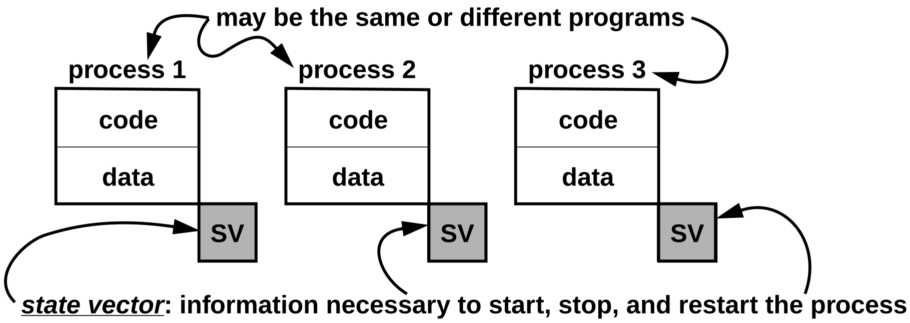
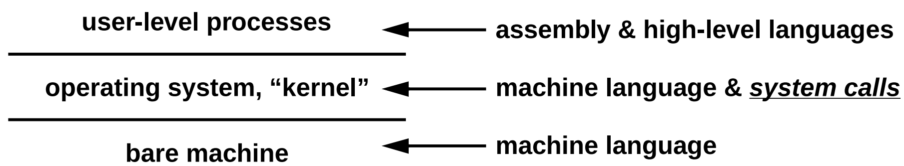
Operating Systems

- Operating systems manage ***processes*** and ***resources***
- Processes are executing ***instances*** of programs



- State vector: registers, program counter, memory mgmt registers, etc.



Privileged Instructions

- Machines have ***two*** kinds of instructions
  1. “normal” instructions, e.g., **add**, **sub**, etc.
  2. “privileged” instructions, e.g.,
     initiate I/O
     switch state vectors or ***contexts***
     load/save from protected memory
     etc.
- Operating systems ***hide*** privileged instructions and provide ***virtual instructions*** to access and manipulate ***virtual resources***, e.g., I/O to and from disc files
- Virtual instructions are ***system calls***
- Operating systems ***interpret*** virtual instructions

Processor Modes

- Machine level typically has 2 modes, e.g., “user” mode and “kernel” mode
- ***User mode***

  processor executes “normal” instructions in the user’s program

  upon encountering a “privileged” instruction, processor ***switches*** to kernel mode, and the operating system performs a service
- ***Kernel mode***

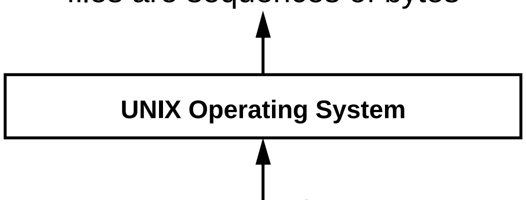
  processor executes both normal and privileged instructions
- User-to-kernel switch saves the information necessary to ***continue*** the execution of the user process
- Another view

  Operating system is a process that runs in kernel mode.

Virtual Resources

- OS provides a ***high-level*** representation of ***low-level*** resources
- For example, low-level disks are presented as file systems

simple to use file system
tree-structured, hierarchical
directories
named files
files are sequences of bytes



network
k disk drives
different capacities, speeds
access via cylinder, sector, track
fixed I/O format and transfer rules
authentication
error correction and handling

System Calls

- Virtual instructions are often presented as a set of ***system calls***
- Typical implementations (in order of prevalence)
  - single privileged instruction with parameters
  - interpret to other privileged instructions
  - jump to fixed locations
- Parameters are passed in a machine-dependent manner
  - in fixed registers
  - in fixed memory locations
  - in an argument block, with the block's address in a register
  - in-line with the system call
  - on the stack
  - combination of the above
- System calls return results in registers, memory, etc., and an error indication

System Calls, cont'd

- System call mechanism is tailored to the machine architecture
  - system calls on the SPARC use a ***trap*** instruction
  - **`ta 0`**
  - trap always; a trap value of 0 indicates a system call
  - parameters are in registers **`%g1`**, **`%o0`** — **`%o5`** and on the stack
- System call interface often designed to accommodate high-level languages
  - system calls are accessed by a library of procedures
  - e.g., on UNIX, system calls are packaged as a library of C functions
- Typical UNIX system call
  - **`nread = read(fd, buffer, n);`**
  - returns the number of bytes read from the file **`fd`**, or **`-1`** if an error occurs
  - what about EOF?

Implementing System Calls as Functions

- In the caller

```
mov fd,%o0
mov buffer,%o1
mov n,%o2
call _read; nop
mov %o0,nread
```

- Implementation of **read**

```
_read:
    set 3,%g1      /* 3 indicates READ system call */
    ta 0
    bcc L1; nop
    set _errno,%g1 /* sets errno to the error code */
    st %o0,[%g1]
    set -1,%o0     /* return -1 to indicate an error */
L1: retl; nop
```

operating system
sets the ***C*** bit if an error occurred
stores an error code in **%o0**; see **/usr/include/sys/errno.h**

note that **read** is a ***leaf*** function

- UNIX has ~150 system calls

see "**man 2 intro**" and **/usr/include/syscall.h**

Exceptions and Interrupts

- Operating systems also field ***exceptions*** and ***interrupts***
- ***Exceptions*** (a.k.a. traps): caused by execution of an instruction

  e.g., divide by 0, illegal address, memory protection violation, illegal opcode

- Exceptions are like ***implicit*** system calls

  operating systems can pass control to user processes to handle exceptions (e.g., "signals")

  operating systems have ways to process exceptions by defaults

  e.g., segmentation fault and core dump

- ***Interrupts***: caused by "external" activity unrelated to the user process

  e.g., I/O completion, clock tick, etc.

