What is Mode Switch? When does it occur?

Program execution switches between different CPU modes

- **User -> Kernel**: system calls, interrupts, exceptions
- **Kernel -> User**: finish handling of sys calls/interrupts/exceptions
Previous class…

Difference between Interrupts, Exceptions, and Signals. Give examples

**Interrupts**: due to hardware; e.g., timer, I/O devices

**Exceptions**: due to software; e.g., breakpoints inserted by debugger, divide-by-zero, null pointer exceptions

**Signals**: due to exception handling or signaling API calls; e.g., SIGSEGV (due to illegal memory access)
# Interrupts, Exceptions and Signals

<table>
<thead>
<tr>
<th>Type</th>
<th>Triggered by</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exceptions (or, s/w interrupts)</td>
<td>Instruction execution</td>
<td>breakpoint; page fault; divide-by-zero; system calls</td>
</tr>
<tr>
<td>Signals</td>
<td>kill(); sent by exception handler; process exit</td>
<td>SIGTRAP; SIGSEGV; SIGFPE</td>
</tr>
<tr>
<td>Interrupts (or, h/w interrupts)</td>
<td>interval timer and I/O</td>
<td>timer; input ready; output finished</td>
</tr>
<tr>
<td>Language exceptions (as in C++ and Java)</td>
<td>throw</td>
<td>throw std::invalid_argument(“”);</td>
</tr>
</tbody>
</table>
How does a computer respond to a keystroke?

**Hardware part:** an interrupt is generated by the keyboard, and CPU automatically sets the program counter to the corresponding interrupt handler (recall IDT[n])

**Software (Kernel):** the interrupt handler executes the device driver for the keyboard, which reads the character from the keyboard buffer to the kernel-space buffer

**Software (User and Kernel):** the process that waits for the input is waken up (blocked -> ready), and the kernel copies the character from the kernel-space buffer to the user-space buffer
How does the system respond to divide-by zero (e.g., a/0) in user code?

1. When a divide-by-zero operation is performed, an exception (also called s/w interrupt or fault) is triggered
2. To handle the exception, execution switches to the kernel mode
3. The kernel handles the exception by generating a SIGFPE signal for the faulty process
4. After exception handling, execution switches back to the user mode
5. The signal handler is invoked (the default handler is implemented and registered by libc; it terminates the process)
Implement a simplest Shell using fork(), exec(), wait()

**fork():** create a child process.
**exec():** replace the execution of the current program with the designated one
**wait():** suspend the calling process until any of its child process exits
What is a Zombie Process? Why is it harmful? Why is it necessary?

When a process exits, it becomes a zombie unless it is waited for by its parent or its parent has explicitly expressed no interest in the exit state.

A zombie process occupies precious kernel resources, such as PCB.

The PCB stores the exit value of the zombie process.
Outline

• Pipe
• FIFO
• Message Queue
• Unix Domain Socket
• Shared Memory
**IPC - Pipe**

- **Pipe** is an IPC technique for passing information from one process to another.
- **Byte stream; no notion of “message boundaries”**
- **Example:**
  - Search lines that contain some word: `cat * | grep “os”`
  - # of files in a directory: `ls | wc -l`
Understand a pipe as a circular ring buffer

Conceptual view of a circular queue

- Not yet processed data in FIFO order
- Read position
- Write position

Capacity: 64kB
pipe() - details

- int pipe(int *fd[2])
  - creates a new pipe and returns two file descriptors
    - fd[0]: refers to the read end
    - fd[1]: refers to the write end
pipe(), fork(), and close() are good friends

- **pipe()**: create a pipe and return fd[0] and fd[1]
- **fork()**: child inherits fd[0] and fd[1]
- **close()**: close unused file descriptors
How do pipes work under the hood?

• There is an in-memory file system, pipefs, in the kernel memory address space.
• Pipefs is mounted at system initialization.
• A pipe corresponds to a file created in pipefs.
• Pipe() creates a pipe and returns two file descriptors:
  – One for the read end, opening using O_RDONLY, and
  – One for the write end, opened using O_WRONLY.
More details about pipe()

- If read() from a write-closed pipe: end-of-file
- If write() to a read-closed pipe: SIGPIPE -> write failure
- By default, blocking I/O
  - If a process attempts to read from an empty pipe, then read() will block until data is available
  - If a process attempts to write buf to a full pipe, then write() blocks until min(buf, PIPE_BUF = 4KB) is available
    - fcntl() can change it to non-blocking behavior
- If write() too large a message: the message will be split into pieces with each = PIPE_BUF (4KB in Linux) by the system
Example: passing an integer between two processes

Child:
int n = something();
write(fd[1], &n, sizeof(n));

Parent:
int n;
read(fd[0], &n, sizeof(n));

Q1: How does a writer indicate the end of data sending?
A: Either send the size to the receiver in the beginning, so the receiver know how many to expect; or send an indicator char (e.g., EOF=-1) in the end

Q2: Can a writer close the pipe immediately after sending data?
A: No. read() will return 0 immediately without receiving the data in the buffer

