Acknowledgement: Slides drawn heavily from Yeongjin Jiang
Part-A Result

• You should get this OK before start exercise 8

• FAQ
  • What if dumbfork halts?
    • Check if your sched_yield()/env_run() is implemented correctly
    • curenv must set as ENV_RUNNABLE state if it is scheduled out...
  • What if I have a syscall error?
    • Check if your implementation returns the return value of the syscall correctly
    • Check syscall arguments and orders
    • There always be syscalls to SYS_getenvid and SYS_cputs
CAUTION:
You Will See LOTS of Page Faults in Part B

• What should I do if I see a page fault?

• Check information related to the fault
  • Check tf_eip (the origin of the fault)
  • Check fault_va (read cr2, rcr2())
    • You can reason a lot from this address, e.g., 0xcabefbe?
    • If it is 0, a null pointer dereference, check your impl!!!
  • Check error code (user/kernel, read/write, present?)

• Think about why this fault happens???
How Can I Get the Code for User Exec?

• Read `obj/user/xxxx.asm`

• E.g., `dumbfork`:
  
  • You can match `eip` and the source code

```c
void duppage(envid_t dstenv, void *addr)
{
  800040:  55      push %ebp
  800041:  89 e5    mov %esp,%ebp
  800043:  56      push %esi
  800044:  53      push %ebx
  800045:  83 ec 20  sub $0x20,%esp
  800048:  8b 75 08  mov 0x8(%ebp),%esi
  80004b:  8b 5d 0c  mov 0xc(%ebp),%ebx

  int r;

  // This is NOT what you should do in your fork.
  if ((r = sys_page_alloc(dstenv, addr, PTE_P|PTE_U|PTE_W)) < 0)
    80004e:  c7 44 24 00 07 00 00  movl $0x7,0x8(%esp)
  800055:  00      
  800056:  89 5c 24 04  mov %ebx,0x4(%esp)
  800058:  89 34 24  mov %esi,(%esp)
  80005d:  e8 81 0d 00 00  call 800de3 <sys_page_alloc>
  800062:  85 c0    test %eax,%eax
  800064:  79 20    jns 800086 <duppage+0x46>

  panic("sys_page_alloc: %e", r);
  800066:  89 44 24 0c  mov %eax,0xc(%esp)
  80006a:  c7 44 24 08 a0 12 80  movl $0x8012a0,0x8(%esp)
  800071:  00      
  800072:  c7 44 24 04 20 00 00  movl $0x20,0x4(%esp)
  800075:  00      
  800077:  c7 44 24 b3 12 80 00  movl $0x8012b3,(%esp)
  80007c:  e8 24 02 00 00  call 80002a <panic>
  800081:  8f 44 24 0b 07 00 00  if ((r = sys_page_map(dstenv, addr, 0, UTEMP, PTE_P|PTE_U|PTE_W)) < 0)
    800086:  c7 44 24 10 07 00 00  movl $0x7,0x10(%esp)
  80008e:  00      
  800090:  c7 44 24 0c 00 00 40  movl $0x400000,0xc(%esp)
  800099:  00      
```
Part B: Copy-on-Write Fork

• We will implement an efficient, copy-on-write fork
  • Purely in user-level with user-level page fault handler
  • Will use syscalls that we implemented in Exercise 7
    • sys_exofork, sys_env_set_status, sys_page_alloc, sys_page_map, sys_page_unmap

• DO NOT implement lib/fork.c the same as user/dumbfork.c
  • Dumbfork does not do Copy-on-write
  • fork() should not copy any of memory page
    • It only copies VA-to-PA mappings (page table entries)
Part B: Copy-on-Write Fork

- We will implement
  - User-level exception handling (page fault handler) (Exercise 8—11)
  - Copy-on-write fork() (Exercise 12)
How Page Fault Works (in Lab 3)?

• 1. User program generates a fault
  • E.g., struct Env *e = NULL;
    • e->env_id; (Null pointer dereference)

• 2. trapentry.S, _alltraps, trap(), and then trap_dispatch()
  • Will call the page_fault_handler(tf)

```c
switch (tf->tf_trapno) {
  case T_PGFLT:
    { return page_fault_handler(tf); }
}
```
How Page Fault Works (in Lab 4)?

