CSC209: Software Tools and Systems Programming

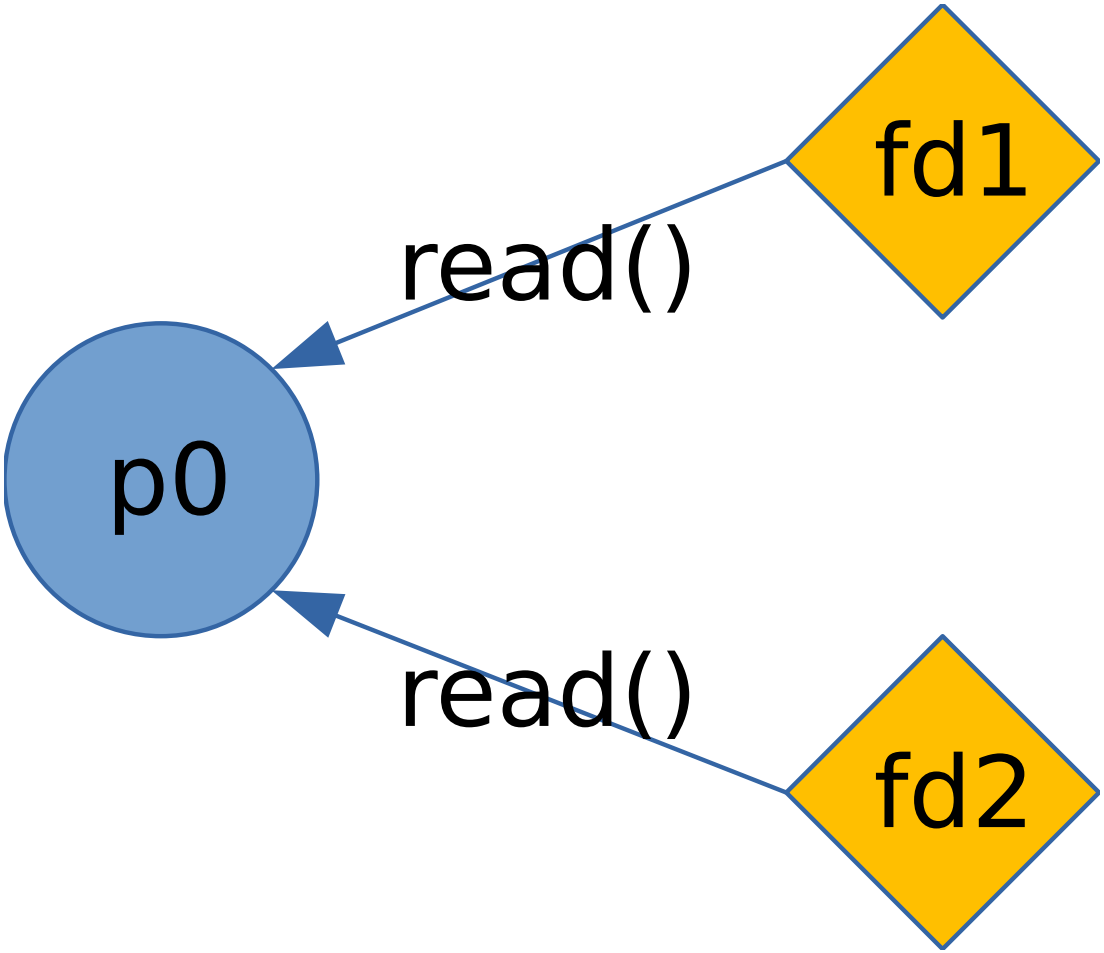
Week 11: Multiplexing I/O [1]

Kianoosh Abbasi

[1] Slides are mostly taken from Andi Bergen's in summer 2021.

