Aniruddha Tekade (B00618834) March 13, 2017

CS 551
System Programming
Homework 2

Solution 1 ⇒ Following mechanisms can be used for specified questions ⇒

1. We can change the interface by adding just three simple things :
   (a) Boolean flag for error checking (If this flag is set, error checking will be done.)
   (b) Hashmap for storing the errors.
   (c) Linking the new library to the existing one.
2. For the users who are only interested in the new changes in the library, they will be able to access it.
3. Users who are interested in the string errors will have the boolean flag = true and will not have to run extra code for explicitly checking the erros.
4. The same flag functionality will be used by some users who have multiple obects alive. Due to linking of library to the previous implementation, the ojects from the client side would be able to access it.

Solution 2 ⇒ Accessing a resource is generally a `critical section` in the system. And to access the critical section there are many solutions like to use the mutex (binary semaphores) or semaphores

If there are N interchangeable resources to be shared, we can use a simple locking mechanism on the shared resources and make processes interested to acquire exclusive access to wait untill the lock is available for them to acquire.

```
while (lock is aquired)
  wait();
if (lock is free)
  Set the lock
  <Enter into the critical section>
  Processing
  <Exit from the critical section>
  Release the lock
```

This mechanism is normally called as mutexes and is best solution to handle mutual access to resources to be shared between different processes.

Solution 3 ⇒ Given code snippet is :

```
1. char c;
2. int f() {
3.     void *fP = &f;
4.     void *cP = &c;
5.     return ((cP < (void *)&cP)<<1) + (fP < (void *)&fP);
6. }
```

In the line#5, the return statement has two logical statements that are returning two different additions. The first part which is `(cP < (void *)&cP)<<1)` produces 2 and the second part `(fP < (void *)&fP)` produces 1.

Reason ⇒ While dealing with the pointers and their respective values, a variable `c` is allocated first in the memory and then the function `int f()`. After these allocation, we are defining pointers pointing to these relative entities ex. `c` has `cP` and `f()` as `fP` in this case.

Since memory grows from lower addresses to higher addresses, it is very obvious that the address-value (integer or hexadecimal would not make any difference;) of `c` is smaller than its pointer `cP` and `f()` smaller than `fp`.

Therefore, first part produces 1 which is *left shifted by 1 bit* and it becomes **2** & added to second part output **1** returning a total of **3**.

Solution 4 ⇒ This kind of situation can arise in `client-server architecture` implemented using pipe or a message passing from multiple children process to a single parent process.

Assuming that we are using `anonymous pipes` & OS is POSIX-compliant in this regard, we can have following simple protocol ⇒

1. Set a `boolean` lock on pipe equals to 1 (available).
2. The writing process will check the lock availablity and write to it. Meanwhile the reading process is in sleep state.
3. If the lock is unavailable, writing process will wait untill it gets released by some other process.
4. After finishing the writing, it will signal the reading process and release the lock.
5. This process will go on untill there exists at least 1 reading or wiring process.

However, there can be some other approach to this problems which is as mentioned below ⇒

- Write requests of `PIPE_BUF` bytes or less shall not be interleaved with data from other processes doing writes on the same pipe. Writes of greater than `PIPE_BUF` bytes may have data interleaved, on arbitrary boundaries, with writes by other processes, whether or not the `NO_BLOCK` flag of the file status flags is set.
- `PIPE_BUF` is, by the way, guaranteed to be at least 512. But this size can be changes with a paramter and we can make sure that every process that is writing or reading from or to the pipe will contrain to the size of bytes.

