System and Device Programming

I/O Multiplexing

Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

I/O Multiplexing

- Let us suppose we must read from one descriptor and write to another

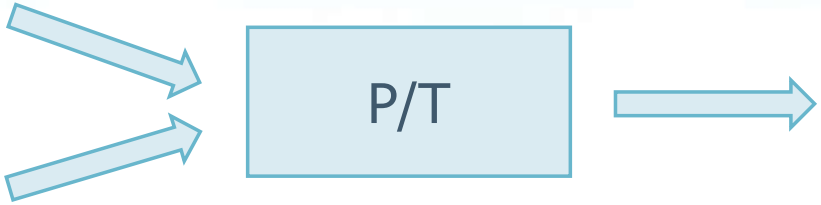
  - We can use blocking I/O in a loop

```
while ((n = read(STDIN_FILENO, buf, BUFSIZ)) > 0)
  if (write(STDOUT_FILENO, buf, n) != n)
    err_sys("write error");
```

  - This form of blocking I/O is very common

I/O Multiplexing

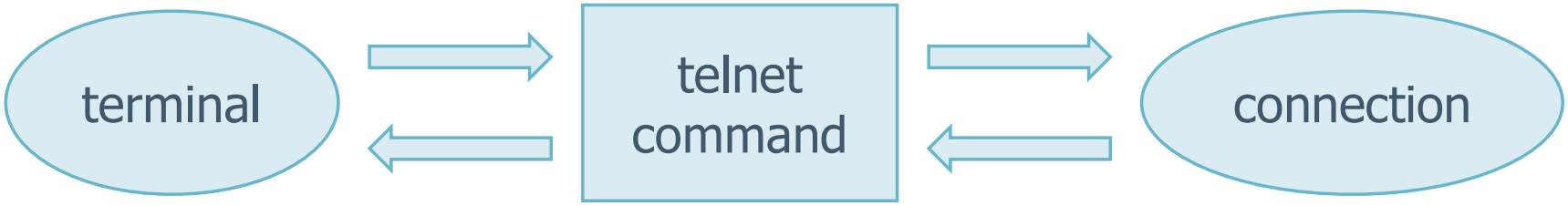
- How can we read from two descriptors?



  - We cannot do a blocking read on either descriptor
    - Data may appear on **one** descriptor while we are blocked in a read on the **other**
    - We need a different technique to handle this case and we have several possibilities
      - Use different processes or different threads
      - Use non-blocking I/O
      - Use asynchronous I/O
      - Use I/O multiplexing

Example

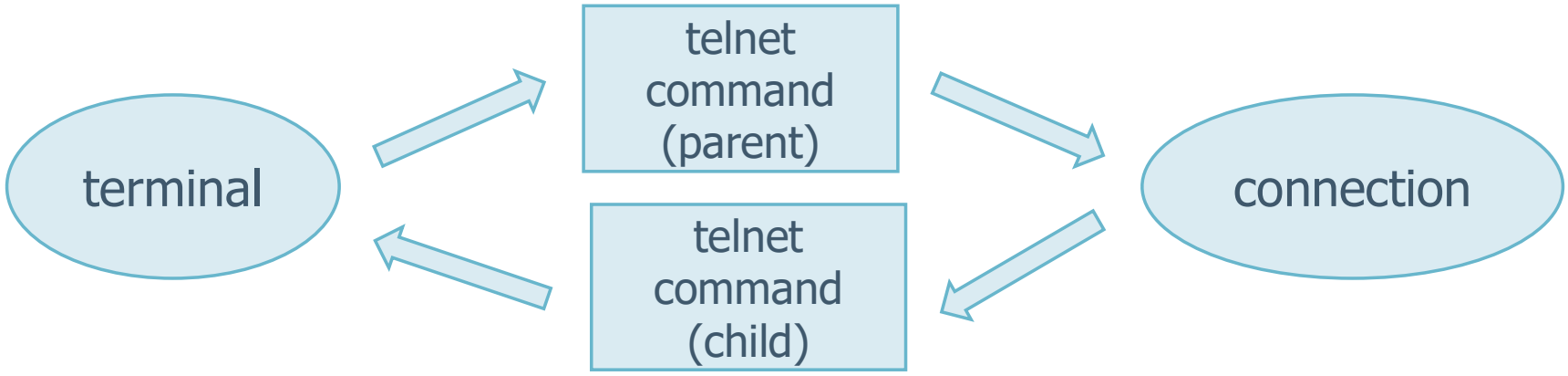
- A telnet command
  - Reads from the terminal (standard input) and write to a network connection
  - Reads from the network connection and write to the terminal (standard output)