• 3. in page_fault_handler(tf)

• Handle user page fault in user space
Page Fault

- A memory access fault caused by:
  - Having no Page Directory Entry or Page Table Entry
  - Insufficient permission to access the memory
    - PTE & PTE_U == 0, accessed by user process (Ring 3)
    - PTE & PTE_W == 0, attempted write access
    - PTE & PTE_P == 0, not available
  - Invalid physical address for PTE...

- CPU will call page fault trap handler from IDT
  - CR2 will store the fault address
  - Error code will store the cause of violation, P/U/W, etc.

- Execution resumes at the faulting address (re-execute)
Copy-on-Write Page Fault Handler

- **Copy-on-write fork**
  - Make pages read-only, mark PTE_COW (in AVL..), and share.
  - Any write to COW page will generate a page fault

- **On fault**
  - CR2 will store the faulting address
  - Error code will say: write on read-only page

- **TODO**
  - Retrieve PTE (using the value in CR2)
  - Check if PTE & PTE_COW == 1
  - Allocate a new physical page, copy the content, and update PTE
    - PTE_W!
JOS Page Fault Workflow (Kernel)

• 1. Fault (user/somewhere.c)
• 2. CPU runs trap handler
• 3. _alltraps (kern/trapentry.S)
• 4. trap (kern/trap.c)
• 5. trap_dispatch (kern/trap.c)
• 6. page_fault_handler (kern/trap.c)
JOS User Fault Handling Workflow

• 6. page Fault Handler (kern/trap.c)
• 7. _pgfault_upcall (lib/pfentry.S)
  • 7-1. _pgfault_handler (lib/pgfault.c)
• 8. return to the faulting instruction
• 9. Resume!

• Blue: Program execution in user
• Purple: Fault handling in user
• Red: Fault handling in kernel

Exercise 9
Exercise 8/10
Exercise 12
Exercise 11
Exercise 8

• Implement `sys_env_set_pgfault_upcall` (kern/syscall.c)
  • Kernel page fault handler will call `_pgfault_upcall`
    • `curenv->env_pgfault_upcall`

```
static int
sys_env_set_pgfault_upcall(envid_t envid, void *func)
```

• Get the Env of `envid`, and set its `env_pgfault_upcall = func`

• 6. `page_fault_handler` (kern/trap.c)
• 7. `_pgfault_upcall` (lib/pfentry.S)
  • 7-1. `_pgfault_handler` (lib/pgfault.c)
Exercise 8

• How can we get an Struct Env * from envid?
• Use envid2env

```c
int
envid2env(envid_t envid, struct Env **env_store, bool checkperm)
```

• How?

```c
struct Env *e = NULL;
if (envid2env(envid, &e, 1) < 0)
    return -E_BAD_ENV;
```

• Checkperm will check if the env is
  • Current env or
  • A child env of the current env

  e->env_pgfault_upcall = func
Exercise 9: page_fault_handler (kern/trap.c)

- What should it do?
  - Execute curenv->env_pgfault_upcall (set by user via syscall)
    - 6. page_fault_handler (kern/trap.c)
    - 7. _pgfault_upcall (lib/pfentry.S)
  - tf->tf_eip = (uintptr_t) curenv->env_pgfault_upcall;

- Requirement?
  - Let env_pgfault_upcall returns to the faulting instruction
  - Restore all CPU context after handing the fault
    - 7. _pgfault_upcall (lib/pfentry.S)
      - 7-1. _pgfault_handler (lib/pgfault.c)
    - 8. return to the faulting instruction
    - 9. Resume!
Exercise 9: page_fault_handler (kern/trap.c)

• How can we execute kernel -> user -> user??
  • 6. page_fault_handler (kern/trap.c)
  • 7. _pgfault_upcall (lib/pfentry.S)
    • 7-1. _pgfault_handler (lib/pgfault.c)
  • 8. return to the faulting instruction
  • 9. Resume!

