xv6

xv6

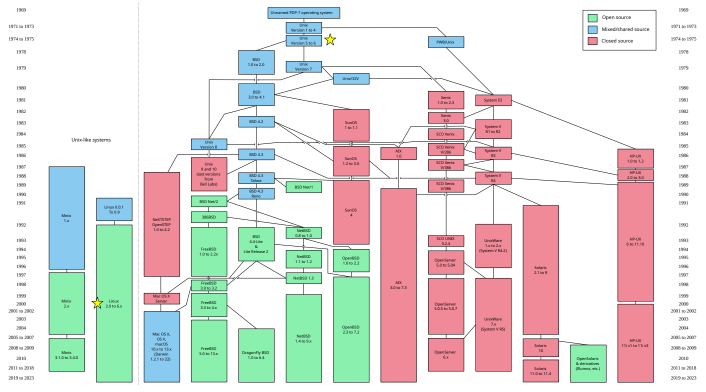
- xv6 is a modern re-implementation of Version 6 Unix (V6)
  - developed as a teaching OS
- early Unix versions (“Research Unix”)
  - V4 (1973) – first portable OS (written mostly in C)
  - V6 (1975) – first version of Unix with wide distribution outside Bell Labs
    - for DEC PDP-11
  - V7 (1979) – ancestor of modern Unix
  - Unix philosophy
    - "the power of a system comes more from the relationships among programs than from the programs themselves" (Kernighan, Pike)
    - key concepts
      - data stored in plain text
      - hierarchical file system
      - devices and inter-process communication treated as files
      - a large collection of software tools that can be connected via pipes rather than a single monolithic system

pdp11

introduced 1970
discontinued 1997

https://en.wikipedia.org/wiki/PDP-11

Unix History



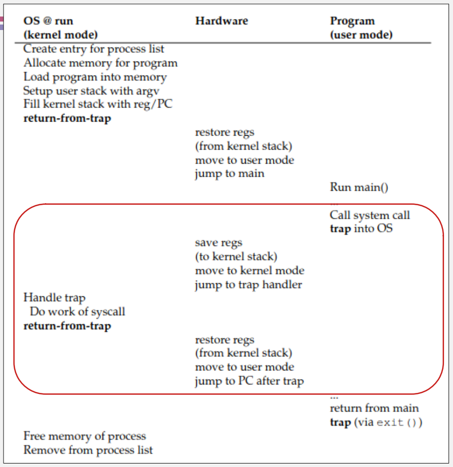
https://commons.wikimedia.org/wiki/File:Unix_history-simple.svg

xv6

- xv6 runs on x86 (32- and 64-bit) and RISC-V
  - x86 (CISC) (still) dominates desktop and laptop market
    - originally developed by Intel (now also AMD)
  - modern Mac laptops (2020-) use ARM (RISC)
    - more energy efficient, also more expensive
  - RISC-V is common in microcontrollers and embedded systems
- our environment
  - 32-bit x86 supported by QEMU
    - QEMU is an open-source system emulator providing a virtual view of an entire system (CPU, memory, devices)
    - allows OS development without needing the physical hardware, or rebooting when it doesn't work
  - QEMU is installed in the Linux VDI (non-GPU and GPU pools) – *not* Demarest 002

System Calls

- direct execution means user code runs directly on the CPU
  - no intervention from the OS
- system calls allow user code to do privileged things in a protected way
- OS elements and responsibilities
  - set up trap handler
  - set up system call handler
  - provide library routine callable from user programs
    - trap to OS
    - "do work of syscall"



OS @ run (kernel mode)	**Hardware**	**Program (user mode)**
Create entry for process list Allocate memory for program Load program into memory Setup user stack with argv Fill kernel stack with reg/PC **return-from-trap**		
	restore regs (from kernel stack) move to user mode jump to main	
		Run main() ... Call system call **trap** into OS
	save regs (to kernel stack) move to kernel mode jump to trap handler	
Handle trap Do work of syscall **return-from-trap**		
	restore regs (from kernel stack) move to user mode jump to PC after trap	
		... return from main **trap** (via `exit()`)
Free memory of process Remove from process list		

