LAB: SYSTEM CALLS

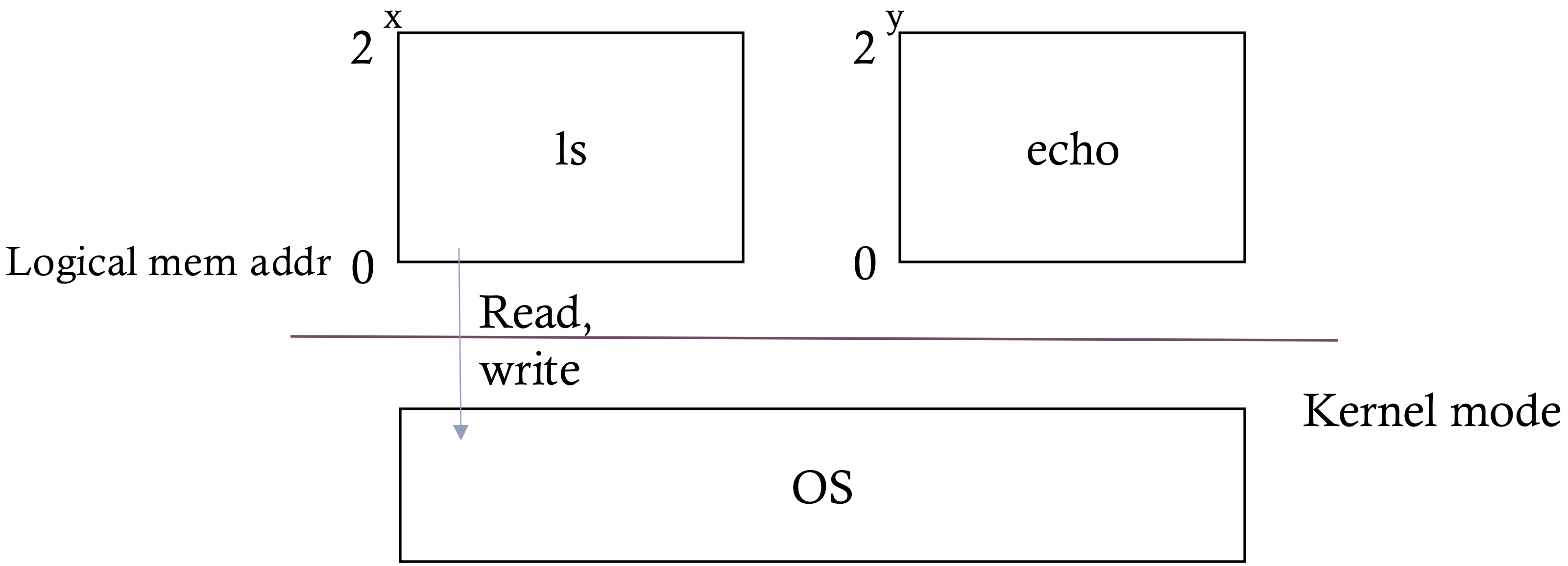
6.1810: Operating System Engineering

UNIX INTERFACE

- Kernel abstracts the CPU
- Kernel manages the use of CPU between the processes
    - Processes do not access the CPU directly
    - Several processes can run simultaneously
- RISC-V
    - If 8 processes are running, the CPU is not used simultaneously
    - 100ms, 100ms, … processes take turns

KERNEL

- Isolates the applications from direct resources by HW support
- HW support
  - User/kernel mode
  - Virtual memory



