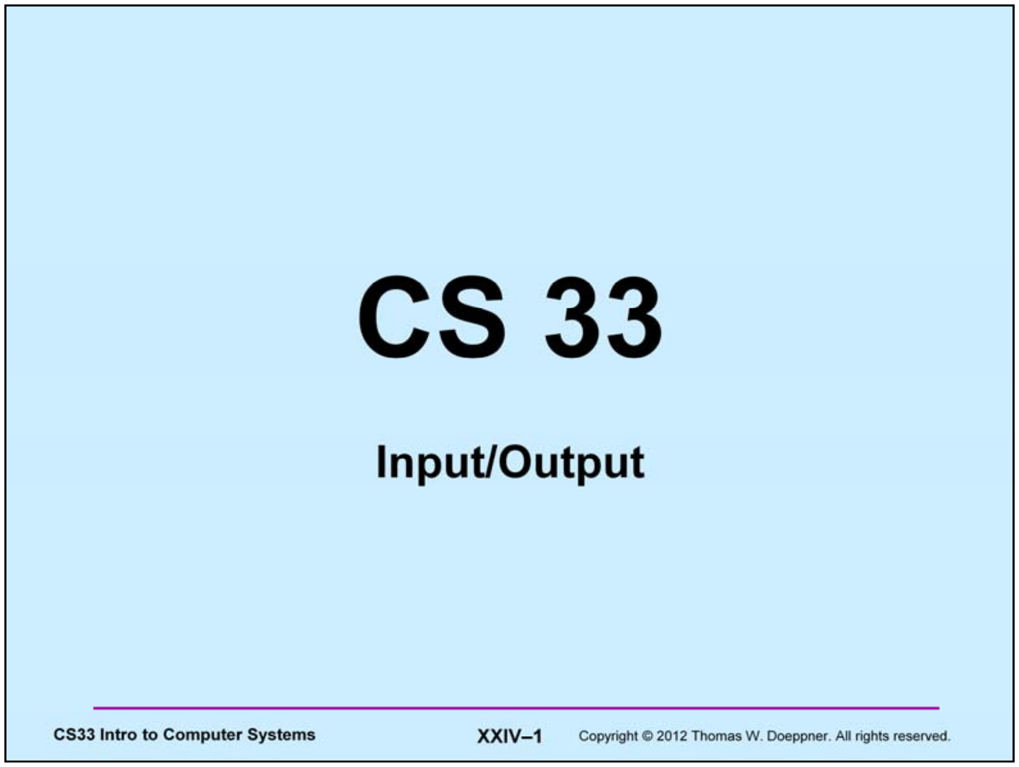
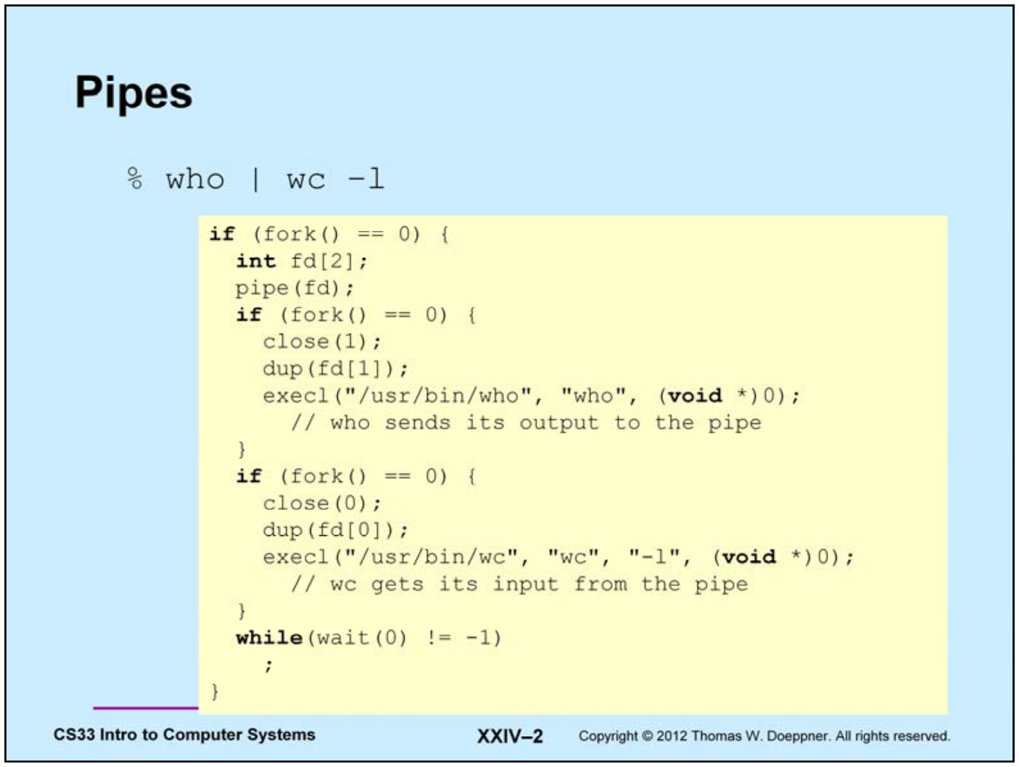


CS 33

Input/Output

CS33 Intro to Computer Systems XXIV–1

The source code used in this lecture is on the course web page.

