Syscalls: Adv. I/O 5: Signal-Driven I/O

1. Overview
2. Nonblocking I/O
3. Multiplexed I/O
4. Additional Uses for I/O Muliplexing
5. **Signal-Driven I/O**
   - **signal-driven I/O**
   - **edge-triggered notification**
   - **signal-driven I/O vs. multiplexed I/O**
6. Memory-Mapped I/O
7. Zero-Copy I/O
8. Scatter-Gather I/O
9. Asynchronous I/O (AIO)

Signal-Driven I/O

Signal-driven I/O refers to a process configuring things so that the kernel sends it a **signal** when I/O is possible on a file descriptor.

Signal-driven I/O is enabled by doing the following steps:

1. Use `sigaction()` to register a **signal handler** for the signal that will be delivered (by default this is `SIGIO`)
2. Set the PID of the process that is to receive I/O signals (known as the FD "owner"):
   ```
   fcntl(fd, F_SETOWN, pid)
   ```
3. Enable *signal-driven I/O* for the FD and set the FD into **nonblocking mode**:
   ```
   fcntl(fd, F_SETFL, fcntl(fd, F_GETFL) | O_ASYNC | O_NONBLOCK)
   ```

Signal-Driven I/O (contd.)

Once the above steps are complete, the calling process can continue with other tasks.

When I/O is possible, the kernel will generate a signal for the process, leading to the signal handler being run.

(Remember that *signals* are effectively *software interrupts*, and are inherently **asynchronous**.)

The signal handler must ensure that the appropriate I/O gets carried out.

Edge-Triggered Notification

Signal-driven I/O provides **edge-triggered notification**.

This means that a signal is sent *only once*, when the FD changes state from I/O not possible to I/O possible.

Edge-triggered notification requires that the process perform *as much I/O as can be done at that time* whenever the handler is invoked.

This is why the FD was put into *nonblocking mode*.

Doing as much I/O as possible means looping, making I/O syscalls, until a call fails with the error `EAGAIN` (or `EWOULDBLOCK`).

(This is similar to what must be done when using `epoll()` in edge-triggered mode.)

Signal-Driven vs. Multiplexed I/O

Why might one be interested in *signal-driven I/O* when we have *multiplexed I/O?*

For **multiplexed I/O** (`poll()`, etc.) to provide **event-driven** behavior, there must be a *dedicated thread of execution* looping and checking for ready FDs.

Signal-driven I/O does not require a dedicated thread of execution monitoring for ready FDs.

Instead, the running thread will be *interrupted by a signal* whenever I/O is possible.