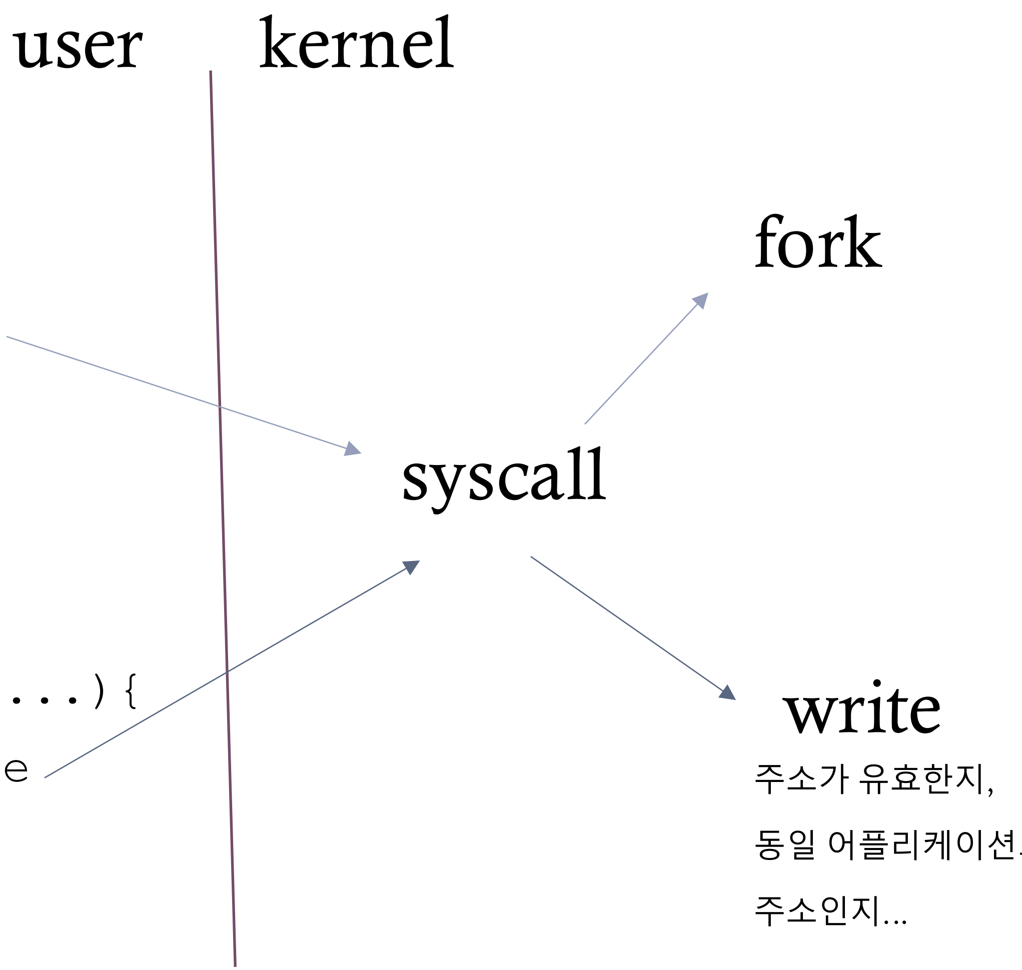
ENTERING KERNEL

ecall $\boxed{(n)}$

Sys call number

```
fork(){
    ecall    sys_fork
}
```

```
write(target, addr, ...){
    ecall    sys_write
}
```



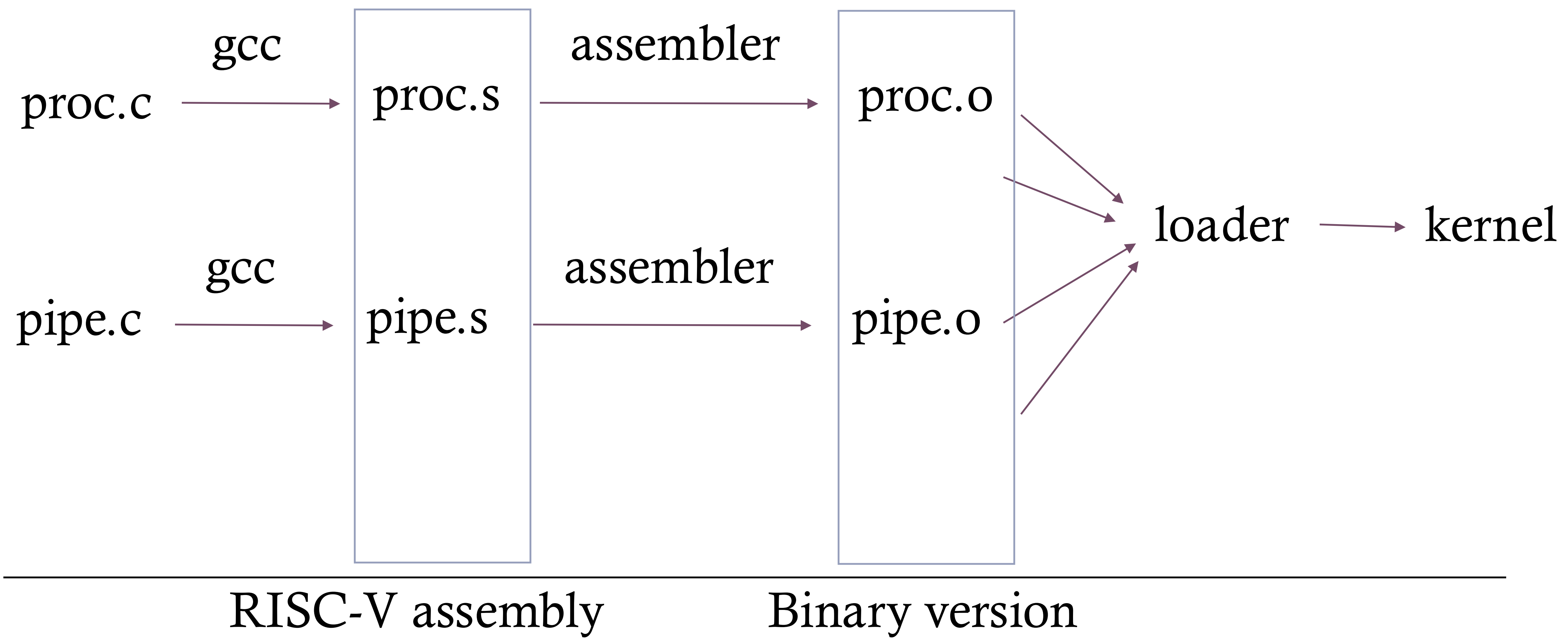
SYSTEM CALL PARAMETER PASSING

- Pass parameters in registers
- Parameters are stored in table(block) and address of block is passed as a parameter in a register
- Pushed into stack

