Assignment UP-04

Data Structures:

In thread.h:

```
extern struct thread* tid_arr[2040];     // Mapper tid to thread pointer
extern int tid_parent[2040];             // Mapper tid to parent's tid
extern bool tid_arr_loaded[2040];        // Mapper tid to load status
extern int tid_arr_exit_status[2040];    // Mapper tid to exit status

struct semaphore loaded;                 // semaphore initialized to 0, for synchronizing exec
struct semaphore exited;                 // semaphore initialized to 0, for synchronizing wait
struct semaphore exit_ack;               // semaphore initialized to 0, for synchronizing wait

struct file* executable;                 // To store file pointer of the executing file used by thread
```

Algorithm:

Exec:

In exec system call command line input is given as parameter.

1. First sanity check is made on the given input.
2. Then process_execute function is run on the given cmd_line_input.
3. Now tid returned by process_execute() function is checked.
4. If tid == TID_ERROR, then return -1
5. Else get thread corresponding to that tid, using tid_arr[]
6. Now ensure synchronization using “loaded” semaphore.
7. If load is failed (which is identified using tid_arr_loaded[] array) , then return -1, else return tid.

Wait:

In wait system call, tid of child thread is given as input.
We just call process_wait () function on given child_tid, return value returned by process_wait()

Process_wait ():

In process_wait function, child_tid is given as input.

1. First we check validity of given pid.
2. Then thread corresponding to that tid is extracted using tid_arr[]
3. Parent-child relationship is ensured using tid_parent[] array,
   If tid_parent[child_tid] != thread_current()-> tid, then return -1. i.e. no direct parent-child relationship
4. To ensure that child has completed exit system call, semaphore exited is used to maintain synchronization.
5. Finally exit status is returned using tid_arr_exit_status[] array.

Deny write to executables

For this another attribute is stored in thread struct. i.e. (struct file* executable), which stores file pointer to the file which is executed by the thread (if any). Now this pointer is set in the load function (which is process.c). After setting this file pointer, file_deny_write() is called on this file pointer, which is a predefined function. Now file_close removed from the end of load function. The file is finally closed in exit system call. In file_close(), function file_allow_write function is called which works with inode_deny_write function which already maintains deny write count. So synchronization in this task is maintained automatically by given kernel code.