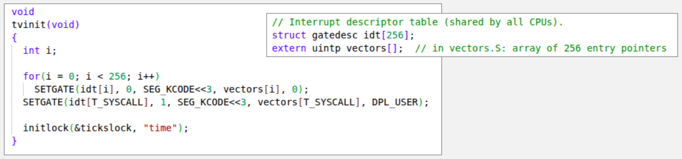
https://pages.cs.wisc.edu/~remzi/OSTEP/cpu-mechanisms.pdf

xv6 – Trap Handler

- on a trap, the hardware switches the privilege level of the CPU to kernel mode and transfers control to the low-level trap handler
- the *trap vectors* are set up on boot
  - defines where to find the low-level trap handler for each type of trap
    - e.g. system call, timer interrupt, …

xv6 – Trap Handler

- trap vector setup in xv6
  - `main` (in `kernel/main.c`) calls `tvinit()`
  - `tvinit` (in `kernel/trap.c`) sets up the *gate descriptors* in the interrupt descriptor table (IDT)
    - in-memory data structure with info about where the trap handler code is



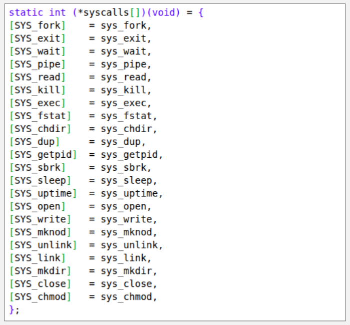
```
void
tvinit(void)
{
  int i;

  for(i = 0; i < 256; i++)
    SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
  SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);

  initlock(&tickslock, "time");
}
```

```
// Interrupt descriptor table (shared by all CPUs).
struct gatedesc idt[256];
extern uintp vectors[];  // in vectors.S: array of 256 entry pointers
```

  - `main` calls `mpmain()`
  - `mpmain` (in `kernel/trap.c`) sets up each processor
    - includes executing the assembly instruction to tell the hardware where to find the IDT in memory

xv6 – System Call Handler

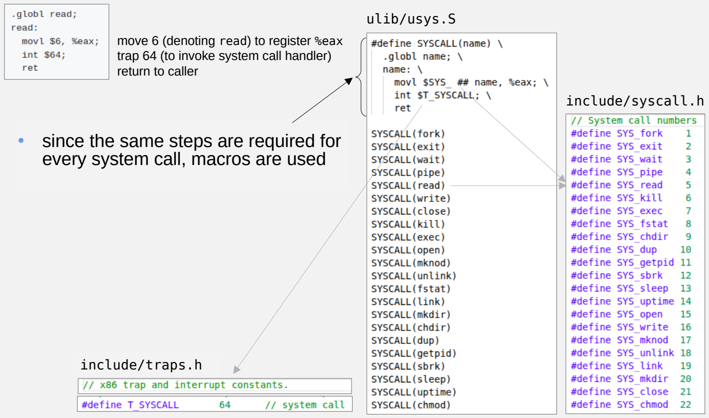
- set up table for system call handlers



```
static int (*syscalls[])(void) = {
[SYS_fork]    = sys_fork,
[SYS_exit]    = sys_exit,
[SYS_wait]    = sys_wait,
[SYS_pipe]    = sys_pipe,
[SYS_read]    = sys_read,
[SYS_kill]    = sys_kill,
[SYS_exec]    = sys_exec,
[SYS_fstat]   = sys_fstat,
[SYS_chdir]   = sys_chdir,
[SYS_dup]     = sys_dup,
[SYS_getpid]  = sys_getpid,
[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,
[SYS_chmod]   = sys_chmod,
};
```

`kernel/syscall.c`

xv6 – Syscall Library Routines



```
.globl read;
read:
  movl $6, %eax;
  int $64;
  ret
```

move 6 (denoting read) to register %eax
trap 64 (to invoke system call handler)
return to caller

- since the same steps are required for every system call, macros are used

ulib/usys.S

```
#define SYSCALL(name) \
  .globl name; \
  name: \
    movl $SYS_ ## name, %eax; \
    int $T_SYSCALL; \
    ret

SYSCALL(fork)
SYSCALL(exit)
SYSCALL(wait)
SYSCALL(pipe)
SYSCALL(read)
SYSCALL(write)
SYSCALL(close)
SYSCALL(kill)
SYSCALL(exec)
SYSCALL(open)
SYSCALL(mknod)
SYSCALL(unlink)
SYSCALL(fstat)
SYSCALL(link)
SYSCALL(mkdir)
SYSCALL(chdir)
SYSCALL(dup)
SYSCALL(getpid)
SYSCALL(sbrk)
SYSCALL(sleep)
SYSCALL(uptime)
SYSCALL(chmod)
```