KERNEL -> TRUSTED COMPUTING BASE (TCB)

- Kernel must have no bug
- Kernel must treat processes as malicious

CONSTRUCTION OF KERNEL (MAKEFILE)



DISASSEMBLED KERNEL

kernel > ASM kernel.asm

```

kernel/kernel:     file format elf64-littleriscv


Disassembly of section .text:

0000000080000000 <_entry>:
    80000000: 00019117          auipc sp,0x19
    80000004: ec010113          addi  sp,sp,-320 # 80018ec0 <stack0>
    80000008: 6505                lui a0,0x1
    8000000a: f14025f3          csrr  a1,mhartid
    8000000e: 0585                addi  a1,a1,1
    80000010: 02b50533          mul a0,a0,a1
    80000014: 912a                add sp,sp,a0
    80000016: 507040ef          jal 80004d1c <start>

000000008000001a <spin>:
    8000001a: a001                j 8000001a <spin>

```

CALLING SYSTEM CALLS - XV6

- User code calls system call **“wrapper”** functions
- Args initially in the registers, a0 – a7 (RISC-V calling convention)

```c
int main() { exit(123); }
```

```
8:   07b00513            li      a0,123
c:   26e000ef            jal     27a <exit>
```

CALLING SYSTEM CALLS – XV6

- Kernel trap code saves these register values into the trapframe
- In trampoline.S

```
# save user a0 in sscratch so
# a0 can be used to get at TRAPFRAME.
csrw sscratch, a0
```

```
# save the user a0 in p->trapframe->a0
csrr t0, sscratch
sd t0, 112(a0)
```

- sscratch: supervisor privilege level CSR (Control and Status Register) – used to hold user register value during trap handling

CALLING SYSTEM CALLS – XV6

- All information in the trapframe
  => actual system call function called

```c
void
syscall(void)
{
  int num;
  struct proc *p = myproc();

  num = p->trapframe->a7;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    // Use num to lookup the system call function for num, call it,
    // and store its return value in p->trapframe->a0
    p->trapframe->a0 = syscalls[num]();
  } else {
    printf("%d %s: unknown sys call %d\n",
            p->pid, p->name, num);
    p->trapframe->a0 = -1;
  }
}
```

mapping to system call function pointer & calling it

CALLING SYSTEM CALLS

- All information in the trapframe
  => arguments are passed to the system call

```c
uint64
sys_sbrk(void)
{
  uint64 addr;
  int n;

  argint(0, &n);
  addr = myproc()->sz;
  if(growproc(n) < 0)
    return -1;
  return addr;
}
```

```c
// Fetch the nth 32-bit system call argument.
void
argint(int n, int *ip)
{
  *ip = argraw(n);
}
```

```c
static uint64
argraw(int n)
{
  struct proc *p = myproc();
  switch (n) {
  case 0:
    return p->trapframe->a0;
  case 1:
    return p->trapframe->a1;
  case 2:
    return p->trapframe->a2;
  case 3:
    return p->trapframe->a3;
  case 4:
    return p->trapframe->a4;
  case 5:
    return p->trapframe->a5;
  }
  panic("argraw");
  return -1;
}
```

USING GDB

Lab: System Call

FIRE UP GDB

```
jina@Jinas-MacBook-Pro xv6-labs % make qemu-gdb
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -gdwarf-2 -DSOL_UTI
ltin-strncmp -fno-builtin-strlen -fno-builtin-memset -fno-builtin-memmove -fno-builtin-mem
uiltin-malloc -fno-builtin-putc -fno-builtin-free -fno-builtin-memcpy -Wno-main -fno-built
e -no-pie  -c -o kernel/syscall.o kernel/syscall.c
riscv64-unknown-elf-ld -z max-page-size=4096 -T kernel/kernel.ld -o kernel/kernel kernel/e
l/swtch.o kernel/trampoline.o kernel/trap.o kernel/syscall.o kernel/sysproc.o kernel/bio.o
.o kernel/sysfile.o kernel/kernelvec.o kernel/plic.o kernel/virtio_disk.o kernel/start.o k
riscv64-unknown-elf-ld: warning: kernel/kernel has a LOAD segment with RWX permissions
riscv64-unknown-elf-objdump -S kernel/kernel > kernel/kernel.asm
riscv64-unknown-elf-objdump -t kernel/kernel | sed '1,/SYMBOL TAB
*** Now run 'gdb' in another window.
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kerne
0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0 -S -gd
```

```
jina@Jinas-MacBook-Pro xv6-labs % riscv64-elf-gdb
GNU gdb (GDB) 15.2
Copyright (C) 2024 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "--host=aarch64-apple-darwin24.1.0 --target=riscv64-elf".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word".
The target architecture is set to "riscv:rv64".
warning: No executable has been specified and target does not support
determining executable automatically.  Try using the "file" command.
0x0000000000001000 in ?? ()
(gdb)
```