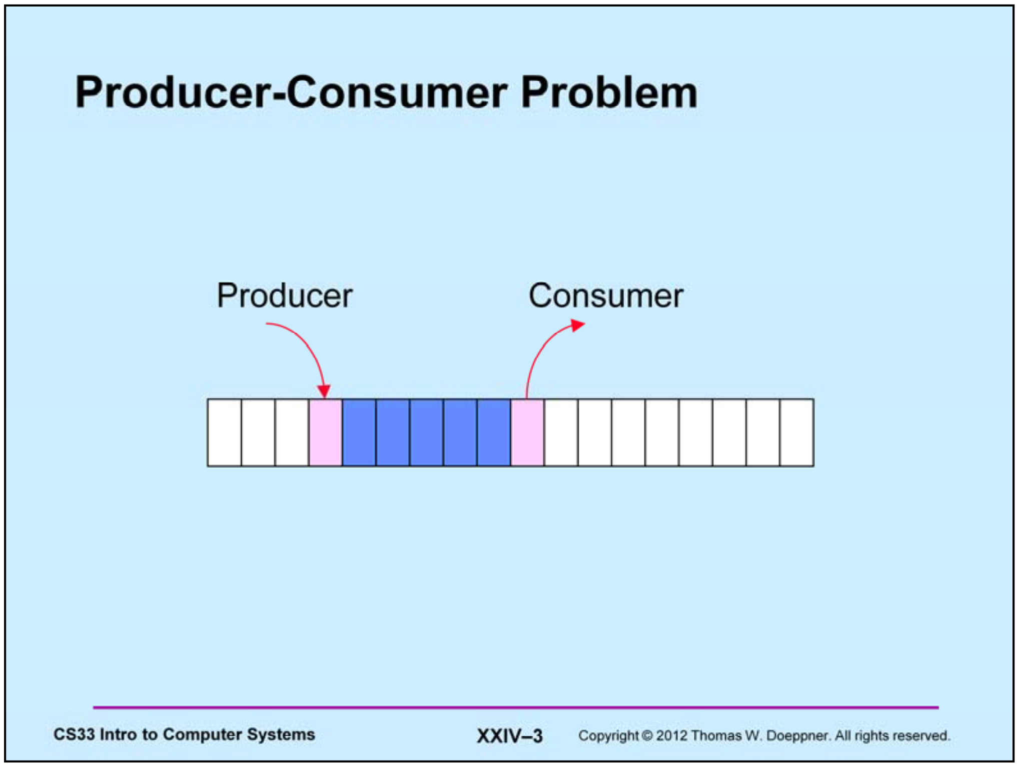


Pipes

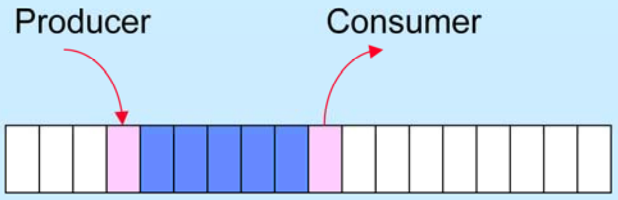
```
% who | wc -l
```

```
if (fork() == 0) {
  int fd[2];
  pipe(fd);
  if (fork() == 0) {
    close(1);
    dup(fd[1]);
    execl("/usr/bin/who", "who", (void *)0);
      // who sends its output to the pipe
  }
  if (fork() == 0) {
    close(0);
    dup(fd[0]);
    execl("/usr/bin/wc", "wc", "-l", (void *)0);
      // wc gets its input from the pipe
  }
  while(wait(0) != -1)
    ;
}
```

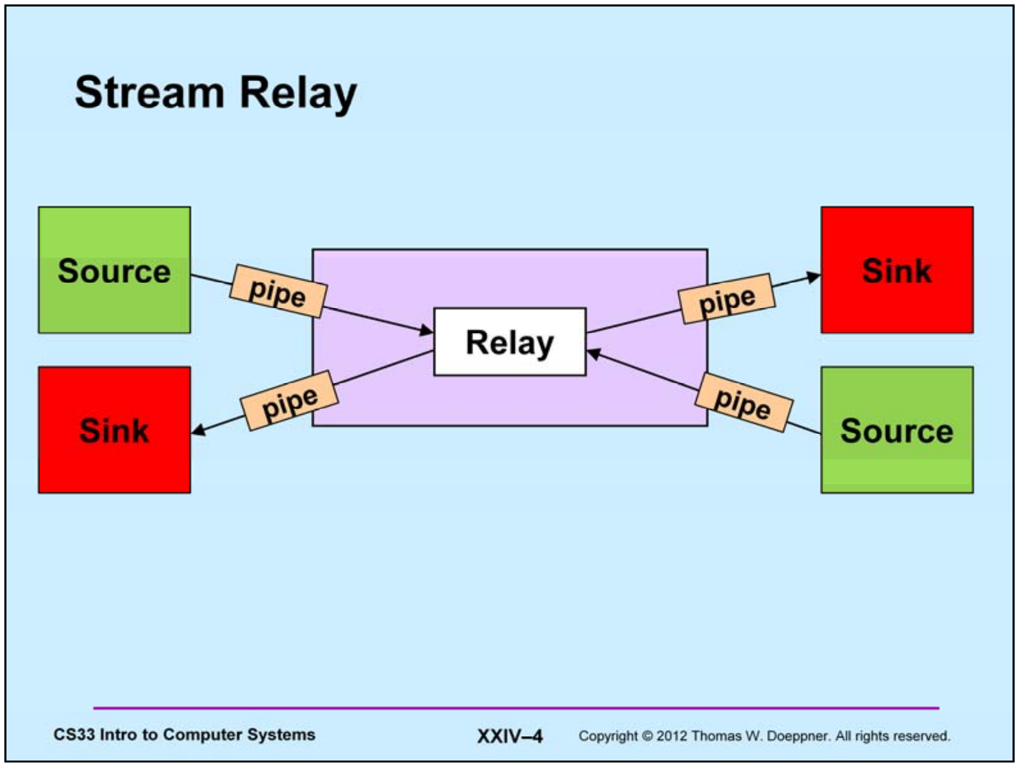
The *pipe* system call returns a pair of file descriptors in its argument, fd. The result is a one-way channel for data: what is written into fd[1] can be read from fd[0]. The operating system buffers the data, so that up to a certain amount of data can be written before any of it is consumed. If the buffer is full, then writes will block until space is freed up (by reading the data). If the buffer is empty, then reads will block until data is written to the pipe.



