1 (a) In object oriented programming, describe the purpose of virtual functions. [10%]

Virtual functions provide polymorphism to classes. That is that two different classes can provide the same interface [5], yet have different functionality. [5]

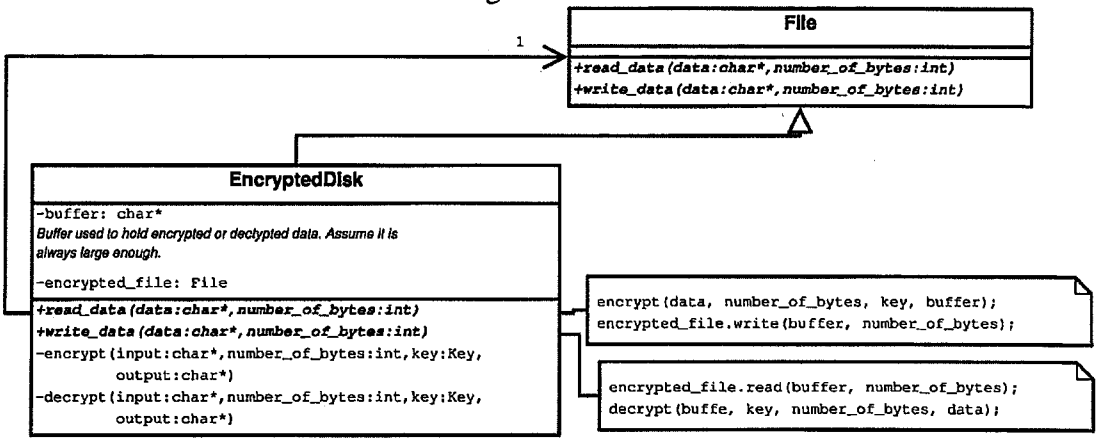
(b) Consider how an object oriented operating system could handle file access, so that files on different media use the same interface. The UML diagram in Fig. 1 describes the `File` class and the associated class hierarchy.

The design includes a proposed class `EncryptedDisk` which transparently treats an encrypted file on disk as a normal file.

Discuss the proposed design for `EncryptedDisk` and give a design which provides encryption for all file types and will extend easily to any new file types. Include a UML diagram for your design. [30%]

The proposed design has the advantage that it is simple to implement[5], however the disadvantage is that the functionality (encyption) depends on something irrelevant (that it is a disk file).

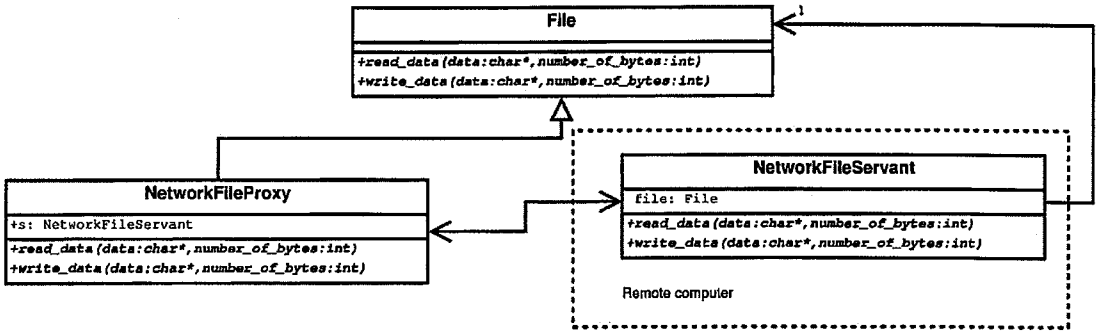
New design should be the decorator pattern. `EncryptedFile` *should inherit from* `File` *[5] and have a private instance of* `File` *[5]. The note about the methods should be rewritten to use the private instance rather than* `Disk::` *[5]. The remaining [10] marks are for correct notation in the UML diagram.*