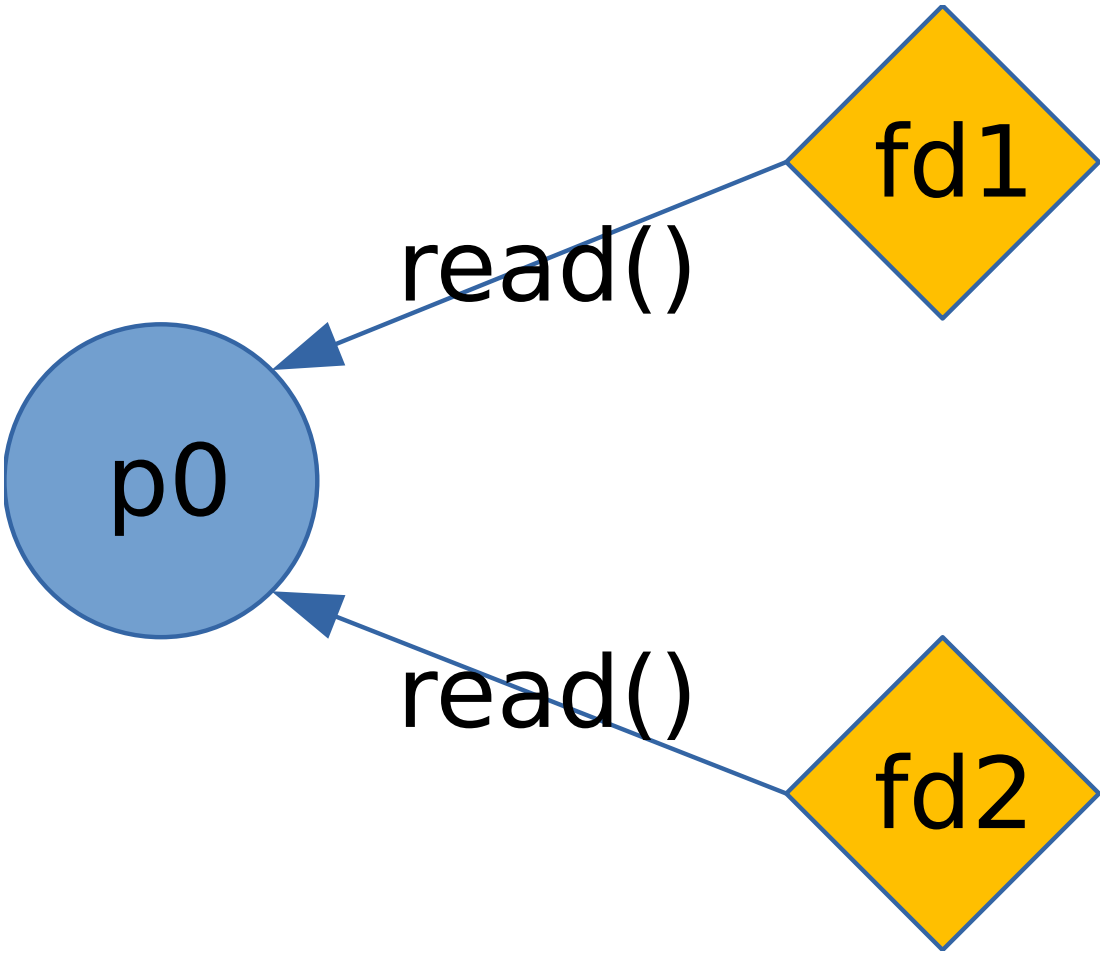
Reading From Multiple File Descriptors

Assume that a process p0 has any two file descriptors open for reading (e.g., from a socket, regular file, pipe)