• Trapframe *tf stores the context at the time of fault
  • Create UTrapframe *utf to deliver this context to the user-level handler

Copy Context
eip, fault_va, err, regs, esp, etc
Exercise 9: page_fault_handler (kern/trap.c)

Exception stack

<table>
<thead>
<tr>
<th>Trapframe *tf</th>
</tr>
</thead>
<tbody>
<tr>
<td>tf_esp</td>
</tr>
<tr>
<td>tf_eflags</td>
</tr>
<tr>
<td>Others..</td>
</tr>
<tr>
<td>tf_eip</td>
</tr>
<tr>
<td>tf_err</td>
</tr>
</tbody>
</table>

We want to do:
1) Copy Trapframe information as Utrapframe
2) Call curenv->env_pgfault_upcall
Exercise 9: How can we run the function `curenv->env_pgfault_upcall()` in Ring 3?

- Via `iret`, `env_pop_tf()`
  - Set the `tf_eip = curenv->env_pgfault_upcall;`
  - Set the `tf_esp = addr_of_UTrapframe;`

---

**Trapframe *tf**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>tf.esp</strong></td>
<td>ESP at the time of fault</td>
</tr>
<tr>
<td><strong>tf.eflags</strong></td>
<td>EFLAGS at the time of fault</td>
</tr>
<tr>
<td><strong>Others..</strong></td>
<td></td>
</tr>
<tr>
<td><strong>tf.eip</strong></td>
<td>Points to the faulting instruction</td>
</tr>
<tr>
<td><strong>tf.err</strong></td>
<td>Error code of the fault</td>
</tr>
<tr>
<td><strong>Registers..</strong></td>
<td>Registers at the time of fault</td>
</tr>
<tr>
<td><strong>UTrapframe?</strong></td>
<td></td>
</tr>
</tbody>
</table>

**We want to do:**
1) Copy Trapframe information as Utrapframe
2) Call `curenv->env_pgfault_upcall`
Use UTrapframe to Transfer Execution Context

• Create Utrapframe, and deliver that to env_pgfault_upcall!
  • Copy necessary information to handle the page fault and return back.

<table>
<thead>
<tr>
<th>Trapframe *tf</th>
<th>UTrapframe *utf</th>
</tr>
</thead>
<tbody>
<tr>
<td>tf.esp</td>
<td>utf.esp</td>
</tr>
<tr>
<td>tf.eflags</td>
<td>utf.eflags</td>
</tr>
<tr>
<td>Others..</td>
<td>utf.eip</td>
</tr>
<tr>
<td>tf.eip</td>
<td>utf.err</td>
</tr>
<tr>
<td>tf.err</td>
<td>utf_fault_va</td>
</tr>
<tr>
<td>Registers..</td>
<td></td>
</tr>
</tbody>
</table>

**addr_ofUtf:** Points to the faulting instruction
**utf_fault_va:** Faulting address (from CR2)
Where Do We Store UTrapframe?

- **UXSTACK**

1 page in size

- Trapframe *tf
  - tf_sp
  - tf_eflags
  - Others...
  - tf_eip
  - tf_err
  - Registers...

- UXSTACK
  - Utrapframe
  - fault_va
  - err
  - utf_eip...

- Stack used by fault handling
- Stack used by regular execution

- Kernel
- Others
- UXSTACK
- EMPTY
- USTACK
- Free...
- Heap
- Global int counter;
- Program
Exercise 9-1
Copy Utrapframe from Trapframe

• A) Create UTrapframe

```c
struct UTrapframe utf;
utf.utf_fault_va = fault_va;
utf.utf_err = tf->tf_err;
utf.utf_regs = tf->tf_regs;
utf.utf_eip = tf->tf_eip;
utf.utf_eflags = tf->tf_eflags;
utf.utf_esp = tf->tf_esp;
```
Exercise 9-1
Copy UTrapframe from Trapframe

• B) Put UTrapframe in UXSTACK

• Two cases
  • If this is a new exception
  • UXSTACKTOP–sizeof(struct UTrapframe)
Exercise 9-1
Copy UTrapframe from Trapframe

• B) Put UTrapframe in UXSTACK

• Two cases
  • If this is a new exception
    • UXSTACKTOP–sizeof(struct UTrapframe)
  • If this is a nested exception
    • utf_esp =
      • tf_esp–4–sizeof(struct UTrapframe)