include/syscall.h

```
// System call numbers
#define SYS_fork    1
#define SYS_exit    2
#define SYS_wait    3
#define SYS_pipe    4
#define SYS_read    5
#define SYS_kill    6
#define SYS_exec    7
#define SYS_fstat   8
#define SYS_chdir   9
#define SYS_dup    10
#define SYS_getpid 11
#define SYS_sbrk   12
#define SYS_sleep  13
#define SYS_uptime 14
#define SYS_open   15
#define SYS_write  16
#define SYS_mknod  17
#define SYS_unlink 18
#define SYS_link   19
#define SYS_mkdir  20
#define SYS_close  21
#define SYS_chmod  22
```

include/traps.h

```
// x86 trap and interrupt constants.

#define T_SYSCALL       64      // system call
```

xv6 – Do Work of Syscall

- system call handler
  - headers – include/user.h
  - system call bodies –
    - kernel/sysproc.c (process-related)
    - kernel/sysfile.c (file-related)

```
// system calls
int fork(void);
int exit(void) __attribute__((noreturn));
int wait(void);     /* POSIX incompatible */
int pipe(int*);
int write(int, const void*, int);
int read(int, void*, int);
int close(int);
int kill(int);      /* POSIX incompatible */
int exec(char*, char**);    /* POSIX incompatible */
int open(const char*, int);
int mknod(const char*, short, short);
int unlink(const char*);
int fstat(int fd, struct stat*);
int link(const char*, const char*);
int mkdir(const char*);     /* POSIX incompatible */
int chdir(const char*);
int dup(int);
int getpid(void);
char* sbrk(int);
int sleep(int);     /* POSIX incompatible */
int uptime(void);   /* POSIX incompatible */
int chmod(const char *, int);

// ulib.c
int stat(const char*, struct stat*);
char* strcpy(char*, const char*);
void *memmove(void*, const void*, int);
char* strchr(const char*, char c);
int strcmp(const char*, const char*);
void printf(int, char*, ...);
char* gets(char*, int max);
uint strlen(const char*);
void* memset(void*, int, uint);
void* malloc(uint);
void free(void*);
int atoi(const char*);
```

getpid – sysproc.c

```
int
sys_getpid(void)
{
  return proc->pid;
}
```

read – sysfile.c

```
int
sys_read(void)
{
  struct file *f;
  int n;
  char *p;

  if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n
    return -1;
  return fileread(f, p, n);
}
```

xv6 – Flow of Control

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

char buf[512];

void
wc(int fd, char *name)
{
  int i, n;
  int l, w, c, inword;

  l = w = c = 0;
  inword = 0;
  while((n = read(fd, buf, sizeof(buf))) > 0){
    for(i=0; i<n; i++){
      c++;
      if(buf[i] == '\n')
        l++;
      if(strchr(" \r\t\n\v", buf[i]))
        inword = 0;
      else if(!inword){
        w++;
        inword = 1;
      }
    }
  }
  if(n < 0){
    printf(1, "wc: read error\n");
    exit();
  }
  printf(1, "%d %d %d %s\n", l, w, c, name);
}
```

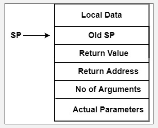
- user program calls a library routine

```
int
main(int argc, char *argv[])
{
  int fd, i;

  if(argc <= 1){
    wc(0, "");
    exit();
  }

  for(i = 1; i < argc; i++){
    if((fd = open(argv[i], 0)) < 0){
      printf(1, "wc: cannot open %s\n", argv[i]);
      exit();
    }
    wc(fd, argv[i]);
    close(fd);
  }
  exit();
}
```

– code goes into the user subdirectory
– add executable to UPROGS in Makefile

xv6 – Flow of Control

- user program calls a library routine
  - the handling of a subroutine call involves pushing an *activation record* with its arguments (and other info) onto the user stack



- jump to the subroutine
  - for a system call, this is the assembly code that traps into the OS

```
.globl read;
read:
  movl $6, %eax;
  int $64;
  ret
```

continued...

https://www.tutorialspoint.com/what-is-an-activation-record

xv6 – Flow of Control