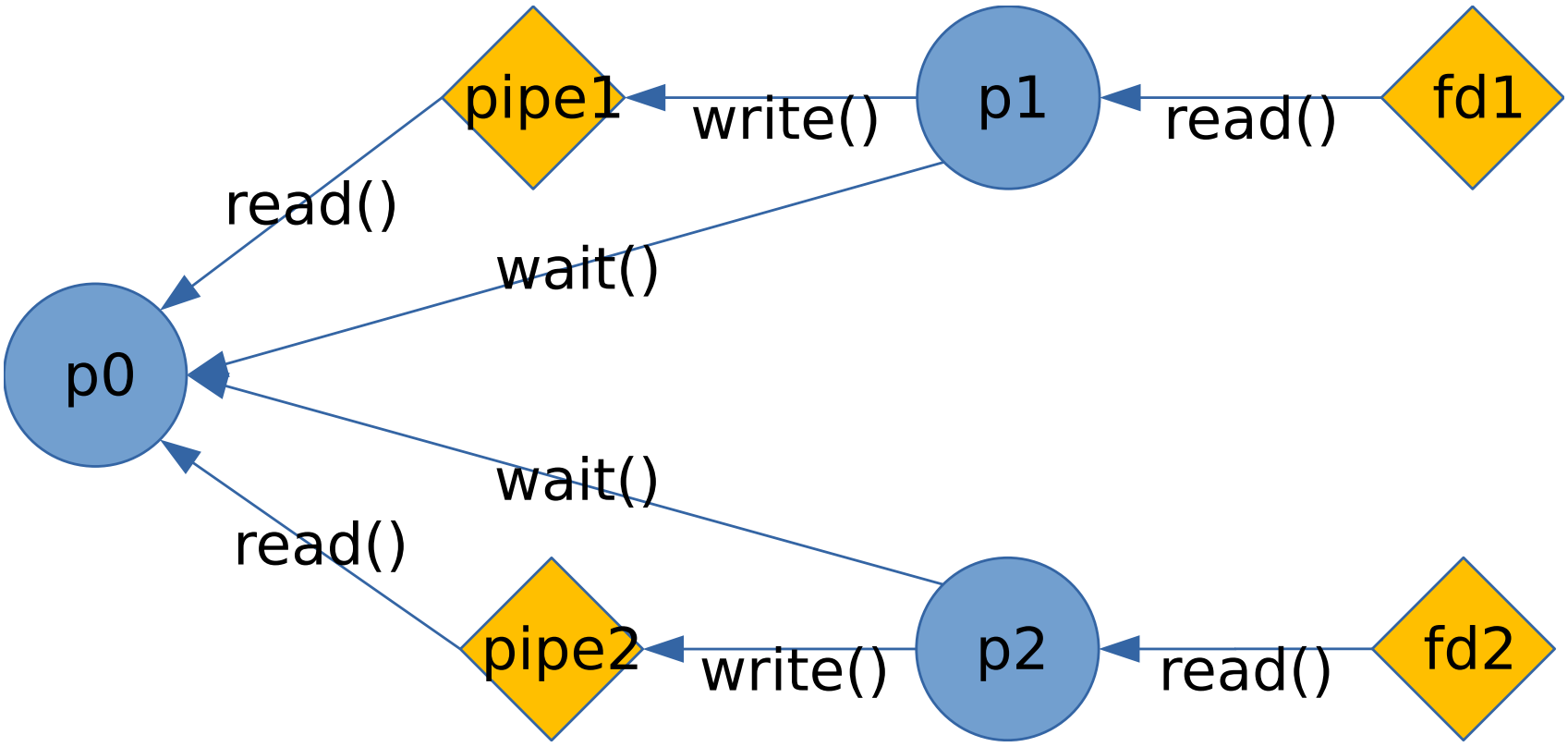
Reading From Multiple File Descriptors...

If `p0` reads from `fdN`, it will block until `fdN` has data ready to read. But what if the other fd already has data available to be read?



Solution 1: Fork

`p0` can `fork` one child process per file descriptor to be read from; each child calls `read` on one file descriptor and communicates data to parent over a pipe.



Which should be called first: `read()` or `wait()`?

Solution 1: Fork (With Sockets)

- It is common for server software to `fork()` a new process for each client that connects: SSH does exactly that
- *Performance* benefit: Solves the issue of blocking `read()` calls that we just discussed
- *Security* benefit: Each process has its own memory space, making it less likely for there to be a bug that allows one user to read confidential information that belongs to another user
- Drawback: Each process takes up memory

Solution 2: Select

- `select()` monitors several file descriptors simultaneously, without needing to `fork()`
- `select()` **blocks** until at least one of the monitored file descriptors is "ready"
- A file descriptor "ready" for reading means that `read()` can be called **once** without blocking
  - Calling `read()` more than once can block
- `select()` also returns on some other conditions, e.g., if a signal is received, or if a predefined timeout period expires

Select: Parameters

```
int select(int nfds, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);
```

- `select()` returns the number of FDs that are ready, or returns `-1` on error
- Set `nfds` to one more than the highest-numbered FD of interest
  - e.g. if you are interested in FDs 3, 9, and 50, pass `51` for `nfds`

Select: Parameters...

- `select()` takes three file descriptor sets as part of its input:
  - First set is monitored for *reading*
  - Second set is monitored for *writing*
  - Third set is monitored for *exceptions*
- The sets are *modified* by `select()` to contain only the FDs with action of the requested type

For more about exceptions: An example of what would trigger an exception

Select and the Listener Socket

- Recall that `accept()` is a blocking call that returns after a new incoming connection is added to the listener socket's queue
- The listener socket file descriptor can be monitored with `select()`: It is considered "ready to read" when there is at least one connection queued up to be accepted

File Descriptor Sets

File descriptor sets are similar to signal sets. Use these macros to operate on them:

1. `FD_ZERO()`: Empty the set
2. `FD_SET()`: Add file descriptor to the set
3. `FD_CLR()`: Remove file descriptor from the set
4. `FD_ISSET()`: Check file descriptor's membership in a set

Select: The Bottom Line

- The bottom line is that we **never** want to block on any calls to `read()` or `accept()`
  - Otherwise, we risk the possibility of waiting forever, even when there might be data ready to be read from other file descriptors
- Instead, we write our client/server programs to block **only** on `select()`

Assignment 3 Debugging Tip

Use the `strace` utility on your client and/or server: If it ever blocks on `read()` or `accept()`, it means there is a bug in your program

Question

Your Assignment 3 server will be written to not block on any `read()` calls, but what if it blocks on `write()`?

