**Q1)**

int allbroadcast(const char \*commands[]) {

int size = (sizeof(commands) / sizeof(char\*)) - 1; // count of commands

struct pollfd \*fds = malloc(size \* sizeof(struct pollfd)); // poll struct

int pipes[size]; // pipe array

char buf[100]; // buffer for read and write

for (int i = 0; i<size; i++) {

pipe(pipes[i]); // create pipe

fds[i]->fd = pipes[i]; // set fds argument for poll() function

fds[i]->events = POLLIN; // warn me when there is data to read

}

if (fork() != 0) { // On parent process start listening pipes

while (1) {

poll(fds, size, -1); // wait for some data to read on any pipe

int ind;

for (ind=0; ind<size; ind++) { // find the index of sender pipe

if (fds[ind]->revents == POLLIN)

break;

}

int n;

memset(buf, 0, 100); // clear the buffer

while ((n = read(pipes[ind], buf, 100)) == 100){ // read the pipe

for (int i=0; i<size; i++){ // write the data to all pipes except sender

if (i != ind)

write(pipes[i], buf, 100);

}

memset(buf, 0, 100); // clear again

}

for (int i=0; i<size; i++){ // write for the last time

if (i != ind)

write(pipes[i], buf, n);

}

}

return 0;

}

else {

for (int i=0; i<size; i++) { // create process for each command

if (fork() == 0) {

dup2(pipes[i], 0); // direct stdin ..

dup2(pipes[i], 1); // and stdout to pipe

if (fork() == 0){ // fork again to execute command

exec(commands[i]);

}

else { // listen stdin (our pipe) and doSomething with input

while (1) {

char buf[100];

read(0, buf, 100);

doSomething();

}

return 0;

}

}

}

return 0;

}

}

**Q2)**

// struct for wait and signal communication

struct wstr {

pthread\_cond\_t cond;

short\* set;

}

class CVSet{

pthread\_mutex\_t \*mutex;

int size;

vector<struct wstr\*> vect;

CVSet(int i, pthread\_mutex\_t\* m){ size = i; mutex = m; }

lock() { pthread\_mutex\_lock(mutex); }

unlock() { pthread\_mutex\_unlock(mutex); }

wait(short\* set) {

// create a new condition variable

pthread\_cond\_t c = INITIALIZE;

// create a new struct wstr element and push\_back to vector

struct wstr elem;

elem.cond = c;

elem.set = set;

vect.push\_back(&elem);

// wait for created condition variable and release mutex (mutex is locked before wait)

pthread\_cond\_wait(&c, mutex);

}

signal(short var) {

// look at the vector for appropriate waiter and signal

pthread\_mutex\_lock(mutex);

for each variable (struct wstr\*) w in vect {

if (w->set[var]){

pthread\_cond\_signal(w->cond);

erase w from vect

break;

}

}

pthread\_mutex\_unlock(mutex);

}

}

**Q3)**



**Q4)**

***a)***

#include <linux/sched.h>

struct task\_struct \*p;

for (p=&init\_task; (p=next\_task(p)) != &init\_task; ){

int pid = (int) p->pid;

int ppid = (int) ( (p != &init\_task) ? (p->parent->pid) : 0 );

// ...

}

***b)***

#include <sys/proc.h>

#include <os/pid.c>

struct proc\_t \*p;

for (p = practive; p != NULL; p = p->p\_next){

int pid = (int) p->p\_pidp->pid\_id;

int ppid = (int) p->p\_ppid;

//..

}

***c)***

void ptree(struct task\_struct \*p) {

list\_head \*children = &p->children;

list\_head \*child;

struct task\_struct \*childTask;

list\_for\_each(child, children) {

childTask = list\_entry(child, struct task\_struct, children);

doSomething(childTask);

ptree(childTask);

}

}

***d)***

void ptree(proc\_t \*p) {

proc\_t \*child = p->p\_child; // first child

for (; child != NULL; child = child->p\_sibling) {

doSomething(child);

ptree(child);

}

}

***e)***

long \_do\_fork(...) is the main routine of fork process.

Defined in <kernel/fork.c> file.

// system call entry

SYSCALL\_DEFINE0(fork)

{

#ifdef CONFIG\_MMU

return \_do\_fork(SIGCHLD, 0, 0, NULL, NULL, 0);

#else

return -EINVAL;

#endif

}

***f)***

int64\_t forksys(int subcode, int flags) is the routine for fork system call.

Defined in <os/fork.c> file

***g)***

struct task\_struct\* myfork(struct task\_struct \*p) {

// create new task\_struct for child process

struct task\_struct \*child = kmalloc(sizeof(struct task\_struct));

// copy the task\_struct of parent

memcpy(child, p, arch\_task\_struct\_size);

// task\_struct's are equal now

// copy credentials but we need to change keyring values of child process

child->creds = kmalloc(sizeof(creds));

copy the content creds of p to child->creds

key\_get(child->session\_keyring);

key\_get(child->process\_keyring);

key\_get(child->thread\_keyring);

key\_get(child->request\_key\_auth);

// pid and parent values must be set

child->pid = get\_task\_pid();

child->parent = p; child->real\_parent = p;

// children and sibling lists must be cleared

INIT\_LIST\_HEAD(child->children);

INIT\_LIST\_HEAD(child->sibling);

foreach child\_of\_parent in p->children:

add child\_of\_parent to child->sibling

add child to p->children

// copy the content of file descriptor table

child->files = dup\_fd(p->files);

// copy the memory mappings

child->mm = dup\_mm(p);

// also we need to copy signals, signal handlers, namespaces, etc.

// as written in the kernel/fork.c LINE 1487

// creation struct of task is completed. now it is time to start process

wake\_up\_new\_task(child);

// child is created and started

return child;

}