Part 1. True False Questions

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

F The threads in a process share the same stack.
F The maximum number of processes in running state is determined at configuration time.
F A short quantum will benefit CPU-intensive applications
T Most of the CPU time is spent in user mode.
F The function strcpy(a,b) will allocate strlen(b)+1 bytes and copy the string from b to a.
T In pthreads, fork() will only create one thread in the child process.
F The arguments in a system call could be checked in user mode instead of kernel mode before the system call is performed.
~~T~~ It is necessary more than one mutex_lock to create a deadlock.
F It is possible to modify the interrupt vector in user space.
T Shared libraries run in user mode

Part 2. Short questions.

1. (3 pts) Explaing "aging" in process scheduling.

3 While a process is in the ready state, each time it is passed up by the scheduler for execution for a higher priority process, the process' priority is increased until it is executed. After execution the process' priority is reset.

2. (3 pts.) Explain how the quantum length can affect the priority of a processes in modern operating systems and how this is going to improve the execution of the applications.

3 Processes that <u>don't</u> use their entire quantum have their priorities increased; processes that <u>do</u> use their entire quantum up have their priorities decreased. This ensures that interactive applications (which don't use the entire quantum) receive higher priority and more responsiveness.

3. (3 pts.) a) Write the code for spinlock/unlock? b) What problems we may encounter if we remove the thread_yield() instruction?

3 (a)

```
Void spin_lock(unsigned long *lock)
{ while (test_and_set(lock) == 1)
  { thr_yield(); }
}
```

```
Void spin_unlock(unsigned long *lock)
{
   *lock = 0;
}
```

(b) if thr_yield() is removed, then if the lock fails the spin_lock() function will poll on the lock variable and it will waste its quantum. The lock will still work as expected.

4. (3 pts.) Enumerate the checks that the kernel does during the system call open(filename, mode)

3

1) checks if filename exists
2) checks if user can open filename in mode given permissions settings
3) checks if file has already been opened for writing
4) checks if there are sufficient number of available file descriptors.

3

5. (3 pts.) What are the steps for processing an interrupt.

1) save registers
2) call handler from interrupt vector/table
3) restore registers
4) retry last instruction if necessary
5) resume process

3/3

6. (3 pts.) Explain why it is a good idea to choose a quantum that is longer than the average CPU burst.

A quantum that is longer than the avg. CPU burst ensures that the burst length is not spread over more than one quantum. If it were this would cause more context switch and scheduler overhead.

3/3

7. (3 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	no
*a + i * sizeof(<type of a[0]>)*	no
&a + i	no
**(a + i)*	yes
*(a[0] + i)	no
*(&a[0] + i)	yes
*(&a[i])	yes
&a[i]	no
*(<type of a[0]> *) ((char *) &a[0] + i)	no
*(<type of a[0]> *) ((char *)&a[0] + i * sizeof(a[0]))	yes

2/3

8. (3 pts.) What are the advantages and disadvantages of using threads vs. using processes?

	Threads	Processes
Advantages	1) less overhead in creation and context switch	1) if one process dies, the rest do not
	2) simpler communication and faster	2) security / protection ✓
Disad V.	1) mutual exclusion (process too)	1) more overhead right
		2) complex IPC more and less

3/3

9. (3 pts.) What is the problem of having long critical sections?

a long critical section means that only one thread will be able to be in that section at a time. In parallel systems this can cause several CPUs to go unused. Long critical sections are the source of bottle necks.

3/3

10. (3 pts.) Explain step by step the life cycle of a program from editing to running.

1) write code
2) compile to object format
3) link into executable, noting shared object dependancies
4) execute program:
5) map program into RAM, give it an entry in the process table
6) map stack, text, data, bss, heap
7) load + map shared object libraries
8) execute "_start" function
9) jump to main(...)

Part 3. Programming questions.