(c) Extend the design to allow network support so that the system can read files on a different computer, and give a UML diagram for the new design. [30%]

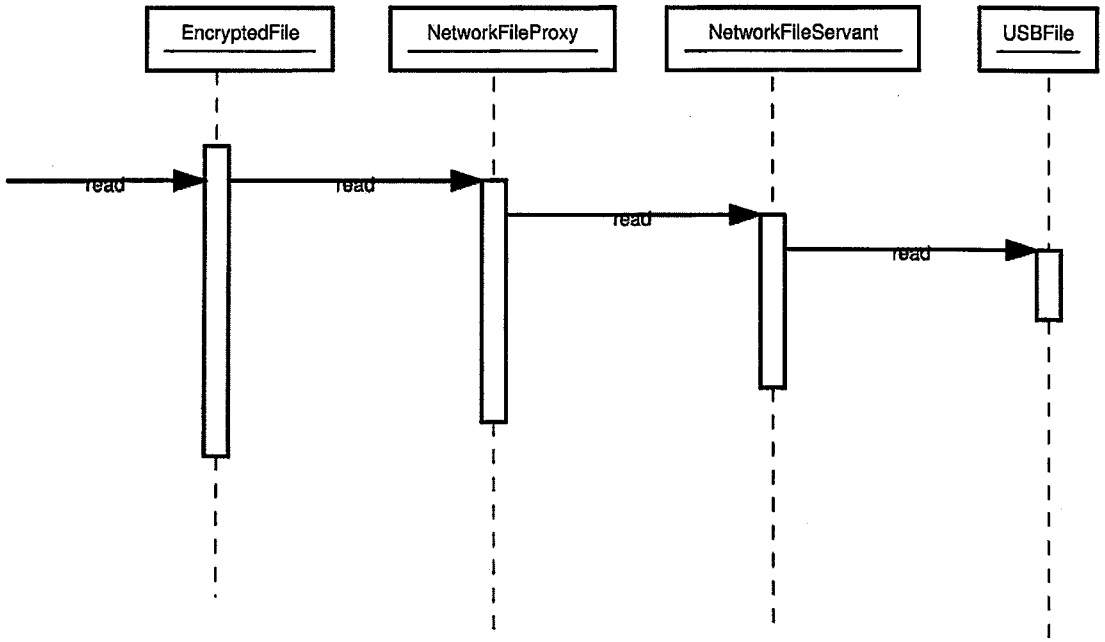
The new design should be an instance of the proxy pattern. There should be two new classes. One class `NetworkFileProxy` *should derive from* `File`*[5]. The other class*

(e.g.) `NetworkFileServant` *should own an instance of* `File`*.[5].* `NetworkFileProxy` *needs to be connected to* `NetworkFileServant` *in both directions [5]. The servant may also derive from* `File`*. The servant mist reside on another computer. [5] The remaining [10] marks are for correct notation in the UML diagram.*



(d) Give a sequence diagram for the act of reading an encrypted file from a USB device, over the network. [30%]

Note, answers may vary if the student gives a very different answer for the previous part.



Correct ordering of classes [10] (note encryption can occur between USB and Servant. The lifetime boxes get shorter from left to right [10]. Remaining notation correct [10].