Answer

We've only scratched the surface with this course. When might a server block on `write()`? Hint: See "extra slide" on flow control from last week. What does TCP do if a server is sending data faster than a client can handle?

Solution: Use `select()` for both reads and writes. (but don't do this for A3)

Level- and Edge-Triggering

"When is an FD ready?"

Two answers:

1. *Level triggered*: when an operation (e.g. `read`) won't block, or
2. *Edge triggered*: when there is new action on the FD since the last time you asked

- `select()` is level triggered
  - If you don't read everything, 'select| will keep telling you that the FD is ready

Limitations of Select

- `select()` has some performance limitations, and can only monitor at most `FD_SETSIZE` (1024, on Linux) file descriptors
  - But it is portable!
- There are Linux-specific (not portable) alternatives that are more efficient
- Common tradeoff: portability vs. efficiency

Reading From Clients

- When a server does a `read`, it is not guaranteed to get a complete line or all of the desired bytes
  - e.g. the client could be sending each character separately
  - e.g. the client could send data that gets split over several segments
- Want to operate only on full lines? The server must keep each partial line in a buffer until it gets the newline from the client
- The following code assumes there's at most one line in the buffer
  - OK for obtaining the filename in A3... not OK in general!

Buffering for Full Lines

The server should keep a buffer for each client, and keep track of the number of bytes in each buffer following the previous message.

```
struct client {
  int fd;
  char buf[300];
  int inbuf;
  struct client *next;
};
```

Buffering for Full Lines...

Read bytes, check for errors, and null-terminate the string.

```
void myread(struct client *p) {
  char *startbuf = p->buf + p->inbuf;
  int room = sizeof (p->buf) - p->inbuf;
  int crlf;
  char *tok, *cr, *lf;

  if (room <= 1)
    // clean up this client: buffer full
  int len = read (p->fd, startbuf, room - 1);
  if (len <= 0)
    // Clean up this client: eof or error
  p->inbuf += len;
  p->buf[p->inbuf] = '\0';
```

Buffering for Full Lines...

If a full line exists, process it and shift it out of the buffer.

```
  lf = strchr(p->buf, '\n');
  cr = strchr(p->buf, '\r');
  if (!lf && !cr)
    return; //No complete line
  tok = strtok(p->buf, "\r\n");
  if (tok)
    // use tok (complete string)
  if (!lf)
    crlf = cr - p->buf;
  else if (!cr)
    crlf = lf - p->buf;
  else
    crlf = (lf > cr) ? lf - p->buf : cr - p->buf;
  crlf++;
  p->inbuf -= crlf;
  memmove(p->buf, p->buf + crlf, p->inbuf);
}
```

Extra Slides

Don't use these for A3!

SIGPIPE

Writing to a broken pipe/socket generates a `SIGPIPE`.

Recall: By default, most signals (including `sigpipe`) will terminate your program.

Here's how you can protect against `sigpipe`:

```
/*
 * Turn off SIGPIPE: write() to a socket that
 * is closed on the other end will return -1
 * with errno set to EPIPE, instead of generating
 * a SIGPIPE signal that terminates the process.
 */
if (signal(SIGPIPE, SIG_IGN) == SIG_ERR) {
    perror("signal");
    exit(1);
}
```

Non-Blocking Reads

- You can change the behaviour of `read()` so that it returns `-1` and sets `errno` to `EAGAIN` if no data is available.
- In this mode, `read()` will never block
- Downside: Leads to inefficient code, e.g., using an infinite loop that repeatedly calls `read()`
  - Remember, `read()` will return **immediately** in non-blocking mode, so you will be calling it **many** times per second

Non-Blocking Reads: Sample Code

```
char buf[1024];
ssize_t bytesread;
/* set O_NONBLOCK flags on fd1 and fd2 */
if (fcntl(fd1, F_SETFL, O_NONBLOCK) == -1) {
  perror("fcntl"); exit(1);
}
if (fcntl(fd2, F_SETFL, O_NONBLOCK) == -1) {
  perror("fcntl"); exit(1);
}

for ( ; ; ) {
  bytesread = read(fd1, buf, sizeof(buf));
  if ((bytesread == -1) && (errno != EAGAIN))
    return;     /* real error */
  else if (bytesread > 0)
    doSomething(buf, bytesread);
  bytesread = read(fd2, buf, sizeof(buf));
  ...
```

Non-Blocking Reads: Sample Code (cont.)

```
...
if ((bytesread == -1) && (errno != EAGAIN))
  return;     /* real error*/
else if (bytesread > 0)
  doSomething(buf, bytesread);
}
```