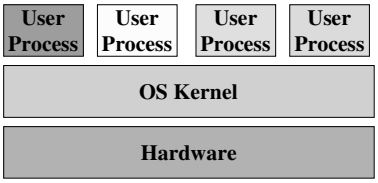
System Calls

CS 217

Operating Systems

- The OS virtualizes the system's resources
  - multiplexes shared physical resources among users
  - turns physical resources (hardware) into logical resources



Processes

- A process…
  - runs an instance of an application program
  - runs on behalf of some user (perhaps root)
  - is an abstraction provided by the kernel
  - is a virtualization of the computer
  - accesses physical resources indirectly through the kernel
  - includes state that is maintained by the kernel

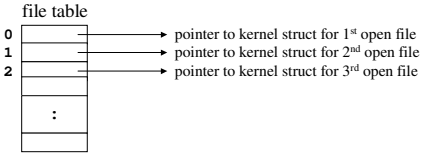
Process State

- CPU registers

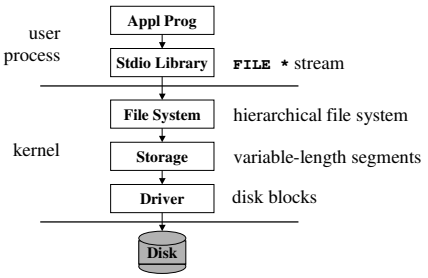
  `PC, PSR, %r0..%r31, %f0..%f31,…`

  saved/restored whenever the process stops/starts

- Kernel data structures

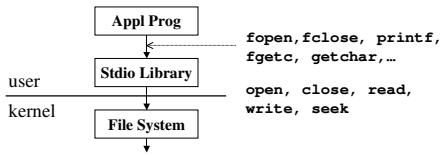


Layers of Abstraction



System Calls

- Method by which user processes invoke kernel services: "protected" procedure call



- Unix has ~150 system calls; see

  `man 2 intro`

  `/usr/include/syscall.h`

System Calls (cont)

- Processor modes
  - user mode: can execute normal instructions and access only user memory
  - supervisor mode: can also execute privileged instructions and access all of memory (e.g., devices)
  - when user process executes a privileged instruction, the processor switches to supervisor mode and jumps to a pre-defined address
- Trap instructions
  - trap instructions (e.g., **`ta`**) are a common example of a privileged instruction
  - system calls are often implemented using traps

System Calls (cont)

- Parameters passed…
  - in fixed registers
  - in fixed memory locations
  - in an argument block, w/ block's address in a register
  - on the stack
- Mechanism is highly machine-dependent; e.g.,
  - **`ta 0`**
  - with parameters in **`%g1`** (function), **`%o0..%o5`**, and on the stack

Read System Call

- Read call
  - **`nread = read(fd, buffer, n);`**
  - returns number of bytes read, or –1 if there's an error
- In the caller

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

Read System Call (cont)

- User-side implementation (**libc**)

```
_read: set 3,%g1
       ta 0
       bcc L1; nop
       set _errno,%g1
       st %o0,[%g1]
       set -1,%o0
  L1: retl; nop
```

- Kernel-side implementation
  - sets the C bit if an error occurred
  - stores an error code in **%o0**
    (see **/usr/include/sys/errno.h**)

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
- TBR: Trap Base Register
  - memory address of 1st instruction of trap handler
  - allows for only 4 instructions for each trap type (tt)

Sparc Traps (cont)

- Trap types (tt)

```
0x00   reset
0x01   instruction_access_exception
...
0x05   window_overflow
0x06   window_underflow
...
0x11   interrupt_level_1
0x12   interrupt_level_2
...
0x2a   divide_by_zero
...
0x80..0xff   trap_instruction
```

Sparc Traps (cont)

- Traps 0 through 127 are hardware traps
  - exceptions (e.g., divide by zero, illegal instruction)
  - interrupts (e.g., from external devices)
- Traps 128 through 255 are software traps
  - tt is set to 128 + argument to trap instruction

Write Safely

```
int safe_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("safe_write:");
   return nbytes;
}
```

perror issues a diagnositic for the code in **errno**

```
safe_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;
}
```

Buffered I/O (cont)

- Solution: read a chunk and dole out as needed

```
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;
}
```


Standard I/O Library

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

typedef struct _iobuf {
   int _cnt;    /* num chars left in buffer */
   char *_ptr;  /* ptr to next char in buffer */
   char *_base; /* beginning of buffer */
   int _bufsize;/* size of buffer */
   short _flag; /* open mode flags, etc. */
   char _file;  /* associated file descriptor */
} FILE;
extern FILE *stdin, *stdout, *stderr;
```


Buffered Writes

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

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;
```

Buffered Writes (cont)

- Flush output stream before reading 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;
```


Line-Buffered Files

```
#define putc(x, p) (--(p)->_cnt >= 0 ? \
 (int)(*(unsigned char *)(p)->_ptr++ = (x) : \
 (((p)->_flag&_IOLBF) && -(p)->_cnt < (p)->_bufsize ? \
    ((*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")
```