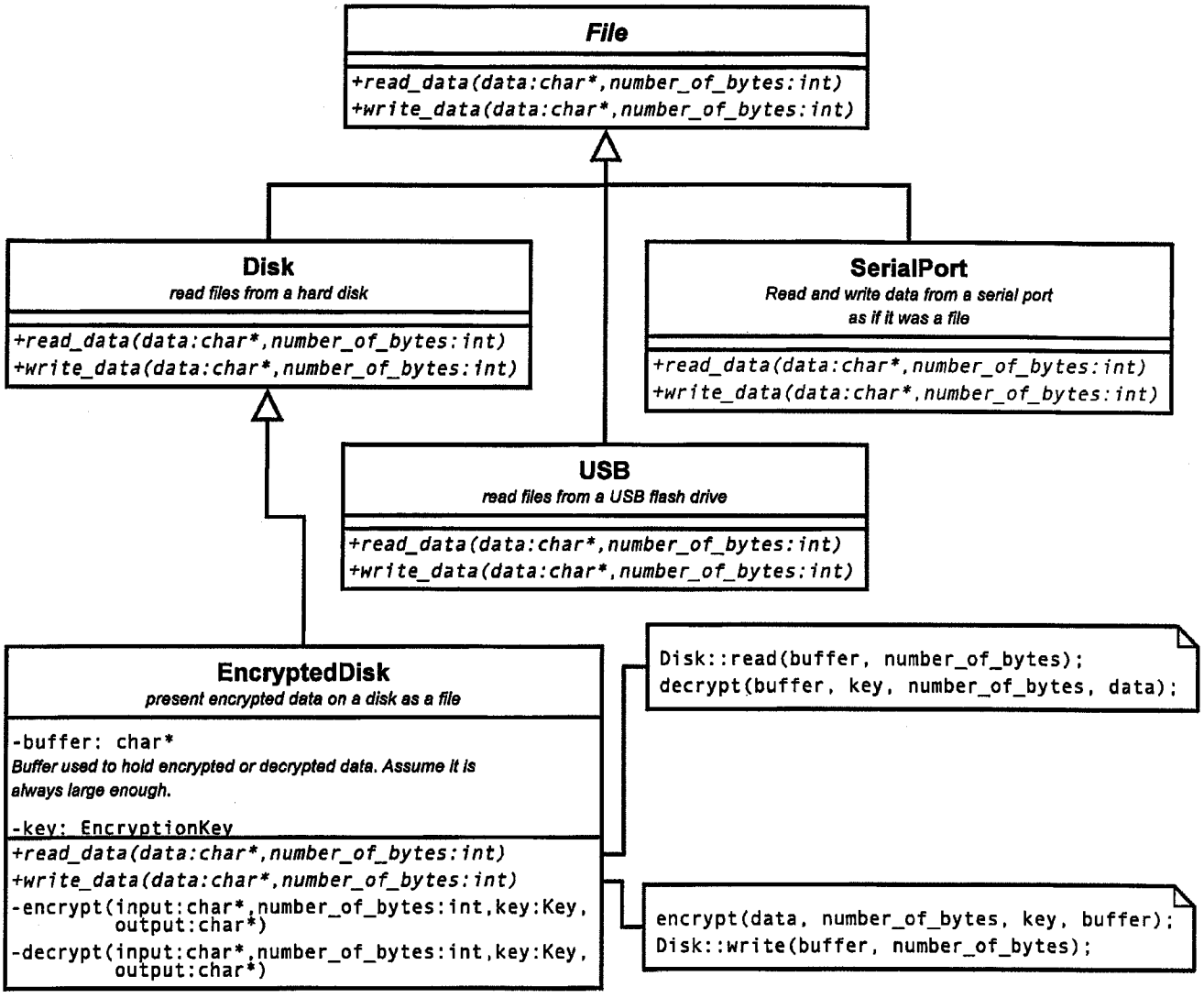


Fig. 1

Version: 3

2 (a) In concurrent programming, describe what a monitor class is, and what the role of signals are. [15%]

A monitor class is a class where methods are protected by mutexes which prevents more than one thread from calling a method at any one time [5]. Inside a method, a thread may need to wait for an event, at which point it has to pause and release the mutex [5]. A signal allows a waiting thread to restart [5].

