\n" );
  rc = close( 2 );
  printf( "==> %d\n", rc );

  printf( "FOUR\n" );
  fprintf( stderr, "ERROR: FOUR\n" );

  return EXIT_SUCCESS;
}
```

4. Given the following C program, what is the **exact** output? Further, what is the **exact** contents of the `newfile.txt` file? If multiple outputs are possible, succinctly describe all possibilities. Assume that all system calls complete successfully.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

int main()
{
  int fd;
  close( 2 );

#if 0
  close( 1 );  /* <== add this line later.... */
#endif

  printf( "HI\n" );
  fflush( stdout );

  fd = open( "newfile.txt", O_WRONLY | O_CREAT | O_TRUNC, 0600 );

  printf( "==> %d\n", fd );
  printf( "WHAT?\n" );
  fprintf( stderr, "ERROR\n" );

  close( fd );

  return EXIT_SUCCESS;
}
```

How do the output and file contents change if the second `close()` system call is uncommented?

5. Given the following C program, what is the **exact** output? Further, what is the **exact** contents of the `newfile.txt` file? If multiple outputs are possible, succinctly describe all possibilities. Assume that all system calls complete successfully. Further, assume that the parent process ID is `168`, with any child processes numbered `768`, `769`, `770`, etc.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/wait.h>

int main()
{
  int fd;
  close( 2 );

#if 0
  close( 1 );  /* <== add this line later.... */
#endif

  printf( "HI\n" );
  fflush( stdout );

  fd = open( "newfile.txt", O_WRONLY | O_CREAT | O_TRUNC, 0600 );

  printf( "==> %d\n", fd );
  printf( "WHAT?\n" );
  fprintf( stderr, "ERROR\n" );
  fflush( stdout );

  int rc = fork();
  if ( rc == 0 )
  {
    printf( "AGAIN?\n" );
    fprintf( stderr, "ERROR ERROR\n" );
  }
  else
  {
    wait( NULL );
  }

  printf( "BYE\n" );
  fprintf( stderr, "HELLO\n" );

  return EXIT_SUCCESS;
}
```

How do the output and file contents change if the second `close()` system call is uncommented?

6. Given the following C program, what is the **exact** output? If multiple outputs are possible, succinctly describe all possibilities. Assume that all system calls complete successfully. Further, assume that the parent process ID is `168`, with any child processes numbered `768`, `769`, `770`, etc.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{
  int rc;
  int p[2];
  rc = pipe( p );
  printf( "%d %d %d\n", getpid(), p[0], p[1] );

  rc = fork();

  if ( rc == 0 )
  {
    rc = write( p[1], "ABCDEFGHIJKLMNOPQRSTUVWXYZ", 26 );
  }

  if ( rc > 0 )
  {
    char buffer[40];
    rc = read( p[0], buffer, 8 );
    buffer[rc] = '\0';
    printf( "%d %s\n", getpid(), buffer );
  }

  printf( "BYE\n" );

  return EXIT_SUCCESS;
}
```

7. Given the table of process burst times, arrival times, and priorities shown below, calculate the turnaround time, wait time, and number of preemptions for each process using the following scheduling algorithms: FCFS; SJF; SRT; RR with a time slice of `3` ms; and Priority scheduling with SJF as secondary scheduling algorithm (note that the lower the priority value, the higher priority the process has).

PROCESS	BURST TIME	ARRIVAL TIME	PRIORITY
P1	7 ms	0 ms	2
P2	5 ms	0 ms	3
P3	5 ms	1 ms	1
P4	6 ms	4 ms	2

8. Write pseudocode for the scheduling algorithms above; for each, determine what the computational complexity of adding a process to the ready queue, of selecting the next process, and of removing a process from the queue.

9. For each scheduling algorithm above, describe whether CPU-bound or I/O-bound processes are favored.

10. For each scheduling algorithm above, describe how starvation can occur (or describe why it never will occur).

11. Given actual burst times for process P shown below and an initial guess of `6` ms, calculate the estimated next burst time for P by using exponential averaging with alpha (α) set to `0.5`; recalculate with α set to `0.25` and `0.75`.

    Measured burst times for P are: `12` ms; `8` ms; `8` ms; `9` ms; and `4` ms.

12. Describe what happens when a child process terminates; also, what happens when a process that has child processes terminates?

13. In a shell, how might pipe functionality actually be implemented? Describe in detail.

14. Why might `fork()` fail? Why does a "fork-bomb" cause a system to potentially crash and how can a "fork-bomb" be avoided by an operating system?