Producer-Consumer Problem

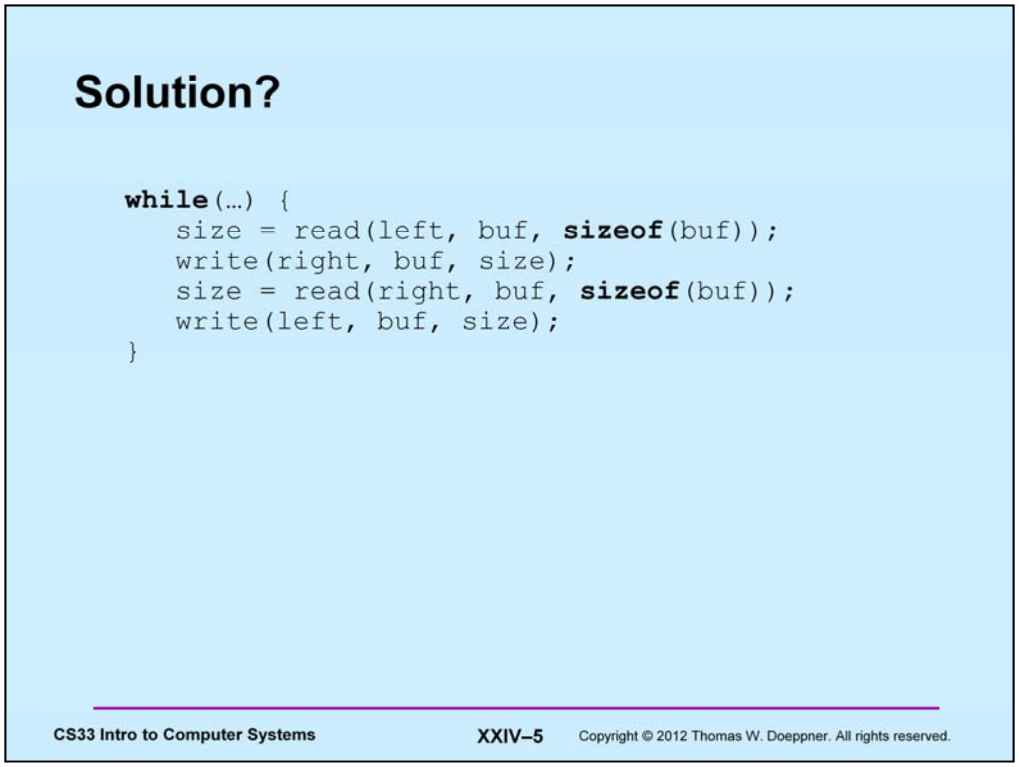


A pipe is an instance of the *producer-consumer problem.* Producers attempt to put data into a buffer starting at the next empty slot, a consumers attempt to take data out of the buffer starting at the first occupied slot. The synchronization conditions are that producers cannot proceed unless there are empty slots and consumers cannot proceed unless there are occupied slots.



Stream Relay

In the remainder of the lecture we discuss this problem: we want to write a relay program that takes data, via a pipe, from the left source and sends it, via a pipe, to the right sink. At the same time it takes data from the right source and sends it to the left sink.



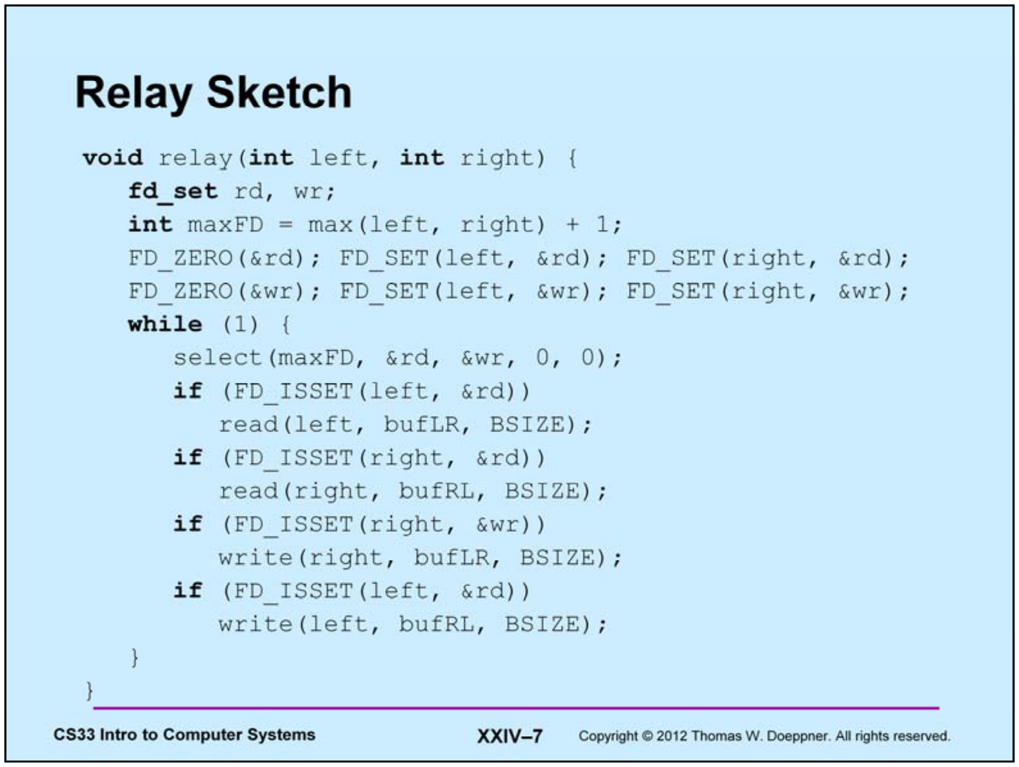
Solution?

```
while(…) {
   size = read(left, buf, sizeof(buf));
   write(right, buf, size);
   size = read(right, buf, sizeof(buf));
   write(left, buf, size);
}
```

This solution is probably not what we'd want, since it strictly alternates between processing the data stream in one direction and then the other.