GDB

```
    [SYS_sbrk]    sys_sbrk,
    [SYS_sleep]   sys_sleep,
    [SYS_uptime]  sys_uptime,
    [SYS_open]    sys_open,
    [SYS_write]   sys_write,
    [SYS_mknod]   sys_mknod,
    [SYS_unlink]  sys_unlink,
    [SYS_link]    sys_link,
    [SYS_mkdir]   sys_mkdir,
    [SYS_close]   sys_close,
    };

    void
    syscall(void)
    {
      int num;
      struct proc *p = myproc();

      num = p->trapframe->a7;
      if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
        // Use num to lookup the system call function for num, call it,
        // and store its return value in p->trapframe->a0
        p->trapframe->a0 = syscalls[num]();
      } else {
        printf("%d %s: unknown sys call %d\n",
                p->pid, p->name, num);
        p->trapframe->a0 = -1;
      }
    }
```

```
remote Thread 1.1 (src) In: syscall                                   L137  PC: 0x80001c80
(gdb) backtrace
#0  syscall () at kernel/syscall.c:133
#1  0x0000000080001a32 in usertrap () at kernel/trap.c:67
#2  0x0505050505050505 in ?? ()
(gdb) n
(gdb) n
(gdb) p /x *p
$1 = {lock = {locked = 0x0, name = 0x800071b8, cpu = 0x0}, state = 0x4, chan = 0x0, killed = 0x0, xstate = 0x0, pid = 0x1,
  parent = 0x0, kstack = 0x3fffffd000, sz = 0x1000, pagetable = 0x87f55000, trapframe = 0x87f56000, context = {
    ra = 0x800012b4, sp = 0x3fffffde90, s0 = 0x3fffffdec0, s1 = 0x80007d50, s2 = 0x80007920, s3 = 0x1, s4 = 0x3,
    s5 = 0x800189f0, s6 = 0x1, s7 = 0x4, s8 = 0xffffffffffffffff, s9 = 0x0, s10 = 0x0, s11 = 0x0}, ofile = {
    0x0 <repeats 16 times>}, cwd = 0x80015e60, name = {0x69, 0x6e, 0x69, 0x74, 0x63, 0x6f, 0x64, 0x65, 0x0, 0x0, 0x0, 0x0,
    0x0, 0x0, 0x0, 0x0}}
(gdb) p /x $sstatus
$2 = 0x200000022
(gdb)
```

KERNEL TRAP

```
void
syscall(void)
{
  int num;
  struct proc *p = myproc();

  // num = p->trapframe->a7;
  num = * (int *) 0;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
```

```
void
syscall(void)
{
    80001c6e: 1101                  addi   sp,sp,-32
    80001c70: ec06                  sd   ra,24(sp)
    80001c72: e822                  sd   s0,16(sp)
    80001c74: e426                  sd   s1,8(sp)
    80001c76: 1000                  addi   s0,sp,32
  int num;
  struct proc *p = myproc();
    80001c78: 8e4ff0ef             jal 80000d5c <myproc>
    80001c7c: 84aa                  mv   s1,a0

  // num = p->trapframe->a7;
  num = * (int *) 0;
    80001c7e: 00002683             lw   a3,0(zero) # 0 <_entry-0x80000000>
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
```

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
scause=0xd sepc=0x80001c7e stval=0x0
panic: kerneltrap
```

```
(gdb) c
Continuing.
[Switching to Thread 1.2]

Thread 2 hit Breakpoint 1, syscall () at kernel/syscall.c:138
(gdb) p p->name
$1 = "initcode\000\000\000\000\000\000\000"
(gdb)
```

Looking at the backtrace output, which function called syscall?

usertrap

What is the value of p->trapframe->a7 and what does that value represent?

7, the system call number of ‘exec’

What was the previous mode that the CPU was in?

1000000000000000000000000000100010 and index 8 is set to 0 (Previous mode: User)

Write down the assembly instruction the kernel is panicing at. Which register corresponds to the variable num?

lw a3,0(zero) # 0

Why does the kernel crash? Hint: look at figure 3-3 in the text; is address 0 mapped in the kernel address space? Is that confirmed by the value in scause above?

The address 0x0 is unused section(not mapped in the kernel address space.) - Page Fault

scause value can confirm it.

What is the name of the process that was running when the kernel paniced? What is its process id (pid)?

1, initcode

SYSTEM CALL TRACING

Lab: System Call

HOW SYSCALL WORKS

1 kernel/syscall.h

```diff
@@ -20,3 +20,4 @@
 #define SYS_link   19
 #define SYS_mkdir  20
 #define SYS_close  21
+#define SYS_trace  22
```

12 kernel/syscall.c

```diff
 extern uint64 sys_link(void);
 extern uint64 sys_mkdir(void);
 extern uint64 sys_close(void);
+extern uint64 sys_trace(void);