• How to distinguish each case?

```c
if (ROUNDUP(tf->tf.esp, PGSIZE) == UXSTACKTOP) {
```
Exercise 10: _pgfault_upcall

• 1) calls _pgfault_handler(utf)
  • _pgfault upcall is called via iret (tf_eip)
  • tfEsp must point to the exception stack (near UXSTACKTOP)
  • tfEsp must point to the address of utf

```
_pgfault_upcall:
    // Call the C page fault handler.
    pushl %esp          // function argument: pointer to UTF
    movl _pgfault_handler, %eax
    call *%eax
    addl $4, %esp       // pop function argument
```

UXSTACK

utf_esp
utf_eflags
utf_eip
Registers...
utf_err
utf_fault_va
Addr of utf

%esp
%esp

UXSTACK
JOS User Fault Handling Workflow

• 6. page_fault_handler (kern/trap.c)
• 7. _pgfault_upcall (lib/pfentry.S)
  • 7-1. _pgfault_handler (lib/pgfault.c)
• 8. return to the faulting instruction
• 9. Resume!

• Blue: Program execution in user
• Purple: Fault handling in user
• Red: Fault handling in kernel
Exercise 10: _pgfault_upcall

- 1) calls _pgfault_handler(utf)
  - _pgfault upcall is called via iret (tf_eip)
  - tf_esp must point to the exception stack (near UXSTACKTOP)
  - tf_esp must point to the address of utf

```asm
_pgfault_upcall:
    // Call the C page fault handler.
    pushl %esp          // function argument: pointer to UTF
    movl _pgfault_handler, %eax
    call *%eax
    addl $4, %esp       // pop function argument
```

UXSTACK

- utf_esp
- utf_eflags
- utf_eip
- Registers...
- utf_err
- utf_fault_va

Addr of utf
Exercise 10: Return to the Faulting Instruction

• UTrapframe stores the original execution context
• _pgfault_upcall should restore all context
  • General purpose registers (eax, edx, ecx, ebx, esi, edi, ebp)
  • EIP
  • EFLAGS
  • ESP

UXSTACK

utf_esp
utf_eflags
utf_eip
Registers..
utf_err
utf_fault_va
%esp
Restoring Context

• General purpose registers
  • popa will pop all registers...

• Assembly
  • add $8, %esp
  • popa
Restoring Context

• General purpose registers
  • popa will pop all registers...

• Assembly
  • add $8, %esp
  • popa
Restoring Context

• General purpose registers
  • popa will pop all registers...
  • eax, edx, ecx, ebx, esi, edi, and ebp

• Assembly
  • add $8, %esp
  • popa

You cannot overwrite the values in those registers after doing this...
Restoring EFLAGS

- POPF
  - add $4, %esp
  - popf
Restoring EFLAGS

• POPF
  • add $4, %esp
  • popf
Restoring EFLAGS

• POPF
  • add $4, %esp
  • popf

You cannot use arithmetic operations after doing this.. Because doing such will change EFLAGS!
Restoring ESP

• LEA (Load Effective Address)
  • lea 4(%esp), %esp
  
• C-style
  • esp = &(*(esp+4))
  • Interpretation
    • 4(%esp) means esp[4] or *(esp+4)
    • lea means getting the address of the operand
    • &esp[4] or &(*(esp+4))
  • Result: esp += 4

• This will not change EFLAGS!
How to Restore EIP?

• In x86, two ways
  • Call/jmp
    • mov $0x8048444, %eax
    • Call *%eax
    • Jmp *%eax
    • But we cannot use general purpose registers...
  • RET
    • Interpretation: ret == pop %eip
      • f = *esp
      • esp += 4
      • f();
    • We can put utf_eip right below utf_esp
    • Why???
How to Restore EIP?

