CS252 Systems Programming

Midterm Exam Preparation

Please write by hand the answers and turn them in the day of the exam. The solutions will be posted the day before the exam. I suggest you try to solve the questions before looking at the solution.

Murtadha Aljubran

Part 1. True False Questions

Answer True/False (T/F) (1 point each)

F ELF stands for Executable Link ~~Format~~ File.
T A process has a stack for every thread it contains.
T A process ID may be reused
T Race conditons are more likely to happen in multiprocessor machines.
F In POSIX threads, the thread calling fork will create a child process that is a copy of the parent inclluding all the threads in the process. Only in Solaris and by calling thr-create

Part 2. Short questions.

2. Write down the fields of an i-node
- Flag/Mode : Read, Write, Execute for Owner, Group, All.
- Owner : User ID, Group ID.
- Time Stamps : Creation Time, Access Time, Modification Time.
- Size : File Size in Kilobytes.
- Ref. Count: Number of times i-node appears in the directory (delete when Ref. Count is 0).

3. Explain what is deadlock and what is starvation and give an example of each.

Deadlock occurs when one or more threads are waiting on a resource that will never be available, and thus will block forever.
-Example: Multiple threads needing to lock Mutexes controlled by each other (as in Question 6, and the Philosopher Spaghetti problem.

Starvation occurs when a thread needs to wait a very long time to get a given resource.
- Example: Read and Write Locks. Writers may not write if the number of Readers is greater than 0.

4. What are the advantages and disadvantages of using threads vs. using processes?

Advantages of threads: fast thread creation, fast context switch, fast communication across threads.
Disadvantages of threads/Advantages of Processes: threads are less robust than processes, threads have more synchronization problems than processes.

5. Mention three important files that are stored in /etc in UNIX.

/etc/passwd - User information
/etc/groups - Group information
/etc/inetd.conf - Configuration of Internet Services (deamons)

6. In the following code,

T1:	T2:	T3:
a) mutex_lock(&m3);	d) mutex_lock(&m2);	g)mutex_lock(&m4);
b) mutex_lock(&m5);	e)mutex_lock(&m5);	h)mutex_lock(&m3);
c) mutex_lock(&m1);	f)mutex_lock(&m4);	i)mutex_lock(&m1);

a) give a sequence of steps using the letters at the left of each statement that will cause the threads to get into a deadlock. (Example: T1a, T2d, T3g ... etc)

T1 a, T2 d, T2 e, T3 g

b) Draw the graph representation of the deadlock.

$$\begin{tikzcd}
m5 \arrow[r] & \boxed{T_2} \arrow[r] & m4 \arrow[d] \\
\boxed{T1} \arrow[u] & & \boxed{T3} \arrow[dl] \\
& m3 \arrow[ul] &
\end{tikzcd}$$

c) Rewrite the code to prevent the deadlock.

T1:	T2:	T3:
a) mutex-lock (& m1);	d) mutex-Lock(& m2);	g) mutex-lock (& m1);
b) mutex-lock (& m3);	e) mutex-Lock(& m4);	h) mutex-Lock(& m3);
c) mutex-lock(& m5);	f) mutex-Lock(& m5);	i) mutex-lock (& m4);

7.Assume that the following code for adding items into an array. addToArray will return the index in the array where the value was added or -1 if the buffer is full. a) What problems do you see can happen if multiple threads try to run it? Rewrite the code to solve the problem.

```
#define MAXCOUNT;
int count = 0;
int array[MAXCOUNT];

int addToArray(int value)
{
  if (count == MAXCOUNT) {
     return -1;
  }

  array[count]=value;

  count = count + 1;

  return count-1;
}
```

New Code

```
#include <pthread.h>
#define MAXCOUNT;
int count=0;
int array[MAXCOUNT];
pthread_mutex_t mutex;
int addToArray(int value)
{
  if(count == MAXCOUNT){
    return -1;
  }
  pthread_mutex_lock(&mutex);
  array[count]=value;
  count = count+1;
  pthread_mutex_unlock(&mutex);
  return count-1;
}
```

There could be a context switch between array[count]=value; and count=count+1; causing what was just put into array[count] to be written over.

8. A program calls the system call write(file_descriptor, buffer, nbytes). Explain step by step the sequence of events, the checks, and the interrupts in user mode and kernel mode that happen from the time write() is called until it returns.

1. User Program calls write(file_descriptor, buffer, nbytes).
2. Write Wrapper in libc generates a Software Interrupt for system call.
3. OS in Interrupt Handler checks arguments (file_descriptor is valid file descriptor and is opened in write mode, [buffer, buffer+nbytes-1] is valid memory address. If error found, return -1 and set errno to error value.
4. OS tells Hard Drive to write buffer in [buffer, buffer+nbytes] to file specified by file_descriptor.
5. OS puts current process in wait state until disk operation is finished. OS switches to other processes.
6. Disk completes write operation and generates interrupt.
7. Interrupt Handler puts process calling write into ready state and the process will be scheduled by OS.