 // An array mapping syscall numbers from syscall.h
 // to the function that handles the system call.
@@ -126,6 +127,13 @@ static uint64 (*syscalls[])(void) = {
 [SYS_link]    sys_link,
 [SYS_mkdir]   sys_mkdir,
 [SYS_close]   sys_close,
+[SYS_trace]   sys_trace,
+};
+
+static char* syscall_list[] = {
+  "fork", "exit", "wait", "pipe", "read", "kill", "exec", "fstat", "chdir",
+  "dup", "getpid", "sbrk", "sleep", "uptime", "open", "write", "mknod", "unlink",
+  "link", "mkdir", "close", "trace"
 };

 void
 syscall(void)
 {
   int num;
   struct proc *p = myproc();

   num = p->trapframe->a7;
   if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
     // Use num to lookup the system call function for num, call it,
     // and store its return value in p->trapframe->a0
     p->trapframe->a0 = syscalls[num]();
+
+    if (p->trace_mask & 1 << num) {
+      printf("%d: syscall %s -> %lu\n", p->pid, syscall_list[num-1], p->trapframe->a0);
+    }
   } else {
     printf("%d %s: unknown sys call %d\n",
             p->pid, p->name, num);
     p->trapframe->a0 = -1;
   }
 }
```

ADD SYSCALL & FORK BEHAVIOR

12 kernel/sysproc.c

Expand all

```
uint64
sys_kill(void)
{
  int pid;

  argint(0, &pid);
  return kill(pid);
}

// return how many clock tick interrupts have occurred
// since start.
uint64
sys_uptime(void)
{
  uint xticks;

  acquire(&tickslock);
  xticks = ticks;
  release(&tickslock);
  return xticks;
}
+
+ uint64
+ sys_trace(void)
+ {
+   int n;
+
+   argint(0, &n);
+   if(n < 0)
+     return -1;
+   myproc()->trace_mask = n;
+   return 0;
+ }
```

1 kernel/proc.h

```
@@ -104,4 +104,5 @@ struct proc {
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
+ uint64 trace_mask;
};
```

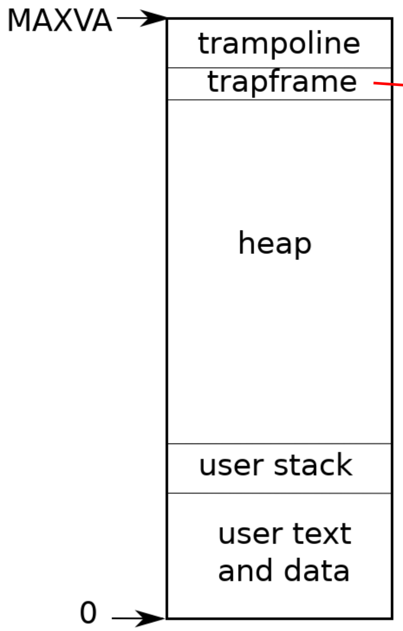
8 kernel/proc.c

Expand all

```
@@ -278,26 +278,27 @@
// Sets up child kernel stack to return as if from fork() system call.
int
fork(void)
{
  int i, pid;
  struct proc *np;
  struct proc *p = myproc();

  // Allocate process.
  if((np = allocproc()) == 0){
    return -1;
  }

  // Copy user memory from parent to child.
  if(uvmcopy(p->pagetable, np->pagetable, p->sz) < 0){
    freeproc(np);
    release(&np->lock);
    return -1;
  }
  np->sz = p->sz;

  // copy saved user registers.
  *(np->trapframe) = *(p->trapframe);
+ np->trace_mask = p->trace_mask;
```

ATTACK XV6

Lab: System Call

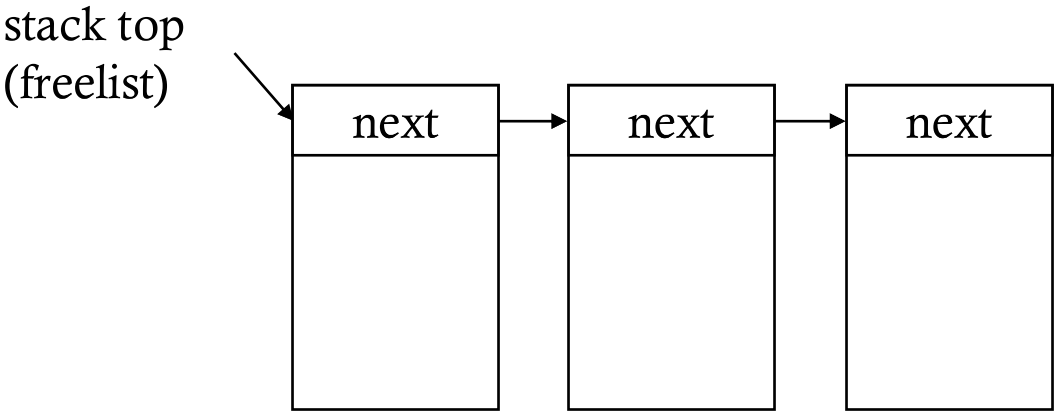


When transitioning into the kernel and back:
Save/restore the state of the user process

Figure 2.3: Layout of a process's virtual address space

xv6 physical memory management

- It manages free physical memory pages in simple stack, implemented using a linked list