• In x86, two ways
  • Call/jmp
    • mov $0x8048444, %eax
    • Call *%eax
    • Jmp *%eax
    • But we cannot use general purpose registers...
  • RET
    • Interpretation: ret == pop %eip
      • f = *esp
      • esp += 4
      • f();
  • We can put utf_eip right below utf_esp
  • Why???
UXSTACK vs USTACK

UXSTACK

%esp
utf_esp
utf_eflags
utf_eip
Registers..
utf_err
utf_fault_va

%esp

pop %esp (%esp will get the value of utf_esp)
lea -4(%esp), %esp (%esp -= 4)
ret

%ebp

ARG 2 = x-1

%esp

MY_ARG1 (5)
Return Addr
Saved EBP
Saved ESI
Saved EBX
UXSTACK vs USTACK

UXSTACK

%esp
utf_esp
utf_eflags
utf_eip
Registers..
utf_err
utf_fault_va

USTACK

%ebp
%esp

Think about how we can put Utf_eip to there...

pop %esp (esp will get the value of utf_esp)
lea -4(%esp), %esp (esp -= 4)
ret
Exercise 11: Finish set_pgfault_handler() (lib/pgfault.c)

• 1) Allocate a page at [UXSTACKTOP-PGSIZE, UXSTACKTOP]
  • To store UTrapframe!
  • Use sys_page_alloc()

• 2) Set env_pgfault_upcall
  • Via syscall, sys_env_set_pgfault_upcall!

• After finishing this (correctly), you should get OKs upto
  • faultallocaclbad
Some Debugging Tips

• Unexpected user_mem_check fails
  • Check your implementation for user_mem_assert

• Why are there 3 faults in faultalloc?
  • Faultalloc reads 2 bad addresses, 0xDeadBeef and 0xCafeBffe

```c
void umain(int argc, char **argv)
{
    set_pgfault_handler(handler);
    cprintf("%s\n", (char*)0xDeadBeef);
    cprintf("%s\n", (char*)0xCafeBffe);
}
```
Some Debugging Tips

• Why are there 3 faults in faultalloc?
  • Faultalloc reads 2 bad addresses, 0xDeadBeef and 0xCafeBffe

// 0xdeadbeef

• Fault at 0xDeadBeef, allocate 0xDeadB000
• Handler writes “this string...”

```c
void
umain(int argc, char **argv)
{
    set_pgfault_handler(handler);
    printf("%s\n", (char*)0xDeadBeef);
    printf("%s\n", (char*)0xCafeBffe);
}
```

```c
void
handler(struct UTrapframe *utf)
{
    int r;
    void *addr = (void*)utf->utf_fault_va;
    printf("fault %x\n", addr);
    if ((r = sys_page_alloc(0, ROUNDDOWN(addr, PGSIZE),
            PTE_R|PTE_U|PTE_W) < 0))
        panic("allocating at %x in page fault handler: %e", addr, r);
    snprintf((char*) addr, 100, "this string was faulted in at %x", addr);
}
```
Some Debugging Tips

• Why are there 3 faults in faultalloc?
  • Faultalloc reads 2 bad addresses, 0xDeadBeef and 0xCafeBffe

```c
void main(int argc, char **argv)
{
    set_pgfault_handler(handler);
cprintf("%s\n", (char*)0xDeadBeef);
cprintf("%s\n", (char*)0xCafeBffe);
}
```

• 0xCafeBffe
  • Fault at 0xCafeBffe, allocate 0xCafeB000
  • Handler writes “this string…”
  • Fault at 0xCafeC000
    • Why? 0xCafeBffe + 2 = 0xCafeC000
    • Not mapped...

```c
void handler(struct UTrapframe *utf)
{
    int r;
    void *addr = (void*)utf->utf_fault_va;
cprintf("fault %x\n", addr);
    if ((r = sys_page_alloc(0, ROUNDDOWN(addr, PGSIZE),
      PTE_P|PTE_U|PTE_W) < 0))
        panic("allocating at %x in page fault handler: %e", addr, r);
    snprintf((char*) addr, 100, "this string was faulted in at %x", addr);
}
```
Handling Multiple Faults

• Page fault can occur during handling a page fault
• In kernel: Panic
• In user:
  • 7. _pgfault_upcall (lib/pfentry.S)
    • 7-1. _pgfault_handler (lib/pgfault.c)

• How?
  • Recursively handle the fault...
JOS Page Fault Workflow (Kernel)

• A-1. Fault (user/somewhere.c)
• A-2. CPU runs trap handler
• A-3. _alltraps (kern/trapentry.S)
• A-4. trap (kern/trap.c)
• A-5. trap_dispatch (kern/trap.c)
• A-6. page_fault_handler (kern/trap.c)
JOS User Fault Handling Workflow