(b) Explain the terms 'critical section' and 'mutual exclusion' and show how semaphores can be used to implement mutual exclusion. [15%]

A critical section is an area of code which cannot be safely accessed by more than one thread. Mutual exclusion means that when one thread is in a critical section, it prevents other threads from entering. [5]

```
    Semapore s(1); //Initialise semaphore to 1
...
s.post();
//Critical section
s.wait();
```

[10]

(c) A bounded buffer class allows messages to be passed from one thread or process to another. In C++ a skeleton of the class implementing the bounded buffer is:

```
class BoundedBuffer{
    public:
        void send(Message m);
        Message receive();
};
```

Using only semaphores, provide implementations for `send()` and `receive()`, and list any private variables which the `BoundedBuffer` class needs. The thread calling `receive()` must wait if there are no pending messages. The thread calling `send()` must wait if the buffer is full. [50%]

Upto 30 marks allocated if the example from the lectures using mutexes and signels is used.

Approximate marks: Three semaphores are required, one to act as a mutex, one

to count the number of full slots and one to count the number of empty slots [20]. The methods must both be protected by the mutex semaphore [5]. `send` *has to wait for an empty slot and later indicate that a slot is full [5].* `receive` *has to do the opposite [5].*

Posting to full/empty must happen outside the mutex otherwise a deadlock will occur [10]. If it happens inside, then extra locking and unlocking must occur.

Semaphores need to be initialised to the correct values.[5]

Example code:

```
class BoundedBuffer{
static const int N=100;
Message buffer[N];
int next_in=1, next_out=0;
Semaphore mutex(1), full(0), empty(N);
public:
    void send(Message m){
        empty.wait();
        mutex.wait();

        buffer[next_in++] = m;
        next_in = next_in
        mutex.post();
        full.post();
    }

    Message receive(){
        full.wait();
        mutex.wait();

        Message m = buffer[next_out++];
        next_out = next_out
        mutex.post();
        empty.post();

        return m;
    }
};
```

An alternative, correct answer using polling:

```
class BoundedBuffer{
static const int N=100;
list<Message> buffer;
Semaphore mutex(1);
public:
    void send(Message m){
        mutex.wait();
        while(buffer.size() == N){
            mutex.post();
            mutex.wait();
        }

        buffer.push_back(m);
        mutex.post();
    }

    Message receive(){

        mutex.wait();
        while(buffer.size() == 0){
            mutex.post();
            mutex.wait();
        }
        Message m = buffer.pop_front();

        mutex.post();

        return m;
    }
};
```

(d) An operating system does not provide semaphores and mutexes to users, but it does provide message passing using the `BoundedBuffer` class. Show how mutual

exclusion for a block of code can be implemented using the `BoundedBuffer` class. [20%]

A BoundedBuffer class is very much like a semaphore, where the number of messages in the buffer are equivalent to the value of s semaphore. The answer should use a BoundedBuffer in this manner.[10]

The BoundedBuffer must be initialised to 1 by sending a single message.[5]

The contents of the messages are irrelevant and should be ignored.[5]

```
BoundedBuffer b;
Message m;
b.send(m);

...
...

b.receive();
...  //Mutual exclusion happens here
b.send(m);
```

3 (a) In CORBA based distributed systems, what is an IOR, what does it contain and why? [15%]

RepositoryID (name of the object, IP address and port number [10]. This aprovides all the information needed to find the remote computer and the object on it [5].

(b) It has been decided to use CORBA to design a simple graphical windowing system which allows a computer display to be used by programs running on another computer. The computer running the display is referred to as the display server.

The display server must provide an interface which allows the client to create `Window` objects of a certain size. The `Window` object must provide a method, `setpixel` to allow the client to set a pixel to a particular value.

Give the CORBA IDL that will be needed to provide this functionality. [25%]

Example IDL [25]:

```
interface Window{
    void setpixel(in int r, in int g, in int b, in int x, in int y);
};

interface DisplayServer{
    Window open(in int width, in int height)
}
```

(c) The design needs to be extended so that the remote program can get a record of any mouse and keyboard events that occur in the window. The proposed design modifies the IDL to be:

```
enum Type {MouseButtonPress, KeyPress};
struct Event
{
    Type event_type;
    short key_or_button_number;
    short mouse_position_x, mouse_position_y;
};

interface Window
```

```
{
    ...
    sequence<Event> get_events();
};
```