```c
struct {
  struct spinlock lock;
  struct run *freelist;
} kmem;
```

```c
struct run {
  struct run *next;
};
```

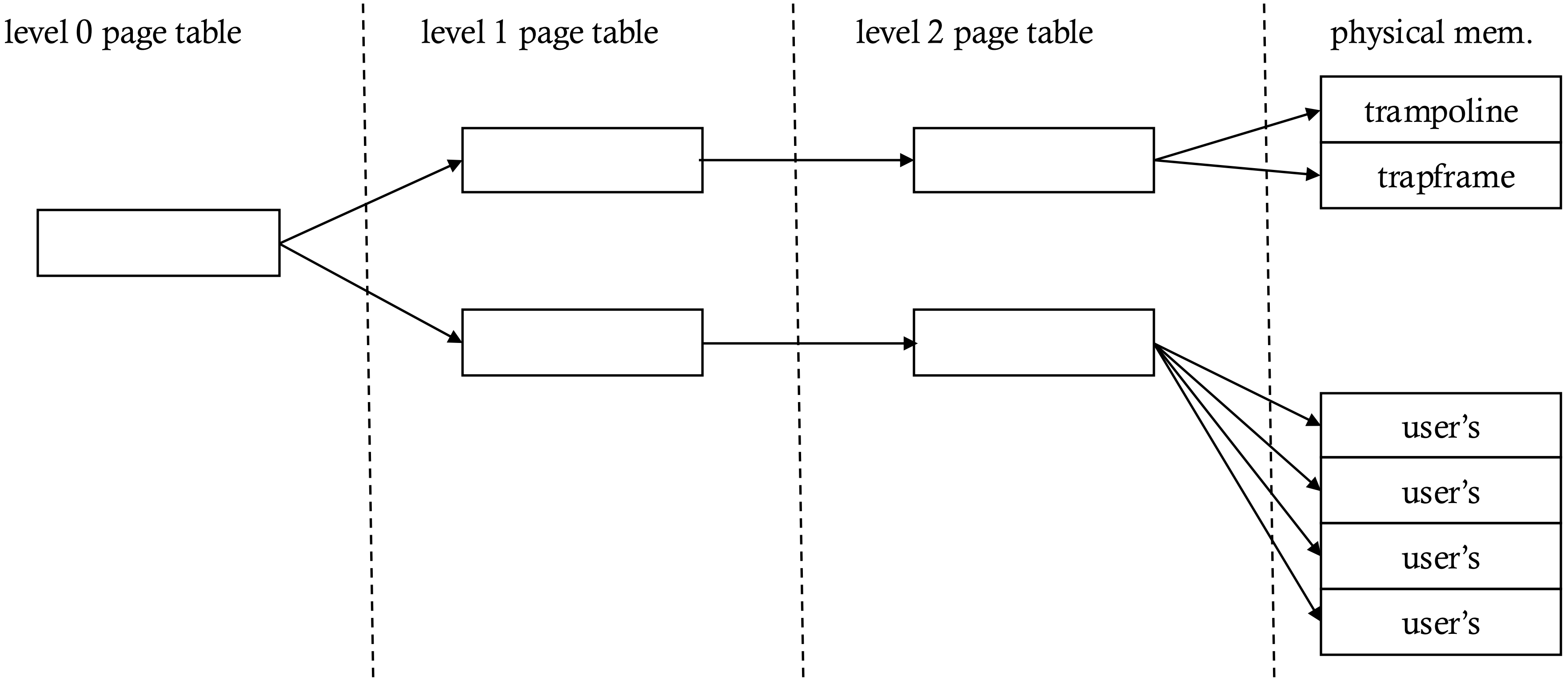
fork

- We want to figure out how many pages are allocated.
  - how? gdb breakpoint at…
    1. start of fork
    2. kalloc
    3. end of fork

=>

```
(gdb) info b
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x000000008000107c in fork at kernel/proc.c:281
        breakpoint already hit 14 times
2       breakpoint     keep n   0x00000000800008ea in uvmcopy at kernel/vm.c:346
        breakpoint already hit 3 times
3       breakpoint     keep y   0x000000008000116c in fork at kernel/proc.c:323
        breakpoint already hit 1 time
4       breakpoint     keep y   0x00000000800000f6 in kalloc at kernel/kalloc.c:78
        breakpoint already hit 10 times
```

Why 10 allocations during fork?



=> 10 pages allocated in total

for exec: read elf of secret.c