• A-6. page_fault_handler (kern/trap.c)
• A-7. _pgfault_upcall (lib/pfentry.S)
  • A-7-1. _pgfault_handler (lib/pgfault.c, FAULT)
JOS Page Fault Workflow (Kernel)

• B-1. Fault (user/fork.c)
• B-2. CPU runs trap handler
• B-3. _alltraps (kern/trapentry.S)
• B-4. trap (kern/trap.c)
• B-5. trap_dispatch (kern/trap.c)
• B-6. page_fault_handler (kern/trap.c)
JOS User Fault Handling Workflow

• B-6. page_fault_handler (kern/trap.c)
• B-7. _pgfault_upcall (lib/pfentry.S)
  • B-7-1. _pgfault_handler (lib/pgfault.c)
• B-8. return to the faulting instruction
• B-9. Resume to A-7
JOS User Fault Handling Workflow

• B-9. Resume to A-7
• A-7-1. _pgfault_handler (lib/pgfault.c, FAULT)
• A-8. return to the faulting instruction
• A-9. Resume!
Exercise 11

• So you must correctly handle nested page fault to pass “faultalloc”
  • Fault at 0xCafeBffe
    • While handling this fault, the handler generates another fault at 0xCafeC000
      • Handle it!

• This is the case that you need to check if Trapframe stack is in UXSTACK region or not
Exercise 12: Copy-on-Write Fork

• Using user-level page fault handler, implement CoW fork! (lib/fork.c)

• Take a look at the impl. of user/dumbfork.c
  • dumbfork()

```c
void
duppage(envid_t dstenv, void *addr)
{
    int r;
    // This is NOT what you should do in your fork.
    if ((r = sys_page_alloc(dstenv, addr, PTE_P|PTE_U|PTE_W)) < 0)
        panic("sys_page_alloc: %e", r);
    if ((r = sys_page_map(dstenv, addr, 0, UTEMP, PTE_P|PTE_U|PTE_W)) < 0)
        panic("sys_page_map: %e", r);
    mmmove(UTEMP, addr, PG_SIZE);
    if ((r = sys_page_unmap(0, UTEMP)) < 0)
        panic("sys_page_unmap: %e", r);

    // Also copy the stack we are currently running on.
    duppage(env, ROUDDOWN(&addr, PG_SIZE));

    // Start the child environment running
    if ((r = sys_env_set_status(env, ENV_RUNNABLE)) < 0)
        panic("sys_env_set_status: %e", r);

    return env;
}
```
Exercise 12: in duppage()

• Unlike the one in dumbfork, we will not call memmove nor sys_page_alloc
  • We will only call sys_page_map
  • You need to duplicate mappings in a parent env to child env
  • No memory copy! This is copy-on-write!

• Caveat
  • For Read-only mapping, you can map the region read-only in child
  • For Writable mapping, you can map
    • The child as read-only with PTE_COW
    • The parent as read-only with PTE_COW
    • You must change the permission of both pages as PTE_P | PTE_U | PTE_COW
Exercise 12: in duppage()

• Another important tip
  • Making the stack copy-on-write will generate an immediate page fault

• Why?

  // Also copy the stack we are currently running on.
  duppage(envid, ROUNDDOWN(&addr, PGSIZE));

• We make both parent and child to have read-only COW mapping
  • If duppage is called for a writable page

• Program stack will become read-only, and any write of stack, e.g., using local variable, will generate a page fault...

To avoid this problem:
Map the child mapping as Copy-on-Write first. And then, change the parent mapping as Copy-on-Write. Then you will have no problem.
Exercise 12: in fork()

• Don’t forget to
  
• 1. set_page_fault_handler(&pgfault);
  
• 2. Allocate a new page at UXSTACKTOP – PGSIZE
    • For having a separate exception handling stack!
  