10

11. (10 pts.) Implement the function char *strstr(const char *haystack, const char *needle). The strstr() function finds the first occurrence of the substring *needle* in the string *haystack* and returns a pointer to the beginning of the substring. The terminating `\0' characters are not compared.

```
char *strstr(const char *haystack, const char *needle)
{
    char * start = haystack;
    while (start != 0) {
        char*a = start;
        char*b = needle;
        while (*a == *b && *a!=0 && *b!=0) {
            a++; b++;
        }
        if(*b==0)
            return start;
        start++;
    }
    return NULL;
}
```

12. (10 pts.) Implement the class Array3D of type **double** that creates a 3D array of 3 dimensions *m x n x r* as indicated in the constructor. Also implement the function getElementAt and setElementAt that gets and sets the element at this location. Base your implementation in the array of pointers to rows that we covered in class.

```
class Array3D {

// Add any variables you need
double ***arr;
int    m, n, r;
public:
  Array3D( int m, int n, int r);

  double getElementAt( int m, int n, int r);
  void setElementAt( int m, int n, int r);
};

Array3D::Array3D( int m, int n, int r)
{ this->m=m; this->n=n; this->r=r;
   arr = new (double **)[m];
   for(int i=0; i<m; i++){
      arr[i] = new (double*)[n]
      for(int j=0; j<n; j++){
         arr[i][j] = new double[r];
}
}
double
Array3D::getElementAt( int i, int j, int k)

{
    if(i<m && j<n && k<r)
       return arr[i][j][k];
    else
       return 0.0;

}

void
Array3D::setElementAt( int i, int j, int k, double val)
{
    if(i<m && j<n && k<r)
       arr[i][j][k] = val;

}
```

#13

15. (10 pts) Assume that there are n threads t_0, t_1, t_2... t_{n-1} where the ith thread needs to lock two mutexes mutexi and mutex(i+1)%1 to do an operation as shown in the code below. Assuming that n=6,

a) Using a time diagram (like the ones we use in class that shows multiple threads) show the sequence of events that can lead to a deadlock.
b) Represent this deadlock in a graph.
c) Modify the code below so the deadlock can be prevented.

write a program that implements "ls -laR arg1 | grep arg2" where arg1 and arg2 are passed as arguments to the program

```
int n = 4;

thread_function(int i)
{
    while(1) {
       mutex_lock(&mutex[i])
       mutex_lock(&mutex[(i+1)%n);

       // Access resource

       mutex_lock(&mutex[i])
       mutex_lock(&mutex[(i+1)%n);
    }
}
```

14

```
int main (int argc, char **argv) {
    if (argc != 3) {
      fprintf (stderr, "incorrect arguments");
      return 0;
    }
    char **arg1 = new (char*)[4];
    char **arg2 = new (char*)[3];
    int pipefd[2];
    pid_t proc1, proc2;
    arg1[0] = "ls";
    arg1[1] = "-laR";
    arg1[2] = argv[1];
    arg1[3] = NULL;
    arg2[0] = "grep";
    arg2[1] = argv[2];
    arg2[2] = NULL;
    if (pipe(pipefd) == (-1)) {
       fprintf(stderr, "err in pipe()\n"); return 0; }
```

```
int inbak = dup(0);
int outbak = dup(1);
dup2(pipefd[1], 1);
proc1 = fork();
if(proc1 == 0)
{  close(inbak);
   close(outbak);
   close(pipefd[0]);
   execvp(arg1[0], arg1);
   return 1;
}else if(proc1 > 0)
{  dup2(outbak, 1);
   close(outbak);
   dup2(pipefd[0], 0);
   close(pipefd[0]);
   close(pipefd[1]);
   proc2 = fork();
   if(proc2 == 0)
   {  close(inbak);
      execvp(arg2[0], arg2);
      return 1;
   }else if(proc1 > 0)
   {  dup2(inbak, 0);
      close(inbak);
      wait pid(proc1);
      wait pid(proc2);
      delete[] arg1;
      delete[] arg2;
   }else
   {  fprintf(stderr, "error in fork()\n");
      return 1;
   }
}else
```

close(pipefd[0])

```
{ fprintf(stderr, "error in fork\n");
  return 1;
}
return 0;
}
```

14. (15 pts.) Write a class ***AlertTable*** that implements two methods: ***void insert(key, data)*** and ***char * remove(key)***. The method ***insert(key,data)*** will add the pair ***(key,data)*** to the table and will wakeup any thread calling ***remove(key)*** that is waiting for that specific key. The method ***remove(key)*** will check if there is an entry already with that key in the table and it will remove it if it is found. Otherwise, it will wait until that entry is inserted. ***remove(key)*** will return the data that corresponds to that key. If two ***remove(key)*** calls wait simultaneously for the same key, the oldest one will wakeup after the first insert call for that key and the newest one will wakeup after a subsequent ***insert*** call with the same key. Implement the table as a list, array, hash table or in any way you want. There is no limit in the size of the table. The type of the key is ***char ****.