- Interrupts are like ***transparent*** system calls

  normally user processes cannot detect interrupts, nor need to deal with them

SPARC Traps

- A **`trap`** instruction

  enters kernel mode

  disables other traps

  decrements ***CWP***

  saves ***PC***, ***nPC*** in **`%r17`**, **`%r18`**

  sets ***PC*** to ***TBR***, ***nPC*** to ***TBR*** + 4

- Hardware trap codes

  1 reset

  2 access exception

  3 illegal instruction

  ...

- Software trap codes

  sets ***TBR*** to trap number + 128

- There are ***conditional traps*** just like conditional branches
- There are separate floating point traps

System Calls for Input/Output

- Associating/disassociating files with ***file descriptors***

```
int  open(char *filename, int flags, int mode)
int close(int fd)
```

- Reading/writing from file descriptors

```
int  read(int fd, char *buf, int nbytes)
int write(int fd, char *buf, int nbytes)
```

- Another version of **`cp`** ***source destination*** (see **`src/cp1.c`**)

```
#include <sys/file.h>
main(int argc, char *argv[]) {
    int count, src, dst;
    char buf[4096];
    if (argc != 3)
        error("usage: %s source destination\n", argv[0]);
    if ((src = open(argv[1], O_RDONLY, 0)) < 0)
        error("%s: can’t read ‘%s’\n", argv[0], argv[1]);
    if ((dst = open(argv[2], O_WRONLY|O_CREAT, 0666)) < 0)
        error("%s: can’t write ‘%s’\n", argv[0], argv[2]);
    while ((count = read(src, buf, sizeof buf)) > 0)
        write(dst, buf, count);
    return EXIT_SUCCESS;
}
```

Write with Confidence

- Most programs don’t check for ***write errors*** or writes that are ***too large***

```
int ironclad_write(int fd, char *buf, int nbytes) {
    char *p, *q;
    int n;

    p = buf;
    q = buf + nbytes;
    while (p < q)
        if ((n = write(fd, p, q - p)) > 0)
            p += n;
        else
            perror("iconclad_write:");
    return nbytes;
}
```

- **perror** issues a diagnostic for the error code in **errno**

```
iconclad_write: file system full
```


Buffered I/O

- Single-character I/O is usually too slow

```
int getchar(void) {
    char c;

    if (read(0, &c, 1) == 1)
        return c;
    return EOF;
}
```

- Solution: read chunks of input into a buffer, dole out chars one at a time

```
int getchar(void) {
    static char buf[1024];
    static char *p;
    static int n = 0;

    if (n--)
        return *p++;
    if ((n = read(0, p = buf, sizeof buf)) > 0)
        return getchar();
    n = 0;
    return EOF;
}
```

- Where’s the bug?

Implementing the Standard I/O Library

- Single-character I/O functions are usually implemented as macros

```
#define getc(p) (--(p)->_cnt >= 0 ? \
        (int)(*(unsigned char *)(p)->_ptr++) : \
        _filbuf(p))

#define getchar() (getc(stdin))
```

- A **`FILE`** holds per-file buffer information

```
typedef struct _iobuf {
    int _cnt;      /* number of characters/slots left in the
buffer */
    char *_ptr;    /* pointer to the next character in the
buffer */
    char *_base;   /* the beginning of the buffer */
    int _bufsiz;   /* size of the buffer */
    short _flag;   /* open mode flags, etc. */
    char _file;    /* associated file descriptor */
} FILE;

extern FILE *stdin, *stdout, *stderr;
```

- See **`/usr/princeton/include/ansi/stdio.h`**

Buffered Writes

- Single-character writes are usually implemented by macros

```
#define putc(c,p) (--(p)->_cnt >= 0 ? \
        (p)->_ptr++ = (c) : \
        _flsbuf((c), (p)))

#define putchar(c) (putc((c),stdout))
```

- Buffering can interfere with interactive streams

```
for (p = "Enter your name:\n"; *p; p++) putchar(*p);
for (p = buf; ; p++)
    if ((*p = getchar()) == '\n')
        break;
for (p = "Enter your age:\n"; *p; p++) putchar(*p);
for (p = buf; ; p++)
    if ((*p = getchar()) == '\n')
        break;
```

bug: program waits for input ***before*** prompt appears

Buffered Writes, cont'd

- Output stream must be flushed before reading the input

```
void fflush(FILE *stream)
for (p = "Enter your name:\n"; *p; p++) putchar(*p);
fflush(stdout);
for (p = buf; ; p++)
    if ((*p = getchar()) == '\n')
        break;
for (p = "Enter your age:\n"; *p; p++) putchar(*p);
fflush(stdout);
for (p = buf; ; p++)
    if ((*p = getchar()) == '\n')
        break;
```

- Standard I/O supports ***line-buffered*** files

```
#define putc(x, p) (--(p)->_cnt >= 0 ?\
    (int)(*(unsigned char *)(p)->_ptr++ = (x)) : \
    (((p)->_flag&_IOLBF) && -(p)->_cnt < (p)->_bufsiz ? \
        ((*(p)->_ptr = (x)) != '\n' ? \
              (int)(*(unsigned char *)(p)->_ptr++)   : \
            _flsbuf(*(unsigned char *)(p)->_ptr, p)) : \
        _flsbuf((unsigned char)(x), p)))
```

- Why is line buffering necessary?

```
f = fopen("/dev/tty", "w")
```