```
~/xv6-labs  syscall  readelf -l user/_secret                    ✔  33m 51s

Elf file type is EXEC (Executable file)
Entry point 0x5c
There are 4 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  RISCV_ATTRIBUT 0x00000000000073d8 0x0000000000000000 0x0000000000000000
                 0x0000000000000053 0x0000000000000000  R      0x1
  LOAD           0x0000000000001000 0x0000000000000000 0x0000000000000000
                 0x0000000000000cdc 0x0000000000000cdc  R E    0x1000
  LOAD           0x0000000000002000 0x0000000000001000 0x0000000000001000
                 0x0000000000000000 0x0000000000000020  RW     0x1000
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10
```

Similarly investigate exec

9 pages allocated, 9 freed

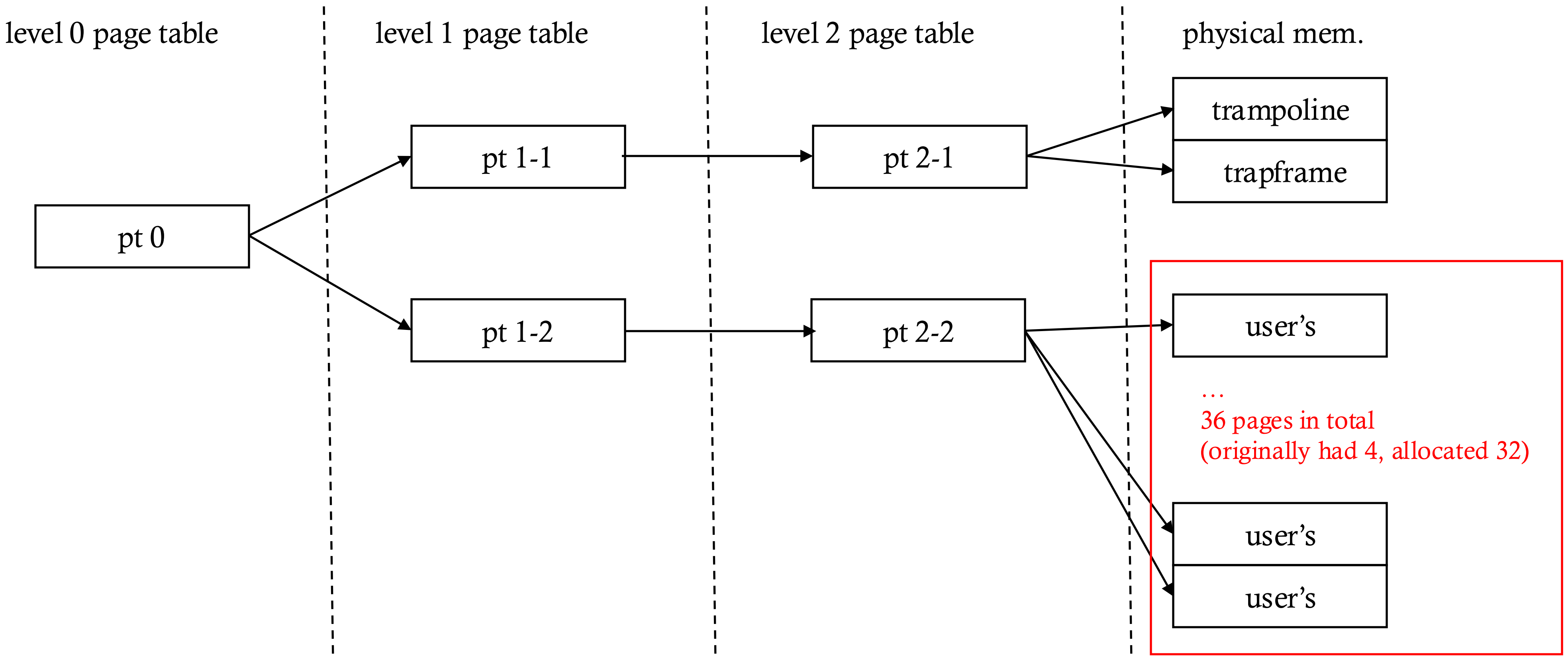
allocate: 5 page tables, 4 user memory pages
(2 for user memory, 1 for stack, 1 for stack guard)
free: 5 page tables of the original process, 4 user memory pages of the original process
(don’t need to free trapframe)

secret.c

- allocate 32 pages, but we do not need more page tables, since it is linear in the virtual memory space