Part 3. Programming questions.

9. Write a shell script that will run forever and it will check every minute if a file has been modified and it will send e-mail to the user running the script when it has been modified with the changes made using "diff".

```
#!/bin/bash
 if [ ! -e $1]
 then
     echo "File does not exist"
     exit 0
 fi
 cp $1 file.Last
 while [ "1" ]
 do
     sleep 60
     cp $1 file.new
     diff file.new file.Last > diff.txt
     if [ -s diff.txt ]
     then
       /usr/bin/mailx -s "File changed" $USER < diff.txt
       echo "Message sent"
    fi
    cp file.new file.last
 done
```

10. Write a program "grepsort arg1 arg2 arg3" that implements the command "grep arg1 | sort < arg2 >> arg3". The program should not return until the command finishes. "arg1", "arg2", and "arg3" are passed as arguments to the program. Example of the usage is **"grepsort hello infile outfile"**. This command will print the entries in file **infile** that contain the string **hello** and will append the output sorted to file **outfile**. Do error checking. Notice that the output is appended to arg3.

```
int main( int argc, char ** argv)
{ const char *usage = "Usage:\t grepsort arg1 arg2 arg3\n\tgrep "
                      "arg1 | sort < arg2 >> arg3\n";

  if (argc != 4) {
     fprint (stderr, "%s\n", usage);
     exit(1);
  }
  int defaultin = dup(0); int defaultout = dup(1); int pipefd[2];
  if (pipe(pipefd) == -1) {
     perror("pipe");
     exit(1);
  }
  int infd = open (argv[2], O_RDONLY);
  if (infd < 0) {
     perror("open in"); exit(1);
  }
  dup2 (infd, 0);
   close (infd);
  dup2 (pipefd[1], 1);
  close (pipefd[1]);
  int ret = fork();
  if (ret == 0) { // Child   char *args[3]; args[0]="grep"; args[1]= argv [1]; args[2]=NULL;
     close ( pipefd[0]);
     execvp (args[0], args); perror("execvp"); _exit(1);
  }
else if (ret < 0) { perror("fork"); exit(1); }
int outfd = open(argv[3], O_WRONLY | O_APPEND | O_CREAT, 0666);
if (outfd < 0) { perror("open out"); exit(1); }
dup2(outfd, 1); close(outfd); dup2(pipefd[0], 0); close(pipefd[0]);
ret=fork();
if (ret == 0) {
   close(pipefd[1]);
   char *args[2]; args[0] = "sort"; args[1]= NULL;
   execvp(args[0], args); perror("execvp"); exit(1);
}
else if(ret < 0) { perror("fork"); exit(1); }
dup2(defaultin, 0); dup2(defaultout, 1) close(defaultin); close(defaultout);
waitpid(ret, 0, 0);
return 0;
}
```

11. **Using C++ and Semaphores** write a class SynchronizedStackSemaphores of int values where pop () will block if the stack is empty and push will block if the stack is full. Write the member variables that you think are necessary. Implement the stack with an array of int's and allocate it dynamically in the constructor. Hint: Use the "Bounded Buffer Problem" with semaphores as an example in your implementation.

```
class SynchronizedStackSemaphores {
      // Add your member variables here pthread_mutex_t _mutex; sema_t emptySem;
      int top;    sema_t _fullSem;
      int * stack;
      public:
         SynchronizedStackSemaphores(int maxStackSize );
         void push(int val );
         int pop();
};
SynchronizedStackSemaphores::SynchronizedStackSemaphores(int maxStackSize ) {
}
void SynchronizedStackSemaphores::push(int val) {
}
int SynchronizedStackSemaphores::pop(){
}
```

```
~~SynchronizedStackSemaphores::SynchronizedStackSemaphores(int maxStackSize)~~{
    top=0;
    stack=(int*)malloc(sizeof(int)*maxStackSize);
    pthread_mutex_init(&_mutex, NULL, NULL);
    sema_init(&_emptySem, 0, USYNC_THREAD, NULL);
    sema_init(&_fullSem, maxStackSize, USYNC_THREAD, NULL);
}
void SynchronizedStackSemaphores::push(int val){
     sema_wait(&_fullSem);
     mutex_lock(&mutex);
     stack[top]=val;
     top++;
     mutex_unlock(&mutex);
     sema_post(&emptySem);
}
int SynchronizedStackSemaphores::pop(){
     sema_wait(&_emptySem);
     mutex_lock(&mutex);
     top--;
     int val=stack[top];
     mutex_unlock(&mutex);
     sema_post(&_fullSem);
     return val;
}
```