```
.globl read;
read:
  movl $6, %eax;
  int $64;
  ret
```

For each system call library routine –

- set up arguments in registers
  - system call handler expects the arguments to be on the stack
    - already done by the call to the library routine – subroutine call in C puts the arguments on the stack
  - ☆ trap handler expects which system call is being invoked to be in register `%eax`
    - system call number is defined by the OS
- ☆ issue trap instruction
  - instruction and trap number are hardware-specific
    - for x86 – the trap instruction is `int`, system calls are trap number 64
- ☆ return
  - returning from subroutine call in C expects return value to be on the stack
    - already done by returning from the system call handler (a C subroutine)

xv6 – Flow of Control

- trap into OS
  - hardware saves registers
    - PC (IP) – program counter
    - eflags – current CPU status
    - SP – stack pointer
    - …
  - control is transferred to `vector64()`
    - low-level trap handler set up in `tvinit()`
    - pushes the trap number onto the stack (64 for system call)
    - jumps to handler for saving the rest of the context into the trap frame

`include/x86.h`

```
// Layout of the trap frame built on the stack by the
// hardware and by trapasm.S, and passed to trap().
struct trapframe {
  // registers as pushed by pusha
  uint edi;
  uint esi;
  uint ebp;
  uint oesp;      // useless & ignored
  uint ebx;
  uint edx;
  uint ecx;
  uint eax;

  // rest of trap frame
  ushort gs;
  ushort padding1;
  ushort fs;
  ushort padding2;
  ushort es;
  ushort padding3;
  ushort ds;
  ushort padding4;
  uint trapno;

  // below here defined by x86 hardware
  uint err;
  uint eip;
  ushort cs;
  ushort padding5;
  uint eflags;

  // below here only when crossing rings, such as from user to kernel
  uint esp;
  ushort ss;
  ushort padding6;
};
```

continued...

xv6 – Flow of Control

- trap into OS
  - …
  - `vector64`
    - pushes the trap number onto the stack
      - 64 for system call
    - jumps to handler for saving the rest of the context into the trap frame
  - `alltraps`
    - saves additional state into the trap frame on the stack
    - jumps to `trap()`

`kernel/vectors.S`

```
.globl vector64
vector64:
  pushl $0
  pushl $64
  jmp alltraps
```

```
  # vectors.S sends all traps here.
.globl alltraps
alltraps:
  # Build trap frame.
  pushl %ds
  pushl %es
  pushl %fs
  pushl %gs
  pushal

  # Set up data and per-cpu segments.
  movw $(SEG_KDATA<<3), %ax
  movw %ax, %ds
  movw %ax, %es
  movw $(SEG_KCPU<<3), %ax
  movw %ax, %fs
  movw %ax, %gs

  # Call trap(tf), where tf=%esp
  pushl %esp
  call trap
  addl $4, %esp
```

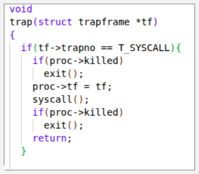
`kernel/trapasm.S`

continued...

xv6 – Flow of Control

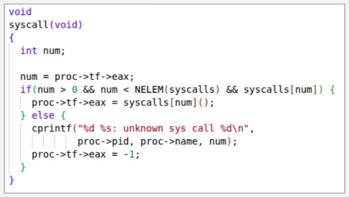
- trap into OS
  - …
  - `trap`
    - checks the trap number
    - if the process hasn’t been killed, calls the appropriate handler
    - if the process hasn’t been killed, returns
  - `syscall`
    - gets the system call number from `%eax`
    - invokes the handler for that system call
    - stores return code in `%eax`

kernel/trap.c



```
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(proc->killed)
      exit();
    proc->tf = tf;
    syscall();
    if(proc->killed)
      exit();
    return;
  }
```

kernel/syscall.c

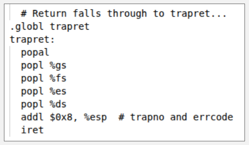


```
void
syscall(void)
{
  int num;

  num = proc->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    proc->tf->eax = syscalls[num]();
  } else {
    cprintf("%d %s: unknown sys call %d\n",
            proc->pid, proc->name, num);
    proc->tf->eax = -1;
  }
}
```

continued...

xv6 – Flow of Control

kernel/trapasm.S



```
  # Return falls through to trapret...
.globl trapret
trapret:
  popal
  popl %gs
  popl %fs
  popl %es
  popl %ds
  addl $0x8, %esp  # trapno and errcode
  iret
```

- return from trap
  - clean up stack
  - executes return from trap (`iret`) to reduce privilege level to user mode and jump to instruction following the trap
  - library routine returns

```
.globl read;
read:
  movl $6, %eax;
  int $64;
  ret
```