```
int
main(int argc, char *argv[])
{
  if(argc != 2){
    printf("Usage: secret the-secret\n");
    exit(1);
  }
  char *end = sbrk(PGSIZE*32);
  end = end + 9 * PGSIZE;
  strcpy(end, "my very very very secret pw is:   ");
  strcpy(end+32, argv[1]);
  exit(0);
}
```

secret.c after allocating 32 pages



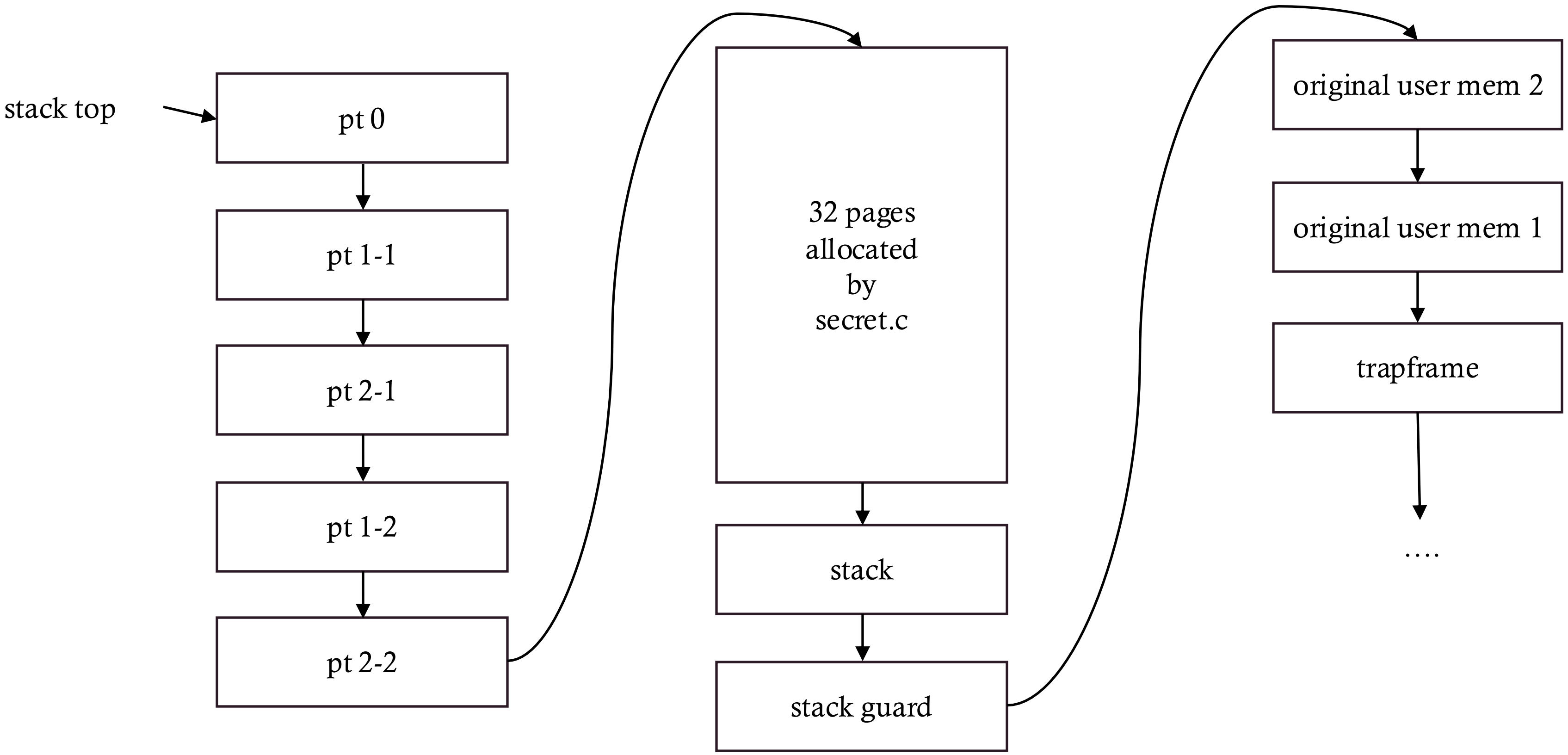
secret.c teardown by attacktest.c wait()

- dealloc everything of secret.c
  - trapframe
  - user memory
    - 1 stack
    - 1 stack guard
    - 34 user memory pages
  - 5 page tables

- => 42 pages in total freed

```c
// free a proc structure and the data hanging from it,
// including user pages.
// p->lock must be held.
static void
freeproc(struct proc *p)
{
  if(p->trapframe)
    kfree((void*)p->trapframe);
  p->trapframe = 0;
  if(p->pagetable)
    proc_freepagetable(p->pagetable, p->sz);
  p->pagetable = 0;
  p->sz = 0;
  p->pid = 0;
  p->parent = 0;
  p->name[0] = 0;
  p->chan = 0;
  p->killed = 0;
  p->xstate = 0;
  p->state = UNUSED;
}
```

freelist after secret.c exits



moving on to attack...

```
~/xv6-labs 〉 syscall   readelf -l user/_attack                         ✔ 〈 29m 13s

Elf file type is EXEC (Executable file)
Entry point 0x3a
There are 4 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  RISCV_ATTRIBUT 0x000000000000731b 0x0000000000000000 0x0000000000000000
                 0x0000000000000053 0x0000000000000000  R      0x1
  LOAD           0x0000000000001000 0x0000000000000000 0x0000000000000000
                 0x0000000000000c7c 0x0000000000000c7c  R E    0x1000
  LOAD           0x0000000000002000 0x0000000000001000 0x0000000000001000
                 0x0000000000000000 0x0000000000000020  RW     0x1000
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10
```

user memory size is same as secret.c => fork & exec would alloc / free same size of memory

How to access the secret in attack.c

- secret.c saved the secret at 10th page

```
char *end = sbrk(PGSIZE*32);
end = end + 9 * PGSIZE;
strcpy(end, "my very very very secret pw is:   ");
strcpy(end+32, argv[1]);
```

- which means, in the freelist, there are 27 pages (22 allocated by secret.c + 5 page tables) in front of the “secret page”
- 10 pages are popped from freelist when fork()ing attack.c
- 9 pages popped from freelist but then 9 pages pushed back (exec())
- access 17th

Answer

```c
int
main(int argc, char *argv[])
{
    char* end = sbrk(PGSIZE * 32);
    end = end + 16 * PGSIZE;
    char* secret = end + 32;
    printf("secret: %s\n", secret);
    write(2, secret, 8);
    exit(0);
}
```

Why it won’t work if the secret is stored without offset 32

- first 8B of each page used during memory managing
- i.e. overwritten with next pointer when maintaining the freelist

```
struct run {
  struct run *next;
};
```