CS3310 – Operating Systems Lab

Assignment Report – Lab 4

SURE MANOJ KUMAR – CS12B028

Exercise 1:

(i) – Allocating an array of Env's of NENV Size

envs=(struct Env*)boot_alloc(NENV*sizeof(struct Env));

(ii) – Mapping to user Read-only at UENVS

boot_map_region(kern_pgdir, UENVS, NENV * sizeof(struct Env), PADDR(envs), PTE_U — PTE_P); *//Read-only permissions "PTE_U — PTE_P"*

After writing the above code,when I ran the kernel, the function check_kern_pgdir() gave succeeded as result.

Exercise 2:

(i) – env_init()

adding the environments to the env_free_list linked list in the same order as of *envs* array and marking them as free.

```
int i;
env_free_list=NULL;
for(i = NENV - 1 ; i >= 0; i - - ) //we have to come down from the array to maintain the // order
{
envs[i].env_link=env_free_list; //Prepending to the linkedlist env_free_list
envs[i].env_status=ENV_FREE; //marking the environment as free
env_free_list=&envs[i]; //updating the head of the linked list
}
```

(ii) – env_setup_vm()

Allocating a page directory and initialising the kernel portion for the environment

```
p->pp_ref++; //p is of type struct PageInfo
e->env_pgdir=page2kva(p); //allocating a page directory for the environment
for(i=PDX(UTOP);i<NPDENTRIES;i++) //initialising the kernel portion
e->env_pgdir[i]=kern_pgdir[i];
e->env_pgdir[PDX(UVPT)] = PADDR(e->env_pgdir) — PTE_P — PTE_U;
//setting the permissions
```

(iii) – Region_alloc()

Allocating and mapping physical memory for the environment

//making va and (va+len) page – aligned.
uint32_t a= ROUNDDOWN((uint32_t)va,PGSIZE); *//rounding va down*
uint32_t b= ROUNDUP((uint32_t)(va+len),PGSIZE); *// rounding (va+len) up*
struct PageInfo* p;
uint32_t i;
for(i=a;i<b;i++)
{
p = page_alloc(0); *// allocating a page*
if (p == NULL) *// panicking if the page can not be allocated*
panic("region_alloc failed");
page_insert(e->env_pgdir,p,(void *)i,PTE_U—PTE_W);
//allocating this page for under e->env_pgdir with read-only permissions.
}
(iv) – load_icode()

parsing elf binary and loading it into user environment space of new environment

lcr3(PADDR(e->env_pgdir));
struct Elf* elfheader = (struct Elf*)binary;
struct Proghdr* ph1,*ph2;
ph1=(struct Proghdr*)(elfheader->e_phoff+binary);
ph2=ph1+elfheader->e_phnum;
e->env_tf.tf_eip=elfheader->e_entry;
char* start,*content;
for(;ph1<ph2;ph1++)
{
region_alloc(e,(void*)ph1->p_va,ph1->p_memsz);
memmove((void*)ph1->p_va,(char*)binary+ph1->p_offset,ph1->p_filesz);
memset((void*)(ph1->p_va+ph1->p_filesz),0,ph1->p_memsz - ph1->p_filesz);
}
lcr3(PADDR(kern_pgdir));
region_alloc(e,(void*)(USTACKTOP-PGSIZE),PGSIZE);
(v) – env_create()

allocating an environment an loading the elf binary into it.
struct Env* env;
int a =env_alloc(&env, 0); *//env_alloc returns 0 on successfull allocation*
if (a != 0)
panic("env_create failed."); *//panic if env_alloc returns non-zero(failed)*

env->env_parent_id=0; *//setting parent's id to zero.*
env->env_type=type;
load_icode(env,binary,size); *//loading the elf binary into the environment.*
(vi) – env_run()

running given environment in user mode.
if(curenv!=e)
{
//changing the 'curenv' status to runnable from running if it was already
//in running state.
if((curenv!=NULL) && curenv->env_status==ENV_RUNNING)
curenv->env_status=ENV_RUNNABLE;
curenv=e;
curenv->env_status=ENV_RUNNING; *//setting env_status to env_running*
curenv->env_runs++; //updating the env_runs
lcr3(PADDR(curenv->env_pgdir)); *//switching to the env_pgdir address space*
}
env_pop_tf(&curenv->env_tf); *//popping the current environment's trap frame.*
Exercise 3:
Reading Part.
Exercise 4:
_alltraps:
pushl %ds
pushl %es
pushal
movl $GD_KD, %eax *//loading GD_KD into es and ds*
movw %ax, %es
movw %ax, %ds
pushl %esp *//pushl esp and calling trap*
call trap
trap_init() - initialising the idt to point to each of the entries in trapentry.S
int i;
//initialising the idt
for (i = 0; i <= 48; i++)
SETGATE(idt[i], 0 , GD_KT, handlers[i], 0);
SETGATE(idt[T_BRKPT], 0, GD_KT, handlers[T_BRKPT], 3); *//for break point test to //work*
SETGATE(idt[T_SYSCALL], 0, GD_KT, handlers[T_SYSCALL], 3);*//added for exercise 7*

Trap numbers from 8 to 15 generate an error code and others do not.