  - The telnet process has two inputs and two outputs
    - We can't do a blocking read on either of the inputs, as we never know which input will have data for us

Solution 1

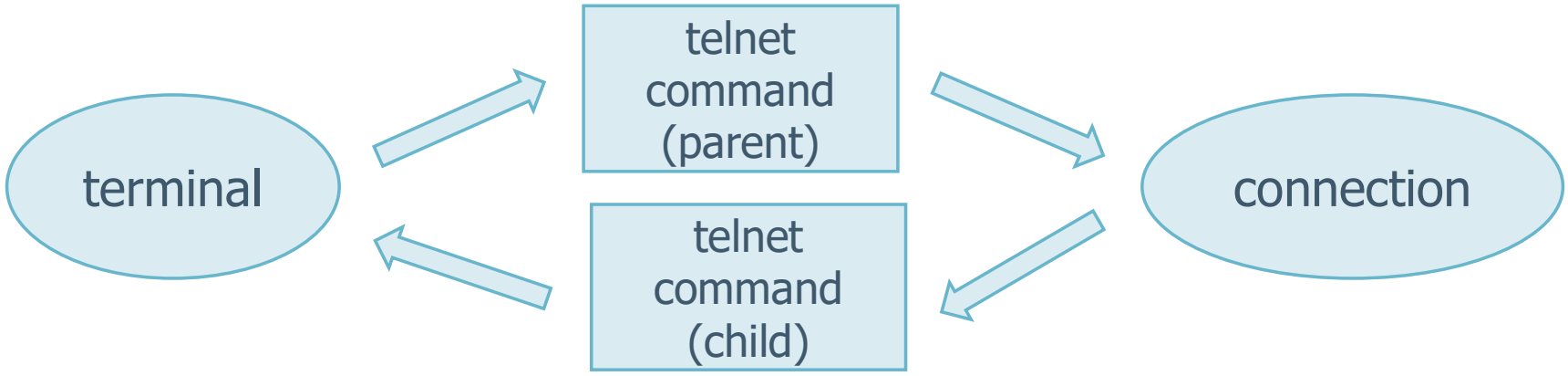
- The multi-process approach to multiplexing
  - Fork the process, with the parent handling one direction of data and the child the other one



  - Each process can perform blocking read
  - When one process receives and EOF must stop the other one (e.g., sending a signal)

Solution 2

- The multi-threads approach to multiplexing
  - Same as before but two threads instead of two processes



  - This avoids the termination complexity
  - It requires that we deal with synchronization between the threads, which could add more complexity than it saves

Solution 3

- The non-blocking I/O approach to multiplexing
  - We use nonblocking I/O in a single process
    - We set both descriptors to be nonblocking
    - We issue a read on the first descriptor
      - If data is present, we read it and process it
      - If there is no data to read, the call returns and we do the same on the second descriptor
    - We wait for some amount of time and then we try to read from the first descriptor again
  - This type of loop is called **polling**
    - Most of the time, there won't be data to read, so we waste time performing the read system calls
    - We have to guess how long to wait before looping

Solution 4

Please, see Section u04s08

- The asynchronous I/O approach to multiplexing
  - We tell the kernel to notify us with a signal when a descriptor is ready for I/O
  - This approach has two problems
    - Portability can be an issue
      - Different systems provide their own limited forms of asynchronous
        - POSIX proposes one strategy
        - System V provides the SIGPOLL
        - BSD has SIGIO
        - etc.