What problems would be encountered when a large number of windows are open with this design. What would an alternative be which did not suffer from the same problem? Give the CORBA IDL necessary for the improved design. [30%]

For responsiveness, applications have to poll regularly for events. With many windows open, this will generate a very large amount of network traffic.[15]

Example IDL [15]:

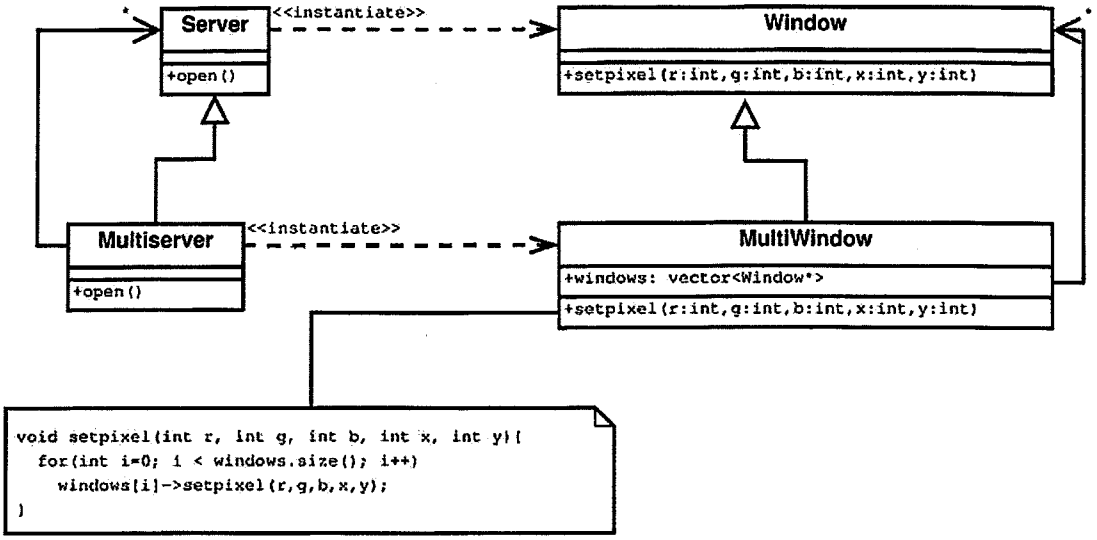
```
interface EventReturn{
    void event(in Event); };
interface Window{
    void setpixel(in int r, in int g, in int b, in int x, in int y);
};

interface DisplayServer{
    Window open(in int width, in int height, in string
event_return_ior)
}
```

Note that a window needs to be given an EventReturn in order to pass events back. The easiest way of doing this is to pass an IOR as a string.

(d) The system can be extended so that clients can open the same window on several display servers simultaneously. Using good design principles, show how the system can be extended by giving a UML diagram for the extended system, and describe how the `setpixel` method works when the client has a single window open on multiple displays. [30%]

The new server has to inherit from server and also contain many servers [10]. Likewise for windows [10]. In order to set a pixel, MultiWindow has to iterate over its list of windows and call setpixel on each one. UML diagram [10]:


Server
+open()
<<instantiate>>
Window
+setpixel(r:int,g:int,b:int,x:int,y:int)
Multiserver
+open()
<<instantiate>>
MultiWindow
+windows: vector<Window*>
+setpixel(r:int,g:int,b:int,x:int,y:int)
void setpixel(int r, int g, int b, int x, int y){
for(int i=0; i < windows.size(); i++)
windows[i]->setpixel(r,g,b,x,y);
}

4 (a) In database systems, give an example of how concurrency can cause the database state to become inconsistent. [15%]

Bookwork: example will be one of uncommitted dependency, lost update or inconsistent analysis [5]. Example must show how the wrong data gets used [10].