So,for them we push the corresponding number and for others we push zero in place of error code.

Exercise 5 & 6 :

trap_dispatch(): call the appropriate function depending on the trapno in the trapframe.

int retval = 0;
switch(tf->tf_trapno)
{
case T_PGFLT: *//pagefault*
page_fault_handler(tf);
return;
case T_BRKPT: *//break point – for exercise 6*
monitor(tf); *//monitor function is already implemented .*
return;
case T_SYSCALL: *//system call – for exercise 7*
retval= syscall(tf->tf_regs.reg_eax,tf->tf_regs.reg_edx,tf- >tf_regs.reg_ecx,tf->tf_regs.reg_ebx,tf->tf_regs.reg_edi,tf->tf_regs.reg_esi);
tf->tf_regs.reg_eax = retval;
return;
default: *//we do not handle any other trap*
print_trapframe(tf);
if (tf->tf_cs == GD_KT)
panic("unhandled trap");
else {
env_destroy(curenv);
return;
}
}

Exercise 7:

sys_cputs():
char *va;
pte_t *p;
//checking whether the user has permissions or not
for (va = (char *)s ; va < s + len; va+=PGSIZE){
p = pgdir_walk(curenv->env_pgdir, (void *)va, 0);
if (p && (*p & PTE_U) && (*p & PTE_P))
continue;
env_destroy(curenv); *//if user dont have necessary permissions we destroy*

return; //*that environment and return ,otherwise print the string(as given*
)
}
syscall.c()
int retval = 0;
//*calls the appropriate function based on the syscallno*
switch(syscallno){
case SYS_cputs:
user_mem_assert(curenv, (void *)a1, a2, PTE_U — PTE_P);
//*the above is for checking if the curenv has the required permissions*
//*or not. Asked in Exercise 9.*
sys_cputs((char *)a1, a2);
retval = a2;
break;
case SYS_cgetc:
retval = sys_cgetc();
break;
case SYS_getenvid:
retval = sys_getenvid();
break;
case SYS_env_destroy:
retval = sys_env_destroy(a1);
break;
default:
-E_INVAL;
}
return retval;
Exercise 8:
libmain.c
thisenv = (struct Env *)envs + ENVX(sys_getenvid());
//*thisenv point to environment's structure in envs array*
Exercise 9 & 10:
page_fault_handler(): panicking if the page fault occurs in kernel mode.
if ((tf->tf_cs & 3) != 3) //*checking the last two bits of tf_cs*
panic("kernel page fault");
user_mem_check():
//*checking if the environment has the permission to read [va,va+len) with*
//permissions perm — PTE_P
uint32_t ia;
perm —= PTE_P;
for (ia = (uint32_t)va; ia < (uint32_t)va + len; ia++){

```
if (ia > ULIM){
user_mem_check_addr = ia;
return -E_FAULT;
}
pte_t *p = pgdir_walk(env->env_pgdir, (void *)ia, 0);
if (p==NULL){
user_mem_check_addr = ia;
return -E_FAULT;
}
if ((*p & perm) != perm){
user_mem_check_addr = ia;
return -E_FAULT;
}
}
return 0;
```

Question 1:

Because, the hardware does not distinguish errors ,for which exception is called.some times we push the error code and some times we do not .if all the exceptions we delivered to the same handler, we lose this feature.

Question 2:

No, we dont have to do anything to make the code work properly.If the chice is given to the user, then there is a security vulnerability as exceptions are handled with kernel privilages and he can run anything he wants ,which should not happen.It gives int13 because there is a violation in the privilages.

Question 3:

In trap_init() we set the dpl to 3 for T_BRKPT so that user can generate breakpoint trap

and it will generate breakpoint trap ,if the dpl is set to 0,user can not generate it himself and it will give general protection fault.

Question 4:

The mechanisms enforce permissions,they create gate so that user can make system calls through the exceptions.depending on what exceptions we allow for the user to generate(depending on how you set the permissions in trap_init using SETGATE).This is useful as a protection to not execute the malicious code written by user.

Exercise 9 – Question:

Because user process do not have permissions to access the code in lib/libmain.c.