Sample Final Exam Questions

15. Given the following C program, what is the **exact** output? If multiple outputs are possible, succinctly describe all possibilities. Assume that all system calls and pthread library calls complete successfully. Further, assume that the main thread ID is `1984`, with any child threads numbered `100`, `101`, `102`, etc.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

void * action( void * arg )
{
  int t = *(int *)arg;
  printf( "THREAD %u: I'm alive %d\n", (unsigned int)pthread_self(), t );
  sleep( t );
  return NULL;
}

int main()
{
  pthread_t tid[4];
  int i, t;

  for ( i = 0 ; i < 4 ; i++ )
  {
    t = 1 + i * 3;
    printf( "MAIN: Creating thread for task %d\n", t );
    pthread_create( &tid[i], NULL, action, &t );
  }

  for ( i = 0 ; i < 4 ; i++ )
  {
    pthread_join( tid[i], NULL );
    printf( "MAIN: Joined child thread %u\n", (unsigned int)tid[i] );
  }

  return 0;
}
```

16. What is the behavior of the given code if the second `for` loop (i.e., the loop that includes the `pthread_join()` call) is entirely removed?

17. Describe how the inconsistencies in the child threads (i.e., in terms of their inputs) would be corrected in the above code.

18. **Contiguous Memory Allocation:**

    Consider a contiguous memory allocation scheme with dynamic partitioning for a 64MB physical memory with five pre-allocated processes (i.e., `A`, `B`, `C`, `D`, and `E`) that just happen to have memory requirements that are evenly divisible by 1MB.

    Given new processes `F`, `G`, and `H`) that arrive (almost) simultaneously in the order shown below, show how memory allocation occurs for each of the given placement algorithms.

Arrival Order	ProcessID	Memory Requirements
1	F	2,987,642 bytes
2	G	4,002,016 bytes
3	H	6,202,454 bytes

    Note that if a process cannot be placed, be sure to state that, then move on to the next process. Do **not** perform defragmentation.

    - For the memory allocation shown below, apply the **First Fit** algorithm:

      ```
      Memory:
      AAAAAAAAA......B   (each character here represents 1MB)
      BBBBBB.....CCCCC
      CCCC...DDDDDDDDD
      DDDD.........EEE
      ```

    - For the memory allocation shown below, apply the **Best Fit** algorithm:

      ```
      Memory:
      AAAABBBBBBBBBBB
      BBBBBBBB....CCCC
      CCCCC...DDDDDDDD
      DDDDDD.......EEE
      ```

    - For the memory allocation shown below, apply the **Next Fit** algorithm, with process `D` being the last-placed process:

      ```
      Memory:
      AAAAAAAAAA.....B
      BBBBBBBB.....CCC
      CCCCCC...DDDDDDD
      DDDD........EEEE
      ```

19. For each of the above algorithms, how much space is unused after the processes are allocated to memory?

20. For each of the above, what kind of fragmentation occurs (i.e., internal or external)?

21. **Noncontiguous Memory Allocation:**

    Consider a noncontiguous memory allocation scheme in which a logical memory address is represented using 32 bits. Of these bits, the high-order 12 bits represent the page number; the remaining bits represent the page offset.

    - What is the total logical memory space (i.e., how many bytes are addressed)?
    - How many pages are there?
    - What is the page size?
    - What is the frame size?
    - How does logical memory address 23,942,519 (binary 10110110101010101110111) map to physical memory (i.e., what is the logical page number and page offset)?
    - If a process requires 78,901,234 bytes of memory, how many pages will it require?
    - How many bytes are unused due to external fragmentation?
    - How many bytes are unused due to internal fragmentation?
    - Given that the page table is stored entirely in memory and a memory reference takes 100 nanoseconds, how long does a paged memory reference take?
    - Adding a translation look-aside buffer (TLB) with a TLB access time of 15 nanoseconds, how long does a paged memory reference take if a TLB hit occurs?
    - Given a TLB hit ratio of 84%, what is the effective memory access time (EMAT)?

22. **Virtual Memory:**

    Given a page reference string and an n-frame memory, how many page faults occur for the following page replacement algorithms: FIFO; OPT; LRU; LFU.

```
Page Reference Stream:
  1 8 4 8 4 3 8 3 4 7 1 7 2 3 2 4 2 7 8
```

23. Given a page reference string and a working set delta, identify the working set at the point indicated below.

```
Page Reference Stream:
  1 8 4 8 4 3 8 3 4 7 1 7 2 3 2 4 2 7 8
                    ^
                    |
```

    Use a delta of 3, 5, then 8.