• 3. SYS_env_set_pgfault_upcall(envid, thisenv->env_pgfault_upcall);
    • Child must have set its page fault handler to handle CoW
  
• 4. SYS_env_set_status(envid, ENV_RUNNABLE);
    • Make child runnable after finishing the Copy-on-Write fork!
Exercise 12: in pgfault()

• What should we do in the page fault handler to support CoW?
  • COPY ON WRITE

• Yes, we need to copy the faulting page if
  • The access is a write attempt (read attempt is true error on unmapped page)
  • The page is set with PTE_COW == 1
    • Otherwise, it’s a write fault on a true read-only page

• So copy the page if all such condition meets, otherwise, panic!
Exercise 12: in pgfault()

- Then, how can we copy a page?

- 1. allocate a page at the address PFTEMP
- 2. `memcpy(PFTEMP, PTE_ADDR(fault_addr), PGSIZE);`
- 3. `sys_page_map(0, PFTEMP, 0, PTE_ADDR(fault_addr), PTE_U | PTE_P | PTE_W);`
- 4. `sys_page_unmap(PFTEMP);`
Debugging Tips

• Check your traps. Recommend to print out some trap information whenever you got a trap...

```c
static void
trap_dispatch(struct Trapframe *tf)
{
    // Handle processor exceptions.
    // LAB 3: Your code here.

    uint32_t envid;
    if (curenv == NULL) envid = 0;
    else envid = curenv->env_id;
    if (tf->tf_trapno == T_SYSCALL) {
        cprintf("Syscall from %p %s(%p, %p, %p, %p, %p) from "
            "eip %p\n",
            envid,
            stringtbl[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_eip);
    } else if (tf->tf_trapno == T_PGFLT) {
        cprintf("Page fault from %p from va %p eip %p\n",
            envid,
            rcr2(), tf->tf_eip);
    } else {
        cprintf("Trap from %p number %d from eip %p\n",
            envid,
            tf->tf_trapno, tf->tf_eip);
    }
```
Debugging Tips

• Check your traps. Recommend to print out some trap information whenever you got a trap...

```c
static void
trap_dispatch(struct Trapframe *tf)
{
    // Handle processor exceptions.
    // LAB 3: Your code here.

    uint32_t envid;
    if (curenv == NULL) envid = 0;
    else envid = curenv->env_id;
    if (tf->tf_trapno == T_SYSCALL) {
        cprintf("Syscall from %p %s(%p, %p, %p, %p, %p) from "
                "eip %p\n",
               envid,
               stringtbl[tf->tf_regs.reg_eax],
               tf->tf_regs.reg_edx,
               tf->tf_regs.reg_ecx,
               tf->tf_regs.reg_ecx,
               [00000000] new env 00001000
Syscall from 0x1000 SYS_getenvid(0x0, 0x0, 0x0, 0x0, 0x0) from eip 0x800bdf
Syscall from 0x1000 SYS_cputs(0xeefb8d88, 0x27, 0x0, 0x0, 0x0) from eip 0x800b4f
I am the parent. Forking the child...
Syscall from 0x1000 SYS_page_alloc(0x1000, 0xeefb000, 0x7, 0x0, 0x0) from eip 0x800c23
Syscall from 0x1000 SYS_env_set_pgfault_upcall(0x0, 0x8012b9, 0x0, 0x0, 0x0) from eip 0x800d6f
Syscall from 0x1000 SYS_exofork(0x0, 0x8012b9, 0x0, 0x0, 0x0) from eip 0x800f77
[00001000] new env 00001001
Syscall from 0x1000 SYS_page_map(0x0, 0x2000000, 0x100, 0x2000000, 0x805) from eip 0x800c76
Syscall from 0x1000 SYS_page_map(0x0, 0x2000000, 0x0, 0x2000000, 0x805) from eip 0x800c76
Trap from 0x1000 number 32 from eip 0x800c76
```
Debugging Tips

• You will get a page fault (due to Copy-on-Write) immediately after making your stack Copy-on-Write
• This is because duppage will make both virtual page in parent and child set with PTE_COW == 1
• So don’t be surprise, that’s an intended behavior
Debugging Tips

• Make sure you set env_pgfault_upcall for both parent and child

• For parent
  • Run set_pgfault_handler

• For child
  • Run sys_env_set_pgfault_upcall(envid, thisenv->env_pgfault_upcall)
  • Right after forking the child
  • Before changing the child to ENV_RUNNABLE