(b) Figure 2 shows the intended actions present in three concurrent database transactions, T1, T2 and T3 operating on four resources, A, B, C and D. The database is designed to provide ACID transactions by protecting resources using locks. Therefore for a resource *R*, the action *R*.`read` automatically attempts to obtain the shared lock *R.S*, and the action *R*.`write` attempts to obtain the exclusive lock *R.X*. Assume that locks are held by a transaction until it commits or aborts.

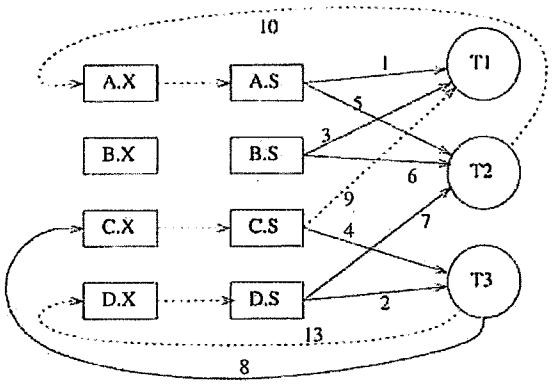
Identify the first point that a deadlock occurs, and draw a resource allocation graph. [25%]

The sequence of actions is as follows. Bracketed actions do not occur because of a wait. The number following an action indicates the timeslot of the action being waited for.

Note that if the wait in slot 9 is missed, then a deadlock will be incorrectly found at time 13 instead of 14. [15] marks for the correct answer.

	T1		*T2*		*T3*
1	*A.read*	*1*		*1*	
2		*2*		*2*	*D.read*
3	*B.read*	*3*		*3*	
4		*4*		*4*	*C.read*
5		*5*	*A.read*	*5*	
6		*6*	*B.read*	*6*	
7		*7*	*D.read*	*7*	
8		*8*		*8*	*C.write*
9	*C.read 8*	*9*		*9*	
10		*10*	*A.write 1*	*10*	
11		*11*		*11*	*B.read*
12		*12*	*(commit)*	*12*	
13	*(a.write 5)*	*13*		*13*	
14		*14*		*14*	*D.write 7*
15	*(commit)*	*15*		*15*	
16		*16*		*16*	*(commit)*

Resource diagram [10]:



(c) Explain a strategy that can be used to recover from a deadlock. [10%]

The database must select a victim transaction to kill, which allows the other transactions to continue [5]. A good one to kill is a transaction holding many locks or a transaction that has not been running long [5].

(d) An alternative to automatically locking the database during transactions is the optimistic lock-free strategy. Explain how the lock-free optimistic strategy operates and maintains the ACID properties of transactions. [30%]

When the transaction starts, shadow copies of the accounts are taken and changes are only made to these copies [5]. When the transaction asks to commit, the transaction undergoes validation to ensure that there could not have been any conflicts with transactions already accepted [5].

Running transactions work with shadow copies, which makes running transactions isolated, and if they fail to stay isolated, then they are replayed [5].

If there are conflicts, then the database may have become inconsistent so the shadow is discarded and the transaction is replayed. This provides consistency.[5].

Otherwise the whole shadow is copied into the database (atomicity) [5].

There is on effect on durability directly. However, the optimistic strategy requires transactions to be replayable which is necessary for durability in the event of a system crash. [5]

(e) Describe the sequence of actions which would happen in part 4(b) in the optimistic case and state which accounts would be updated and which transactions would have to be restarted. [20%]

Transaction 2 commits first so it will succeed. Transaction 1 will notice that A has been written, so it will have to restart. Transaction 3 will succeed because none of its resources have been written.[20].

Time	Action	Transaction	Time	Action	Transaction
1	A.read	1	9	C.read	1
2	D.read	3	10	A.write	2
3	B.read	1	11	B.read	3
4	C.read	3	12	commit	2
5	A.read	2	13	A.write	1
6	B.read	2	14	D.write	3
7	D.read	2	15	commit	1
8	C.write	3	16	commit	3

Fig. 2

END OF PAPER