15

```
struct AlerTableEntry {
   char * key;          struct AlerTableEntry * next;
   void * data;         int count;
   // Other fields
   sema_t sem;
};


class AlertTable {
  // Declare alert table and other fields
  struct AlerTableEntry * head;
  mutex_t mutex;

public:
  AlertTable();
  void insert(char * key, void * data);
  void * remove(char * key);
};

Alerttable::AlertTable()
{
  // Write ay initializations.
  head = new AlertableEntry; head -> next = NULL;
  head -> key = NULL;   mutex_init(&mutex, USYNC_PROCESS, NULL);
}  head -> data = NULL;

void
AlertTableEntry::insert(char * key, void *data)
{  mutex_Lock (&mutex);
   AlerTableEntry *curr = head -> next; AlerTableEntry*prev = head;
   while (curr != NULL && strcmp(curr->key, key) ==0) {
       prev = curr; curr = curr -> next; }
   if(curr == NULL){curr = new AlerTableEntry; curr->key=key;
       curr -> data = data; sema_init(&curr -> sem, 0, USYNC_PROCESS,0);
       curr -> next = NULL; curr -> count = 0; }
   Prev -> next = curr; curr -> data = data;
char *
AlertTableEntry::Remove(char * key)
{  for(int i =0; i < curr->count; i++)
       sema_post(curr -> sem);
   curr->count = 0;
   mutex_unlock(&mutex);
}
```

```
char * AlertTableEntry::remove(char *key) {
    mutex_lock(&mutex);
    AlerTable Entry *curr = head->next;
    AlerTable Entry* prev = head;
    while (curr != NULL && strcmp(curr->key, key) == 0) {
        prev = curr; curr = curr->next; }
    if(curr == NULL) {
        curr = new AlerTable Entry; curr->key = key;
        curr->data = NULL; curr->next = NULL;
        sema_init(&curr->sem, 0, USYNC_Process, 0);
        curr->count = 0; prev->next = curr;}
    if (curr->data == NULL) {
        curr->count += 1;
        mutex_unlock(&mutex);
        sema_wait(&curr->sem);
        mutex_lock(&mutex);
        curr->count --;
        for(prev = head; prev->next != curr; prev = prev->next){}
    }
    char * data = (char*) curr->data;
    if (curr->count == 0) {
        prev->next = curr->next;
        delete curr;
    }
    mutex_unlock(&mutex);
    return data;
}
```

```
t}
```

15. (10 pts) Assume that there are n threads t_0, t_1, t_2... t_{n-1} where the ith thread needs to lock two mutexes $mutex_i$ and $mutex_{(i+1)\%n}$ to do an operation as shown in the code below. Assuming that n=6,

10

a) Using a time diagram (like the ones we use in class that shows multiple threads) show the sequence of events that can lead to a deadlock.
b) Represent this deadlock in a graph.
c) Write a second version of *thread_function* where the deadlock is prevented.

