**Midterm Fall 2001**

**Name:**

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

\_F\_\_ The size in bytes of a variable of type (char \*) is 1 byte.   
\_T\_\_ In a C program the expressions a[i] and \*(a + i) are equivalent.   
\_T\_\_ The strdup function calls malloc.   
\_F\_\_ On a 1-CPU computer, a program that runs in time T will run in time T/n if decomposed in n threads.   
\_F\_\_ The "S" in SMP stands for Simultaneous.   
\_F\_\_ A call to "strcmp" will show in the truss output.   
\_T\_\_ The time command in mentor could show that the user time is larger than the real time.   
\_T\_\_ A long time quantum may cause a program to finish sooner.   
\_T\_\_ The arguments of a system call are checked in kernel mode.   
\_F\_\_ The file descriptors of a process are closed when a process calls execvp().   
\_T\_\_ A process that uses pipes may hang due to unclosed file descriptors.   
\_T\_\_ A program that runs with non preemptive scheduling runs faster than one with preemptive scheduling.   
\_F\_\_ Most of the processes' CPU bursts do not finish before a context switch.   
\_F\_\_ Programs that run round-robin scheduling have faster average response time than programs that run SJF.   
\_F\_\_ When a process calls fork, the number of open file objects in the kernel is duplicated.   
\_F\_\_ POSIX threads are better than Solaris threads because the former are faster.   
\_T\_\_ The input/output redirection to files can be done by the child.   
\_T\_\_ Kernel threads in a process share  the same file descriptors.   
\_F\_\_ A section of code that is guarded by sema\_wait/sema\_post calls can be executed by only one thread at a time.   
\_F\_\_ A process table entry contains one set of registers for each user and kernel level thread in a process.

2. (3 pts.) Enumerate the fields of a process table entry 

* Process ID
* Process state
* Saved Registers
* File Descriptors
* Page Table

3. (3 pts.) Mention the checks done by the kernel during the open() system call..

* File Permissions: If the file is opened in write-mode, the user should have write permissions to the file either as user, group or others. The same for read-mode.
* If the file is opened with the flag O\_CREAT and the file does not exist, the user should have write permissions to the directory the file will be created into.

4. (3 pts.) What are the steps involved in a context switch?

* Save registers in process-table entry
* Jump to timer interrupt-handler
* Change the state of the process from running to ready
* Choose the next process to run from the ready processes.
* Set this process in running state.
* Restore the registers of the next process in the CPU
* Return from interrupt.

5. (3 pts.) What are the advantages and disadvantages of using kernel threads vs. user threads?

Disadvantages of kernel-threads

* Context Switch of kernel threads is slower because it needs to switch to kernel-mode
* Programs may take longer because of the context switch overhead of preemptive scheduling.

Advantages of kernel-threads

* User-level threads use non-preemptive scheduling so one non-cooperative thread may hang others. That does not happen with kernel-threads that use preemptive scheduling.
* Kernel-level threads may use multiple processors in SMP machines

6. (3 pts.) What are the advantages and disadvantages of using threads vs. using processes?

Advantages of threads

* Context switches among threads is faster than among processes.
* Thread creation is faster than process creation

Disadvantages of threads

 If one thread crashes the entire process crashes.

 Thread synchronization is necessary to prevent the corruption of shared data structures.

7. (3 pts.) What factors have to be considered when choosing the length of a quantum time?

* Response Time
* Context Switch Overhead
* Average CPU burst length

|  |
| --- |
| 8. (6 pts.) Assume a context switch time of 1ms, a quantum of 10ms, and preemptive round-robin scheduling. Write the time of completion for each process. Also write the context switch overhead in ms. |
| |  |  |  | | --- | --- | --- | | Process | Time needed for completion (ms) | Time of Completion(ms) | | P1 | 50 |  | | P2 | 5 |  | | P3 | 30 |  |  |  |  | | --- | --- | | Context Switch Overhead (ms) |  |     Solutions may be different. However, you should consider that:   * Processes run round-robin * Take into account context switch time. * Context switch time is part of the time-slice. * A processes may give up the CPU immediately after it completes even before the time slice expires. |

  

|  |
| --- |
| 9. (6 pts.) Mark the "C" expressions that are quivalent to *a[i].*The type of *a[i]* is *<type of a[0]>* |
| |  |  | | --- | --- | | *Expression* | *Equivalent to a[i]. Yes or Not* | | *a + i* | N | | *a + i \* sizeof( <type of a[0]>)* | N | | *&a + i* | N | | *\*(a + i)* | Y | | \*(a[0] + i) | N | | \*(&a[0] + i) | Y | | \*(&a[i]) | Y | | &a[i] | N | | \*(<type of a[0]> \*) ((char \*) &a[0] + i ) | N | | \*(<type of a[0]> \*) ((char \*)&a[0] + i \* sizeof(a[0])) | Y | |

