1 Problem 1

The user program for this part is *test_program_1.c*.

user/test_program_1.c

```
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

int
main(int argc, char** argv) {

 if (argc != 2) {
     fprintf(2, "Usage: %s <string>\n", argv[0]);
     exit(1);
 }

 if (echo_simple(argv[1]) < 0) {
   fprintf(2, "%s: echo_simple failed\n", argv[0]);
   exit(1);
 }

 exit(0);
}
```

The program invokes the custom syscall *echo_simple(char*)*, which takes as argument:

- The single string passed to the program.

In *kernel/sysproc.c*, the *sys_echo_simple()* function is implemented as follows:

- It assumes a maximum length for the string, say *MXLEN* (I use $MXLEN = 100$ throughout this assignment).
- It uses *argstr()* to get the string argument passed to the syscall from the user space and prints it.

2 Program 2

The user program for this part is *test_program_2.c*.

user/test_program_2.c

```
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

int
main(int argc, char** argv) {

 if (echo_kernel(argc-1, argv+1) < 0) {
   fprintf(2, "%s: echo_kernel failed\n", argv[0]);
   exit(1);
 }

 exit(0);
}
```

The program invokes the custom syscall *echo_kernel(int, char**)*, which takes as arguments:

- The number of strings passed to the *test_program_2.c* (excluding the name of the program itself).
- The array of strings/char* passed to the program.

In *kernel/sysproc.c*, the *sys_echo_kernel()* function is implemented as follows:

- It assumes a maximum length *MXLEN* for each string of the array.
- It uses *argint()* to get the number of arguments and if the number of arguments is > 0, also gets the base address of the array of strings through *argaddr()* from the user space.
- A loop is used to get the strings of the array. Firstly, the address corresponding to the first character of the string is obtained from the base address via *argaddr()*, then subsequently the string itself is obtained using *argstr()*. The base address is incremented by *sizeof(char*)* at the end of every iteration.
- The strings are joined by a space and printed.

3 Problem 3

The user program for this part is *trace.c*.

user/trace.c

```
#include "kernel/param.h"
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

int
main(int argc, char *argv[])
{
int i;
char *nargv[MAXARG];

if(argc < 3 || (argv[1][0] < '0' || argv[1][0] > '9')) {
 fprintf(2, "Usage: %s mask command\n", argv[0]);
 exit(1);
}

if (trace(atoi(argv[1]), 0) < 0) {
 fprintf(2, "%s: trace failed\n", argv[0]);
 exit(1);
}

for(i = 2; i < argc && i < MAXARG; i++){
 nargv[i-2] = argv[i];
}
exec(nargv[0], nargv);

exit(0);
}
```

The program invokes the custom syscall *trace(int, int)*, which takes as arguments:

- The mask representing which syscalls to trace.
- The mode: 1 - represents tracing the syscall arguments also, 0 - represents not tracing syscall arguments. For this problem, we set it to 0.

In *kernel/sysproc.c*, the *sys_trace()* function is implemented as follows:

- The mask and mode are obtained from the userspace through *argint()*.
- The custom added attributes *trace_mask* and *print_args* in the proc structure are set to the mask and mode respectively.

Now in *kernel/syscall.c*, the following changes are added:

- Array of char* *syscall_names* is added which stores the name of the syscall for each syscall ID.
- After the syscall handler function returns in *syscall()*, the process's PID, name (from *syscall_names*) and the return code (from *p→trapframe→a0*) is printed if the bit corresponding to the ID of the syscall is set in the process's *trace_mask*.

NOTE: To avoid the *trace_mask* and *print_args* values in the proc structure to be carried over from this process to other unrelated processes, I initialise those attributes to zero in *kernel/proc.c/allocproc()*.

4 Problem 4

The user program for this part is *trace_ext.c*.

user/trace_ext.c

```
#include "kernel/param.h"
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

int
main(int argc, char *argv[])
{
int i;
char *nargv[MAXARG];

if(argc < 3 || (argv[1][0] < '0' || argv[1][0] > '9')) {
 fprintf(2, "Usage: %s mask command\n", argv[0]);
 exit(1);
}

if (trace(atoi(argv[1]), 1) < 0) {
 fprintf(2, "%s: trace failed\n", argv[0]);
 exit(1);
}

for(i = 2; i < argc && i < MAXARG; i++){
 nargv[i-2] = argv[i];
}
exec(nargv[0], nargv);

exit(0);
}
```

The program invokes the custom syscall *trace(int, int)* again, which takes as arguments:

- The mask representing which syscalls to trace.
- The mode: 1 - represents tracing the syscall arguments also, 0 - represents not tracing syscall arguments. For this problem, we set it to 1.

In *kernel/sysproc.c*, the *sys_trace()* function is the same as for problem 3.

Now in *kernel/syscall.c*, the following additional changes are added:

- Function *print_joined_str_array(uin64, int)* is defined that takes in the base address of an array of strings and the maximum number of strings in the array, and prints the strings separated by a single space on a single line.
- Function *print_syscall_args(int, struct proc*)* is defined that takes the current syscall's ID and pointer to current process's proc structure. It uses a switch-case control structure to get the arguments specifically for each syscall as seen from the prototypes declared in *user/user.h*, and uses utility functions *argint(), argaddr(), argstr()* defined in *kernel/syscall.c* itself for the same.

- In *syscall()* function, before the corresponding syscall handler function is called, if the syscall's bit is set in the process's *trace_mask* and *print_args* is set, the *print_syscall_args()* function is called to trace the arguments to the syscall.

5 Program 5

The user program for this part is *test_program_5.c*.

user/test_program_5.c

```
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"
#include "kernel/processinfo.h"

int
main(int argc, char** argv) {

 if (argc != 1) {
     fprintf(2, "Usage: %s\n", argv[0]);
     exit(1);
 }

 struct processinfo pi;
 if (get_process_info(&pi) < 0) {
   fprintf(2, "%s: get_process_info failed\n", argv[0]);
   exit(1);
 }

 printf("Process ID -> %d\n", pi.pid);
 printf("Process Name -> %s\n", pi.name);
 printf("Memory Size -> %d Bytes\n", pi.sz);

 exit(0);
}
```

The program invokes the custom syscall *get_process_info(struct processinfo*)*, which takes as argument:

- The pointer to the processinfo structure (defined in *kernel/processinfo.h*) in which the current process's info needs to be stored from the kernel space.

In *kernel/sysproc.c*, the *sys_get_process_info()* function is implemented as follows:

- The address to the processinfo structure is obtained from the user space through *argaddr()*.
- The current process's PID, memory size (in Bytes) and name derived from the proc structure is stored into a temporary *struct processinfo* object.
- *copyout()* function is used to copy the processinfo structure's content from the kernel space to the user space.