Select System Call

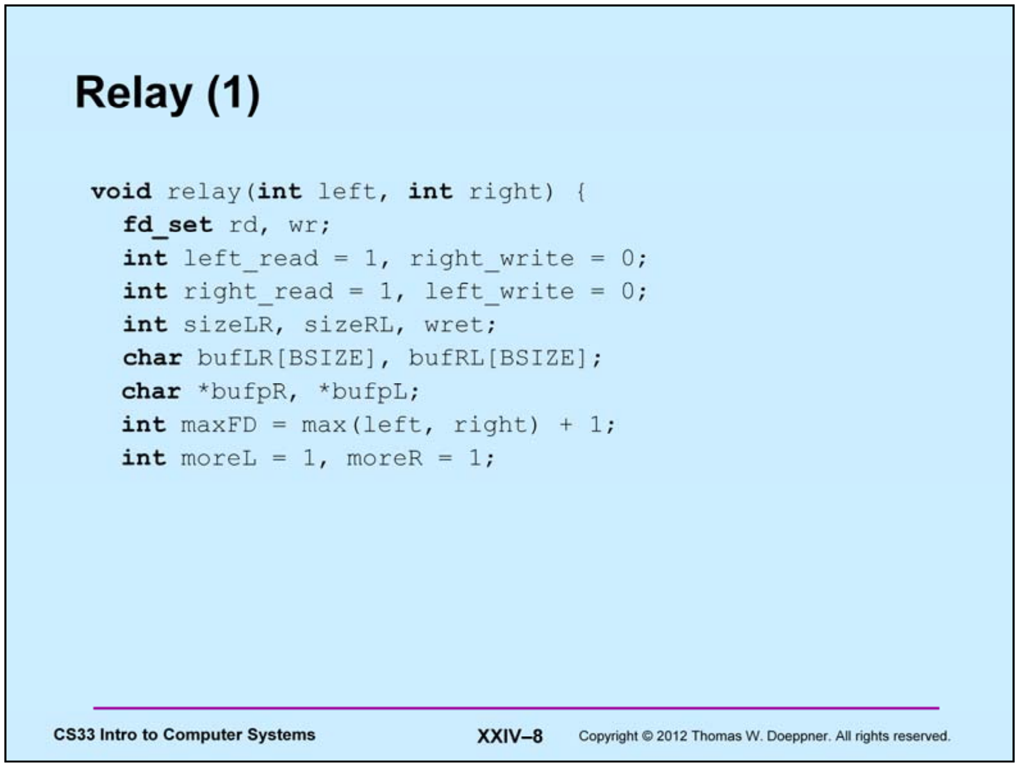
```
int select(
  int nfds,          // size of fd_sets
  fd_set *readfds,   // descriptors of interest
                     // for reading
  fd_set *writefds,  // descriptors of interest
                     // for writing
  fd_set *excpfds,   // descriptors of interest
                     // for exceptional events
  struct timeval *timeout
                     // max time to wait
);
```



Relay Sketch

```
void relay(int left, int right) {
   fd_set rd, wr;
   int maxFD = max(left, right) + 1;
   FD_ZERO(&rd); FD_SET(left, &rd); FD_SET(right, &rd);
   FD_ZERO(&wr); FD_SET(left, &wr); FD_SET(right, &wr);
   while (1) {
      select(maxFD, &rd, &wr, 0, 0);
      if (FD_ISSET(left, &rd))
         read(left, bufLR, BSIZE);
      if (FD_ISSET(right, &rd))
         read(right, bufRL, BSIZE);
      if (FD_ISSET(right, &wr))
         write(right, bufLR, BSIZE);
      if (FD_ISSET(left, &rd))
         write(left, bufRL, BSIZE);
   }
}
```

Here a simplified version of a program to handle the relay problem using select.



Relay (1)

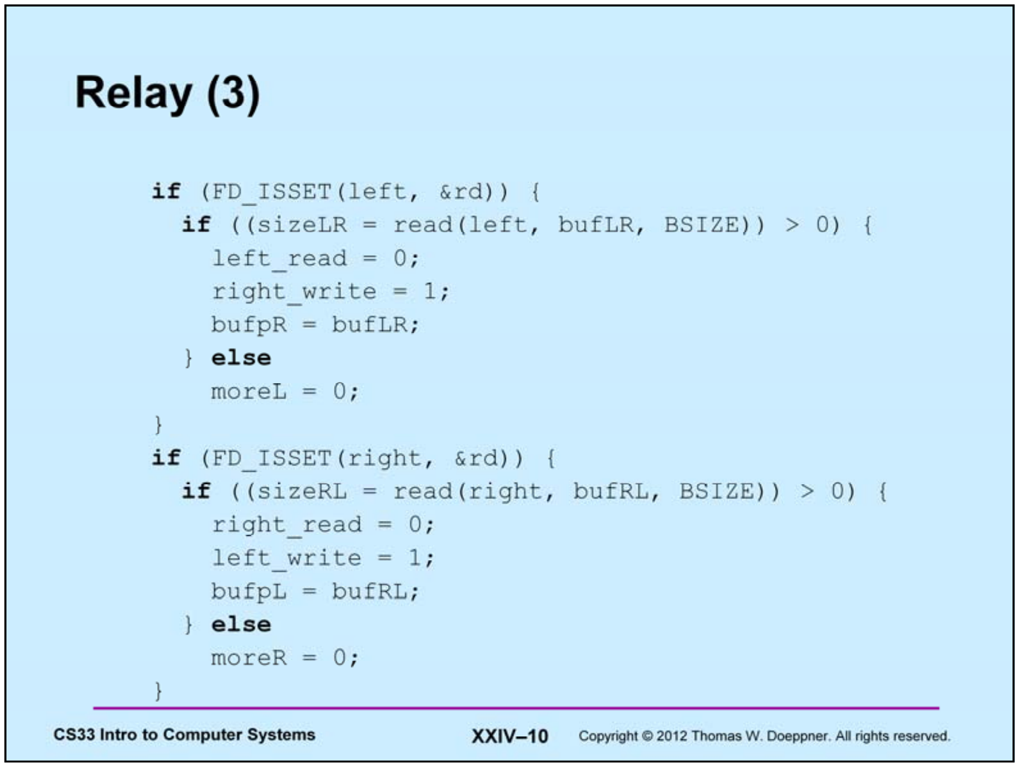
```
void relay(int left, int right) {
  fd_set rd, wr;
  int left_read = 1, right_write = 0;
  int right_read = 1, left_write = 0;
  int sizeLR, sizeRL, wret;
  char bufLR[BSIZE], bufRL[BSIZE];
  char *bufpR, *bufpL;
  int maxFD = max(left, right) + 1;
  int moreL = 1, moreR = 1;
```

This and the next three slides give a more complete version of the relay program.