|  |
| --- |
| 10. (15 pts.) The following class implements a multi-threaded name table. The add()  function adds the name passed as argument to the table. add() will block if the table is full. add() will allow duplicated names in the table. The remove() function removes from the table the name passed as argument . remove() will remove the first occurrence of the name. It will return 0 if the name exists or -1 if it does not. remove() will block if the table is empty. The add() and remove() calls can be called simultaneously by different threads. Insert the necessary code and member variables so multiple threads can run the operations simultaneously without corrupting the name table. The NameTable constructor receives as parameter the number of entries in the table. You have to call new/malloc to allocate enough memory for the name table and name entries. The name table is of type (char \*\*), that is, an array of type (char \*). |
| class NameTable {    /\* The size of the table \*/    int \_tableSize;    /\* This points to the array that stores the name table \*/    char \*\* \_nameTable;    /\* Extra variables here \*/    sema\_t \_full;    sema\_t \_empty;    mutex\_t \_mutex;  public:     NameTable( int tableSize ) {      /\* Initialization code here \*/      nameTable = (char \*\*) malloc( tableSize \* sizeof(char\*));      for ( int i = 0; i < tableSize; i++ ) {        \_nameTable[ i ] = NULL;      }      sema\_init( &full, tableSize );      sema\_init( &\_empty, 0 );      sema\_init( &\_mutex);      \_tableSize = tableSize;    }    void add( char \* name ) {      // Wait until there is an available entry      sema\_wait( &\_fulll);      mutex\_lock( &\_mutex );      // Find available entry      for ( int i = 0; i < \_tableSize; i++ ) {        if ( \_nameTable[ i ] == NULL ) break;      }      nameTable[ i ] = strdup( name );      mutex\_unlock( &\_mutex );      sema\_post( &\_empty);    }    int remove( char \* name )     {       sema\_wait( &\_empty );      mutex\_lock( &\_mutex );      for ( int i = 0; i < \_tableSize; i++ ) {        if ( \_nameTable[ i ] != NULL && !strcmp( \_nameTable[ i ], name ) ) {          \_nameTable[ i ] = NULL;          mutex\_unlock( &\_mutex );          sema\_post( &\_full );          return( 0 );        }      }      mutex\_unlock( &\_mutex );      return -1;    }  }; |

|  |
| --- |
| 11. (15 pts.) Complete the procedure *int redirectStdOutToPipe( char \*\* args )* that pipes the current standard output to the input of a child process running the command described in *args*. The function returns 0 if success or -1 if it fails. Look at main() to see how *redirectStdOutToPipe( )*is used. After calling *redirectStdOutToPipe()*the output of the parent process will be the pipe; the input of the parent process will be the default input (the original input of the parent); the output of the child process executing the command described in args will be the default output (the original output of the parent); and the input will be the pipe*.*Also write the procedure *restoreOutput*() that will restore the output to the way it was before calling *redirectStdOutToPipe***.** |
| int prevout;  int redirectStdOutToPipe( char \*\* args )   {    int fdpipe[ 2 ];    // Create pipe for communication    int err = pipe( fdpipe );    if ( err ) {       perror( "pipe" );      return -1;    }    // Create child    int ret = fork();    if ( ret < 0 ) {      close(fdpipe[0]);      close( fdpipe[1] );      perror( "fork" );      return -1;    }    if ( ret == 0 ) {      // Child      // Redirect input from pipe       dup2( fdpipe[ 0 ] , 0 );      close( fdpipe[ 0 ] );      close( fdpipe[ 1 ] );      execvp( args[ 0 ], args );      exit( -1 );    }    // Parent    // Save previous output    prevout = dup( 1 );    // Redirect output to pipe    dup2( fdpipe[ 1 ], 1 );    close( fdpipe[ 0 ] );    close( fdpipe[ 1 ] );    return 0;  }  int restoreStdout() {    // Restore previous output    dup2( prevout, 1 );    close( prevout );  }  main()  {      char args[ 3 ];       args[ 0 ] = "grep";       args[ 1 ] = "will be";       args[ 2 ] = NULL;      if ( redirectStdOutToPipe( args ) < 0 ) {          perror( "redirectStdOutToPipe" );          exit( -1 );      }      printf( "This Hello world will be printed to the screen\n" );      printf( "This Hello World will not be printed to the screen\n");      printf( "This Hello World will not be printed to the screen either\n");      printf( "However, this Hello World will be printed to the screen.\n");      restoreStdout();  } |

|  |
| --- |
| 12. (10 pts.) Write a script ***check-file*** that when ran in the background it will print "File modified" every time a file passed as a parameter has being modified. |
| #!/bin/sh   # Usage:  ***check-file filename***  ***# Example:***  ***# check-file  ~/.plan***  ***# It will print "File modified" every time ~/.plan is modified.***  fileName=$1  timeStamp=`ls -lu $fileName`  prevTimeStamp=$timeStamp  while [ "1" ]; do    timeStamp=`ls -lu $fileName`    if [ "$timeStamp" != "$prevTimeStamp" ]; then      echo "File Modified";    fi;    prevTimeStamp=$timeStamp;  done; |

|  |
| --- |
| 13. (10 pts.) Write the code for the function          **int strlcpy(char \*dst, const char \*src, int dstsize);**      The **strlcpy**() function copies at most **dstsize-1**  characters (**dstsize** being the  size of the string buffer **dst**) from **src** to **dst**,  truncating **src** if necessary.  The  result is always  null-terminated.  The  function  returns **strlen(src)**. Buffer overflow can be checked as  follows:       if (strlcpy(dst, src, dstsize) >= dstsize)                // dst truncated |
|  |
| **int strlcpy(char \*dst, const char \*src, int dstsize)**  {     char \* srcEnd = src + dstsize -1;    char \* s = src;    while ( \*s && s < srcEnd ) {      \*dst = \*s;      dst++;      s++;    }    \*dst = 0;    while (\*s) {      s++;    }    return s - src;  } |