Operating Systems

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

ssor Modes

odes, e.g., "user" mode and "kernel"

tions in the user's program

truction, processor ***switches*** to kernel mode, and
rvice

d privileged instructions

e information necessary to ***continue*** the

runs in kernel mode.

Copyright ©1995 D. Hanson, K. Li & J.P. Singh

state vector: information necessary to start, stop, and restart the process

- registers, program counter, stack ptr, memory mgmt registers, etc.
- *user level processes* execute machine language & *system calls*
- *operating system, "kernel"* machine language & assembly & high-level languages
- *base machine* machine language

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

Copyright ©1995 D. Hanson, K. Li & J.P. Singh

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

UNIX Operating System

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

Copyright ©1995 D. Hanson, K. Li & J.P. Singh

stem Calls

resented as a set of ***system calls***

der of prevalence)

ameters

ons

achine-dependent manner

's address in a register

egisters, memory, etc., and an error

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?

stem Calls as Functions

```
dicates READ system call */

 errno to the error code */

rn -1 to indicate an error */
```

ed
ee **/usr/include/sys/errno.h**

clude/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 default
  - 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

ARC Traps

ke conditional branches

nt 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;
}
```


th Confidence

write errors or writes that are ***too large***

```
 char *buf, int nbytes) {
```

```
p, q - p)) > 0)
```

```
d_write:");
```

the error code in `errno`

n 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?

e Standard I/O Library

are usually implemented as macros

```
nt >= 0 ? \
ar *)(p)->_ptr++) : \


din))
```

rmation

```
er of characters/slots left in the

ter to the next character in the

eginning of the buffer */
of the buffer */
mode flags, etc. */
ciated file descriptor */


t, *stderr;
```

nsi/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

Copyright ©1995 D. Hanson, K. Li & J.P. Singh

l Writes, cont'd

l before reading the input

```
\n"; *p; p++) putchar(*p);


= '\n')

n"; *p; p++) putchar(*p);


= '\n')
```

ered files

```
>_cnt >= 0 ?\
r)(p)->_ptr++ = (x)) : \
& -(p)->_cnt < (p)->_bufsiz ? \
 != '\n' ? \
ned char *)(p)->_ptr++)    : \
ned char *)(p)->_ptr, p)) : \
har)(x), p)))
```

y?