Relay (2)

```
while(moreL || moreR) {
  FD_ZERO(&rd);
  FD_ZERO(&wr);
  if (left_read && moreL)
    FD_SET(left, &rd);
  if (right_read && moreR)
    FD_SET(right, &rd);
  if (left_write)
    FD_SET(left, &wr);
  if (right_write)
    FD_SET(right, &wr);

  select(maxFD, &rd, &wr, 0, 0);
```

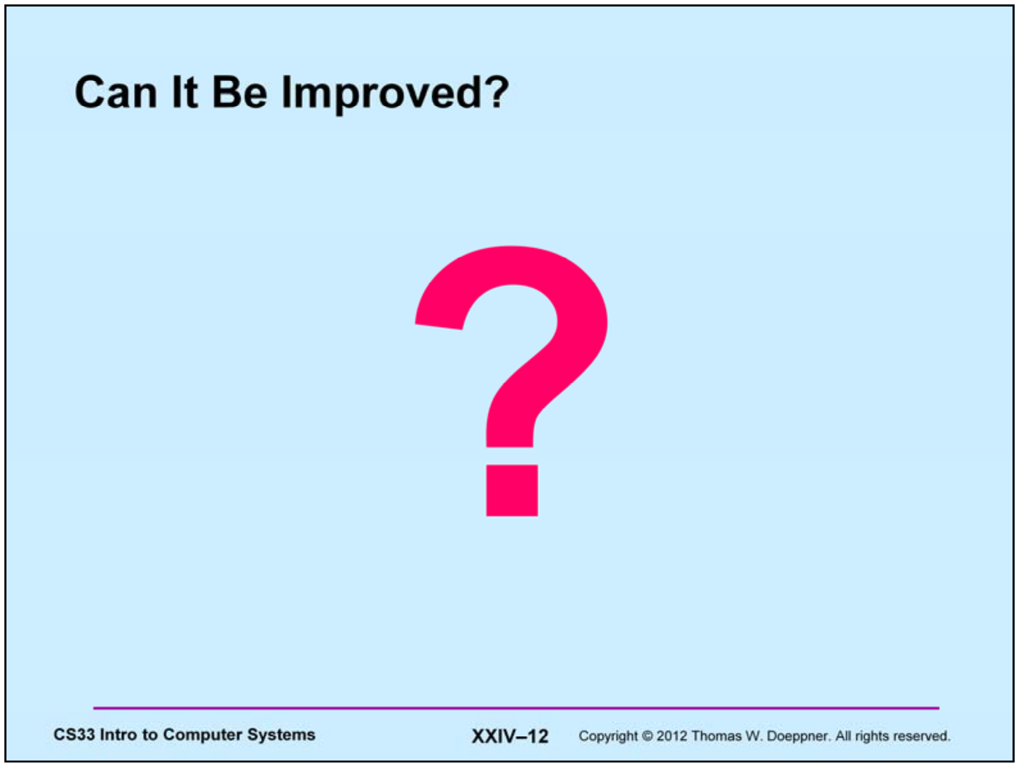


Relay (3)

```
if (FD_ISSET(left, &rd)) {
  if ((sizeLR = read(left, bufLR, BSIZE)) > 0) {
    left_read = 0;
    right_write = 1;
    bufpR = bufLR;
  } else
    moreL = 0;
}
if (FD_ISSET(right, &rd)) {
  if ((sizeRL = read(right, bufRL, BSIZE)) > 0) {
    right_read = 0;
    left_write = 1;
    bufpL = bufRL;
  } else
    moreR = 0;
}
```

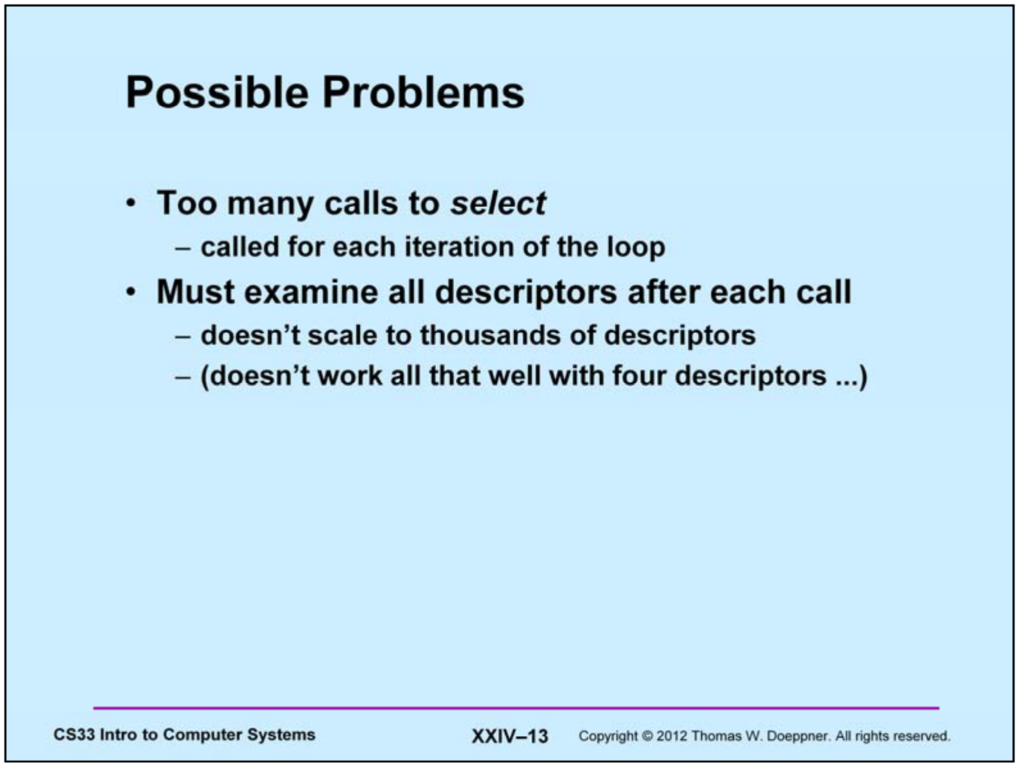
Relay (4)

```
    if (FD_ISSET(right, &wr)) {
      if ((wret = write(right, bufpR, sizeLR)) == sizeLR) {
        left_read = 1; right_write = 0;
      } else {
        sizeLR -= wret; bufpR += wret;
      }
    }
    if (FD_ISSET(left, &wr)) {
      if ((wret = write(left, bufpL, sizeRL)) == sizeRL) {
        right_read = 1; left_write = 0;
      } else {
        sizeRL -= wret; bufpL += wret;
      }
    }
  }
  return(0);
}
```