Q3: Then how can the writer know when to close the pipe?
A: Semaphore (to be covered soon)
Explain what happens under the hood when you run the command “ls –ls | sort”

1. Process creation: fork and exec
2. Pipe creation
3. Redirection: set ls’s STDOUT to pipe’s write end, and set sort’s STDIN to pipe’s read end
4. Scheduling:
   • ls: write->block->write->…;
   • sort: read->block->read->....
Example: `ls | sort`

```c
FILE *pipe_fp, *infile;
char readbuf[80];

/* Open up input file */
infile = fopen(fileName, "rt");
/* Create one way pipe line with call to popen() */
   It does pipe creation, fork, and exec
*/
pipe_fp = popen(childProgramName, "w");
while(true) {
    fgets(readbuf, 80, infile);
    if(feof(infile)) break;
    fputs(readbuf, pipe_fp);
}

fclose(infile);
pclose(pipe_fp); // close the pipe and wait()
```

parent

STDIN_FILENO = 0

child

```
read() a closed pipe will return error!
```
IPC – FIFO (also called Named Pipe)

- Unlike a pipe, a FIFO is not limited to be used between the parent and child processes
Example of FIFO

```c
#define FIFO_FILE "MYFIFO"

FILE *fp;
char readbuf[80];

/* Create the FIFO if it does not exist */
mkfifo(myfifo, 0666);

fp = fopen(FIFO_FILE, "r");
fgets(readbuf, 80, fp);
printf("Received string: %s\n", readbuf);
fclose(fp); // fclose() will not delete fifo
```

```c
#define FIFO_FILE "MYFIFO"

fp = fopen(FIFO_FILE, "w")
fputs("Hello!", fp);
fclose(fp);
return(0);
```
Comparison

• Unnamed pipes
  – Purely in-memory
  – The pipe is deleted after all file descriptors that refer to it are closed
  – Have to be used between parent and child
  – Typically, 1 writer and 1 reader

• FIFOs
  – A file is created in your hard disk, but the I/O is through the kernel memory
  – Not deleted even you close the file descriptor (you need to explicitly remove it like removing a file)
  – Not limited to parent-child processes
  – There can be multiple writers and one reader
IPC – Message queues

Conceptually, a queue in kernel

<table>
<thead>
<tr>
<th>POSIX APIs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>qid = mq_open(QUEUE_NAME, O_CREAT</td>
<td>O_RDONLY, 0644, &amp;attr);</td>
</tr>
<tr>
<td>qid = mq_open(QUEUE_NAME, O_WRONLY);</td>
<td>Open a queue</td>
</tr>
<tr>
<td>mq_send(mq, buffer, MAX_SIZE, 0)</td>
<td>Send a message; can specify the priority of the message</td>
</tr>
<tr>
<td>mq_receive(mq, buffer, MAX_SIZE, NULL)</td>
<td>Receive a message; can specify the priority also</td>
</tr>
</tbody>
</table>
Unix Domain Socket

• The power of Socket = Pipe + Message Queue
  – Support both Byte Stream and Datagrams (packets)
• Plus, it is bi-directional
  – Both ends of a socket pair can send and receive
• Recommended way for data passing!
Unix Domain Socket

• Unlike a network socket, a Unix domain socket is known by the path name
  – E.g., /usr/home/unique_socket_name
• Server: bind(unique path) -> listen() -> accept()
• Clint: connect(unique path)
• Example:
IPC – Shared Memory

- A region of physical memory that is mapped to two (or more) processes, such that when a process updates the share memory the other process can see the modification immediately, and vice versa
Shared Memory

POSIX Shared Memory
Shared Memory

• The previous three mechanisms (Pipes, FIFOs, and Message Queues) all require mode switch when sending/receiving information (why? recall system calls), while shared memory does NOT
How to create a Shared Memory?

- `shmget`: System V API
- `shm_open`: POSIX API
- `mmap` + `fork` -> the easiest way to create a shared memory between processes
  - `mmap(MAP_ANONYMOUS | MAP_SHARED)`
# Compare different IPCs

<table>
<thead>
<tr>
<th>IPC method</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pipes</td>
<td>Can only be used among parent and child</td>
</tr>
<tr>
<td>FIFO (named pipes)</td>
<td>Pipe is named using a string, so doesn’t have the limitation above</td>
</tr>
<tr>
<td>Message Queues</td>
<td>Supports message boundary and message types / priorities</td>
</tr>
<tr>
<td>Unix-domain Socket</td>
<td>Bi-directional; support both stream and packets; can pass file descriptors</td>
</tr>
<tr>
<td>Shared Memory</td>
<td>Data passing doesn’t go through kernel, so it is usually the most efficient one</td>
</tr>
</tbody>
</table>
Interesting Readings

• System V IPC vs POSIX IPC
  – https://stackoverflow.com/q/4582968/577165
  – I personally prefer POSIX
• Linux IPC
• Microsoft IPC
• A Summary list of IPC:
  – https://en.wikipedia.org/wiki/Inter-process_communication
• Unix Domain Socket vs. Pipe