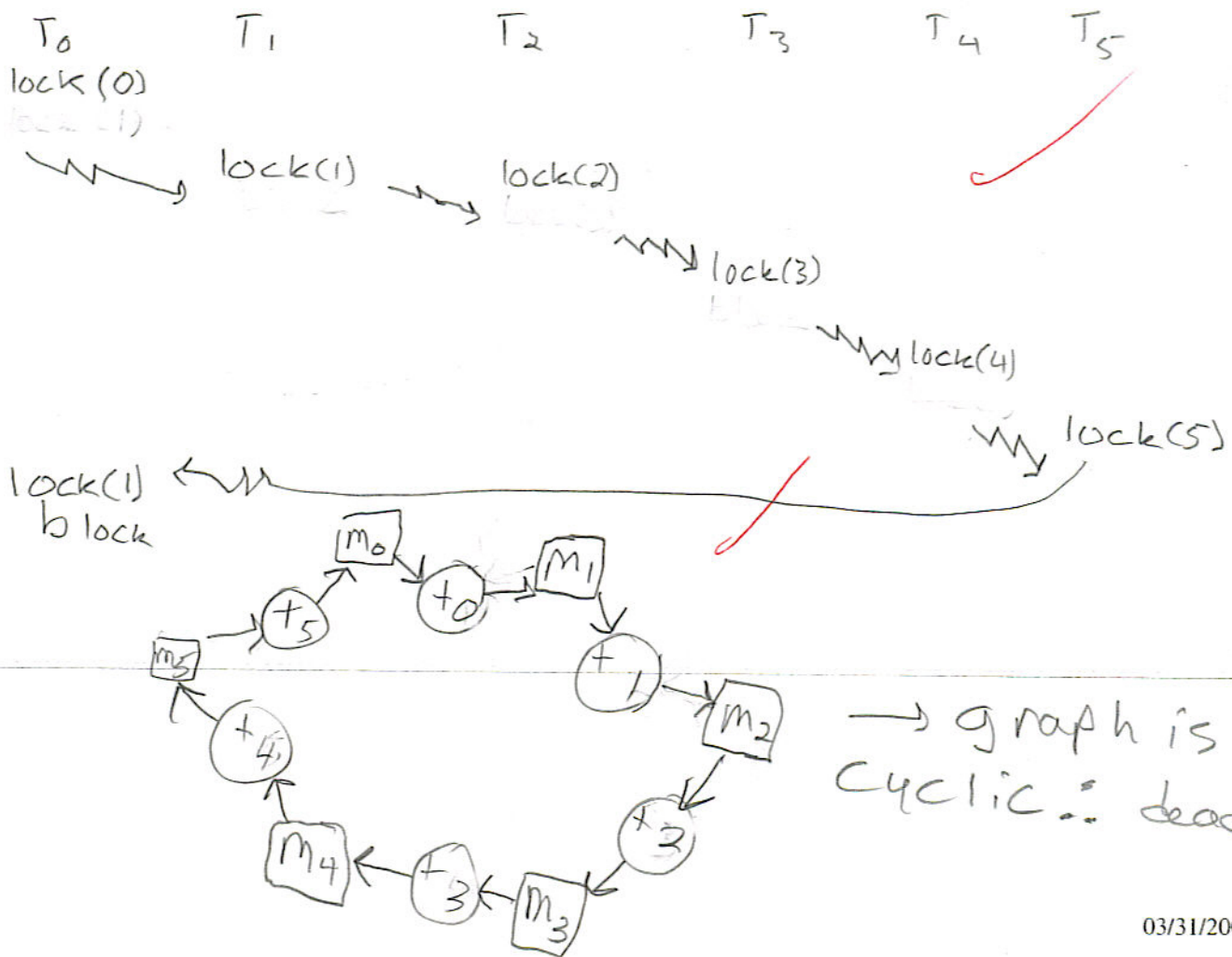
```
int n = 6;

thread_function(int i)
{
    while(1) {
        mutex_lock(&mutex[i])
        mutex_lock(&mutex[(i+1)%n);

        // Do operation

        mutex_lock(&mutex[i])   mutex_unlock
        mutex_lock(&mutex[(i+1)%n); mutex_unlock
    }
}
```

Start

lock 0 → lock0
lock 1 → lock1
lock 2 → lock2
lock 3 → lock3
lock 4 → lock4
lock 5 → lock5

block

deadlock



→ graph is cyclic ∴ deadlock!

```
int n = 6

hread_function(int i)
{
    int low = (i <((i+1)%n) ? i : ((i+1)%n);
    int high = (i >((i+1)%n) ? i : ((i+1)%n);
    while(1){
        mutex_lock(&mutex[low]);
        mutex_lock(&mutex[high]);
         // do stuff...
        mutex_unlock(&mutex[high]);
        mutex_unlock(&mutex[low]);
    }
}
```