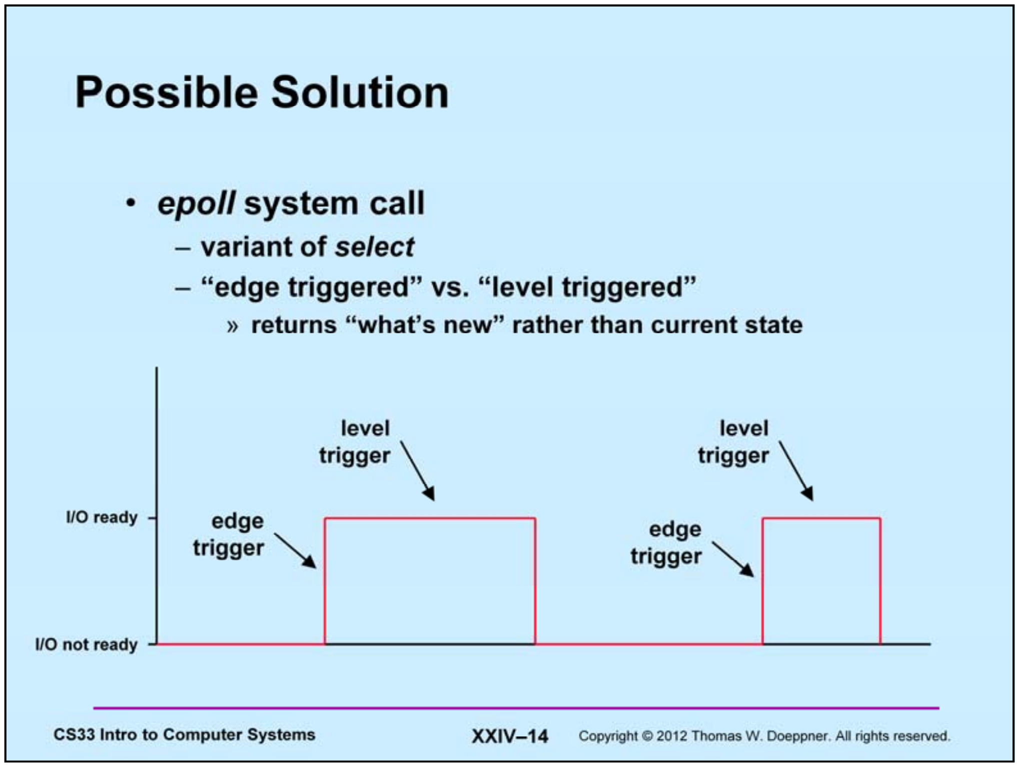
Can It Be Improved?

?



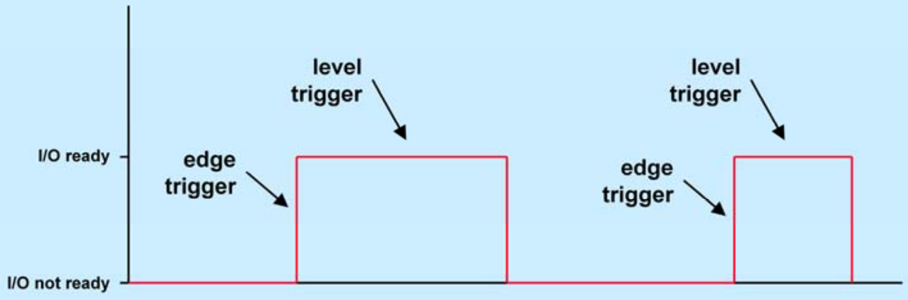
Possible Problems

- Too many calls to *select*
  - called for each iteration of the loop
- Must examine all descriptors after each call
  - doesn’t scale to thousands of descriptors
  - (doesn’t work all that well with four descriptors ...)

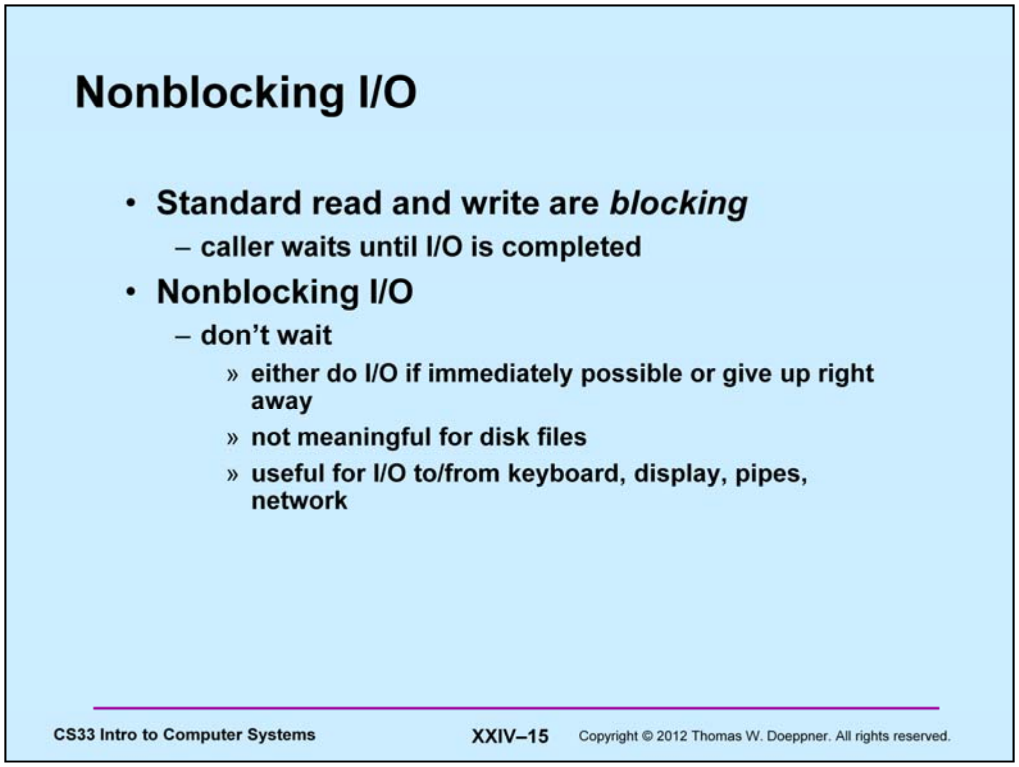


Possible Solution

- *epoll* system call
  - variant of *select*
  - "edge triggered" vs. "level triggered"
    - » returns "what's new" rather than current state