Solution 5 ⇒ Password re-hashing

When user logs in next time, we can use the password extension's `password_verify()` function. If it does not succeed, the process falls back on the old MD5 hash algorithm. If the MD5 hash matches then we can rehash the password using `blowfish_hash()` and save it in the old hash's place. The sample code can be as below ⇒

```
if (password_verify(passwd, hash)) {
    /**
     * Logic if they are matched
     */
} elseif (hash('MD5', passwd) == hash) {
    /**
     * Logic if they are matched
     */
    newHash = blowfish_hash(password);
    /**
     * Replace old hash with newHash
     */
} else {
    fprintf(stderr, "Invalid Password!");
}
```

Solution 6 ⇒ Initially, `uid` = 1000, `euid` = 1000, `saved set-user-ID` = 1000

(a) There is no change in the effective uid or set-user-ID. `uid` = 1000
`euid` = 1000
`saved set-user-ID` = 1000

(b) Since setuid bit is set, effective uid is going to be 2000. `uid` = 1000
`euid` = 2000
`saved set-user-ID` = 2000

(c) setuid() sets the uid as well as effective uid of the process. And since we've exec() it, the same euid will be copied to the set-user-id.
`uid` = 2000 `euid` = 2000 `saved set-user-ID` = 2000

(d) This will set the effective uid as well as saved set-user-id.

`uid` = 2000
`euid` = 1000
`saved set-user-ID` = 1000

(e) This will set the effective and real uid.
`uid` = 1000
`euid` = 2000
`saved set-user-ID` = 2000

(f) This will set the real uid, effective uid and saved set user id = 1000.
`uid` = 1000
`euid` = 1000
`saved set-user-ID` = 1000

Solution 7 ⇒ Following table describes the file access required as per the question ⇒

	data1 R	**data1 W**	**data2 R**	**data2 W**
u1 runs exec1	N	N	Y	Y
u2 runs exec1	Y	Y	-	-
u1 runs exec2	N	N	Y	Y
u2 runs exec2	-	-	-	-

Solution 8 ⇒ Setting up the ring communication structure with all IPC handled using anonymous pipes can be simply build as follows-

- The `parent` communicates with the first child process through its file descriptor for writing.
- This first process `P[i]` uses its file descriptor (for reading) to read the data that is sent by the parent.
- It also uses its file descriptor (for reading) to send data to the second child, and this process continues from second child to third child and so on. (We may find the usage of `dup()` usefule for using the file desciptor)
- This should be forming a ring topological structure between the processes.
- The last (child) process sends the data back to the parent process.

To achieve this mechanism we can follow following simple steps ⇒

- `Parent` creates pipe for it to write to 1st child `P[0]` & Parent keeps open the write end of pipe to `P[0]`.
- Parent keeps open the read end of the pipe from Nth child.
- For each child `P[i]` = `P[0], P[1], . . . P[(i+1)%N]`, `Parent` creates output pipe for i^{th} child to talk to $i + 1^{th}$.

- Parent forks n^{th} child and n^{th} child closes the write end of its input pipe and the read end of its output pipe.
- n^{th} child reads the data from previous child, process it (if any) & writes new data to the output pipe, then exits.
- Meanwhile, Parent closes both ends of the input pipe to the nth (except for the descriptors it must keep open), and loops back to create the n+1th child's pipe and then the child.

Solution 9 ⇒ `execl()` can be used as the primitive function instead of `execve()`

The definition of this call is ⇒ `int execl(const char *path, const char *arg0, ..., const char *argn, (char *)0);`

The reason for this statement is that, most of the functions expect a pathname or a filename as the specification of the new program to be loaded. This basic requirement of every other function in the family of `exec()` can be fulfilled using `execl()` function since its can be customized with the number of parameters.

Solution 10 ⇒ Discuss the validity

(a) `Valid`. Because when a pipe is no longer in use, there are chances of getting errors like `broken-pipe` which may stop the program exucution in progress accidently. Therefore, should always take care of closing the pipe ends whenever not in use.

(b) `Valid` when, file is attributed with only three types of permission as far as users are concerned and has not been added by creator into any such user group which is administrative level.

`Invalid` when, file is not set up with any kind of other group ids that are administrative level

(c) `Valid`. For example, when we login to the unix via local user, the programs like `terminal` and others are always started with `root` priviledges and we can use then. The basic reason behind this is that they are always started using the `effective uid` of the `root`.

(d) `Valid`. It is true that when the execute (X) permission are set to a directory, the process can always apply `ls` command on it.

(e) `Valid`. After running `exec()` call, the process gets an effective uid. This effective uid is generally copied fro the owner of the file.