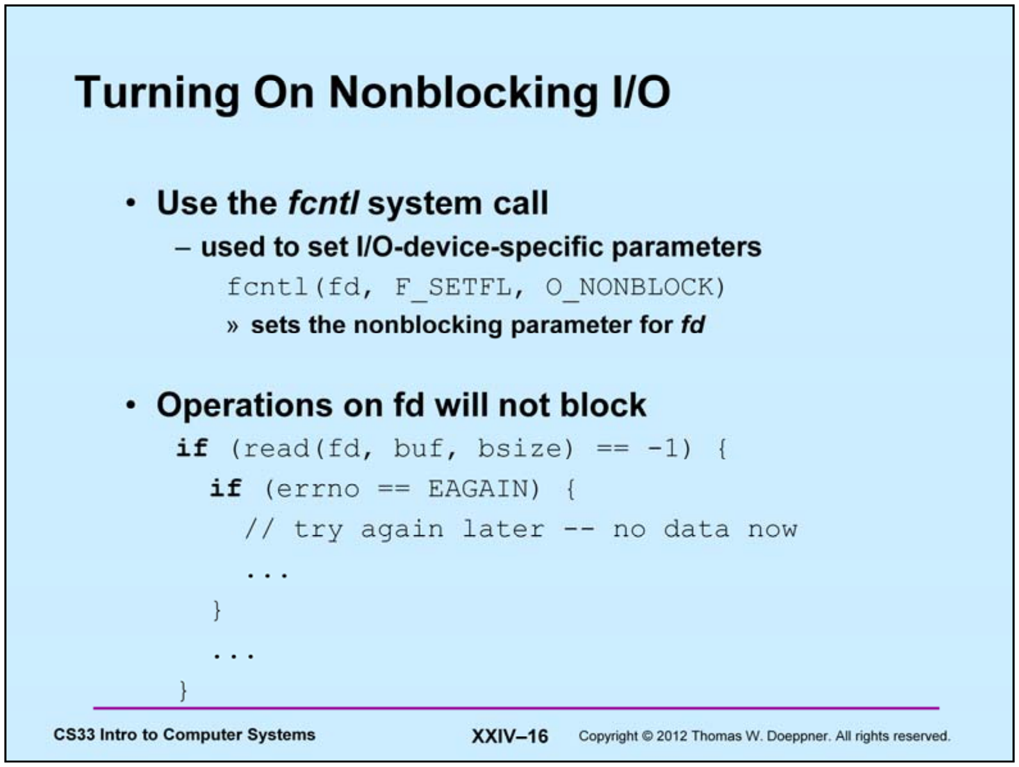
Note that epoll may be either edge-triggered or level-triggered, depending on parameters. Rather than returning a bit vector of file descriptors (as select does), it returns variable-sized vectors of event structures indicating what events have occurred. This potentially scales better than the select approach: if, say, there are 10,000 file descriptors and two events, it returns just the two events.



Nonblocking I/O

- Standard read and write are *blocking*
  - caller waits until I/O is completed
- Nonblocking I/O
  - don't wait
    - » either do I/O if immediately possible or give up right away
    - » not meaningful for disk files
    - » useful for I/O to/from keyboard, display, pipes, network

Nonblocking I/O must be used in conjunction with epoll; its edge triggering tells us when I/O becomes possible, but it doesn't tell us when its no longer possible. By setting the file descriptors to be non-blocking, the caller won't wait when attempting I/O on a descriptor that has become not ready for I/O.



Turning On Nonblocking I/O

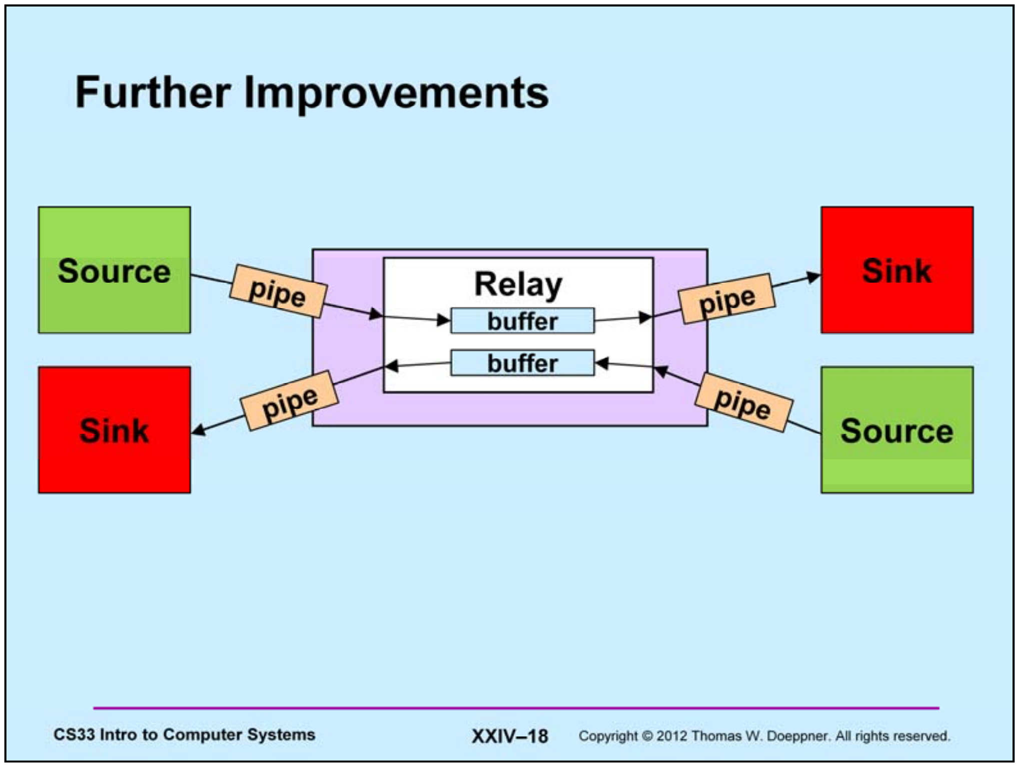
- Use the *fcntl* system call
  - used to set I/O-device-specific parameters

    ```
    fcntl(fd, F_SETFL, O_NONBLOCK)
    ```
    - sets the nonblocking parameter for *fd*
- Operations on fd will not block

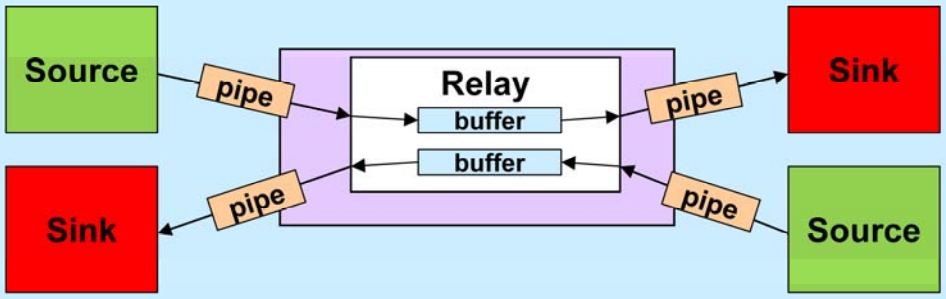
```
if (read(fd, buf, bsize) == -1) {
  if (errno == EAGAIN) {
    // try again later -- no data now
    ...
  }
  ...
}
```

Event-Based Programming

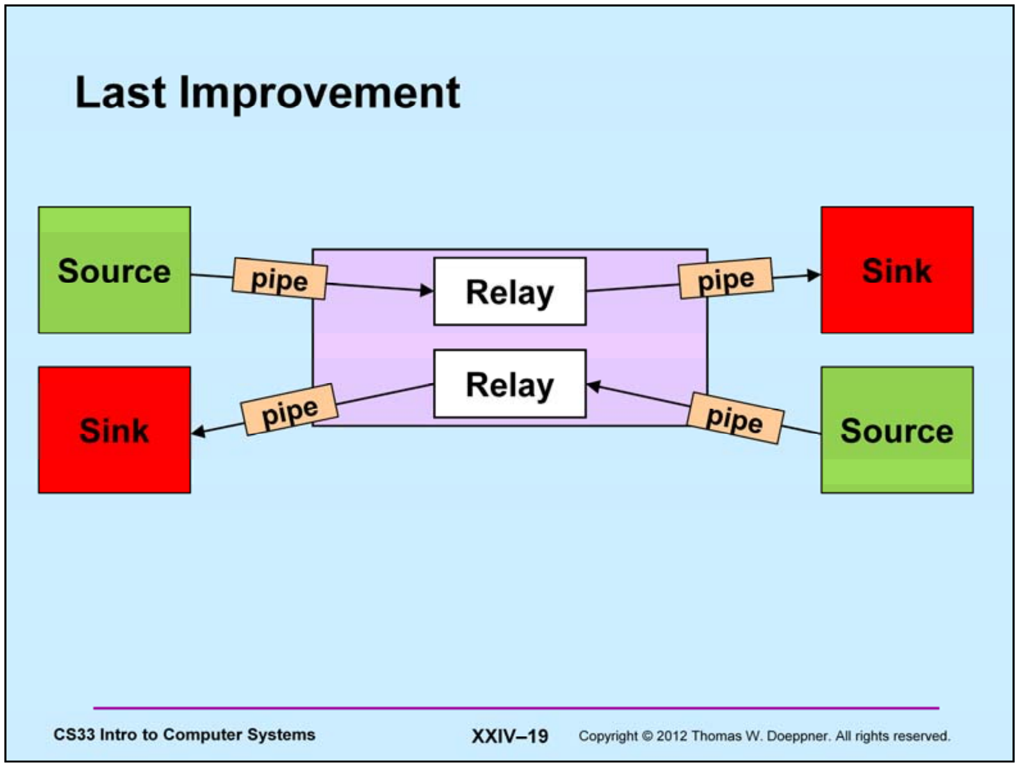
- Program is structured around the processing of events
  - as in the select and epoll versions of stream-relay
  - wait for events to occur, then respond to them
  - straightforward if events are independent
  - not so straightforward if not
    - see epoll-based solution for stream-relay



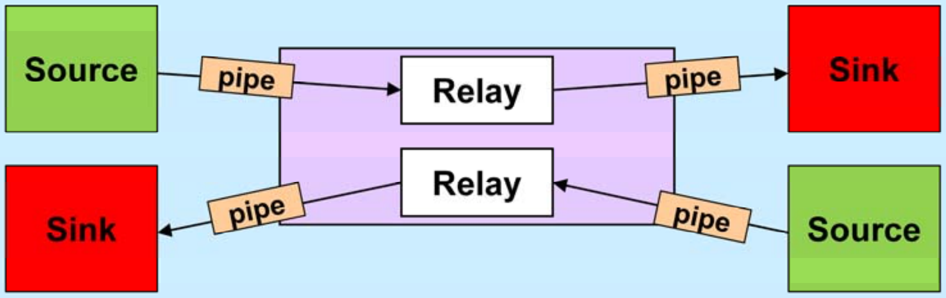
Further Improvements



Our next improvement is to add a pair of buffers to the relay program itself, creating two more instances of the producer-consumer scenario. This makes it possible, for example, for the source side to continue to copy data into the buffer, even if the sink side is temporarily held up.



Last Improvement



The last improvement is to split the relay program into two processes, one for each direction.