Solution 4

- The number of signals allowed for each process can be a problem
  - A few implementation of asynchronous I/O allow only one signal
    - POSIX allows us to select which signal to use for the notification
    - Anyway, the number of signals we can use is still far less than the number of possible open file descriptors
  - To determine which descriptor is ready, we would need to set each file descriptor to nonblocking mode and try the descriptors in sequence
    - This is again, a form of **polling**

Solution 5

- The "I/O multiplexing" approach to multiplexing
  - The idea is to use a function that
    - Accepts more than one I/O descriptor
    - Doesn't return until one of the descriptors is ready for I/O
    - On return, it tells us which descriptors are ready for I/O

Solution 5

- The "I/O multiplexing" logic flow
  - We build a list of the descriptors that we are interested in
    - Usually more than one descriptor
    - We need to perform I/O on each one of them
  - We use a multiplexing I/O function
    - Three functions allow us to perform I/O multiplexing
      - select, pselect, poll

Functions pselect and poll are variation of select

    - They differ from the user interface point of view
    - Include POSIX <sys/select.h>
      - Or <sys/types.h>, <sys/time.h>, and <unistd.h>

The select function

```
#include <sys/select.h>
int select (
  int maxfdp1,
  fd_set *restrict readfds,
  fd_set *restrict writefds,
  fd_set *restrict exceptfds,
  struct timeval *restrict tvptr
);
```

- Function select allows I/O multiplexing under all POSIX-compatible platforms
- The arguments tell the kernel
  - Which descriptors we are interested in
  - Which conditions we focus on (for each descriptor)
  - How long we want to wait

The select function

- The last argument specifies how long we want to wait in terms of seconds and microseconds
  - If tvptr == NULL
    - We wait forever
    - If a signal interrupts the function, select returns −1 with errno set to EINTR
  - If tvptr−>tv_sec == 0 && tvptr−>tv_usec == 0
    - We do not wait at all
    - All the specified descriptors are tested, and return is made immediately
    - This is a way to poll

```
int select (
  int maxfdp1,
  fd_set *restrict readfds,
  fd_set *restrict writefds,
  fd_set *restrict exceptfds,
  struct timeval *restrict tvptr
);
```

The select function

- If tvptr->tv_sec != 0 || tvptr->tv_usec != 0
  - We wait the specified number of seconds and microseconds
  - Return is made when one of the specified descriptors is ready or when the timeout value expires
  - If the timeout expires before any of the descriptors is ready, the return value is 0

```
int select (
  int maxfdp1,
  fd_set *restrict readfds,
  fd_set *restrict writefds,
  fd_set *restrict exceptfds,
  struct timeval *restrict tvptr
);
```

The select function

- The middle three arguments are pointers to descriptor sets
  - They specify the descriptors we check and for which conditions
    - Readable, writable, or exceptions
    - Any of them can be null pointers if we are not interested
  - A descriptor set is stored in an **fd_set** data type
    - We can consider it to be just an array of bits
    - If all three pointers are NULL, then we have a higher-precision timer (provided by sleep)

```
int select (
  int maxfdp1,
  fd_set *restrict readfds,
  fd_set *restrict writefds,
  fd_set *restrict exceptfds,
  struct timeval *restrict tvptr
);
```

The select function

- Objects of type fd_set data type can be manipulated by a function (or macro) to
  - Set to all zero bits, by FD_ZERO
  - Turn on a single bit in a set, by FD_SET
  - Clear a single bit, by FD_CLR
  - Test whether a given bit is turned on, by FD_ISSET

```
#include <sys/select.h>

void FD_SET (int fd, fd_set *fdset);
void FD_CLR (int fd, fd_set *fdset);
void FD_ZERO (fd_set *fdset);
int FD_ISSET (int fd, fd_set *fdset);
```

Example

- The following code show how to prepare a read set

```
fd_set rset;
int fd;

FD_ZERO (&rset);
FD_SET (fd, &rset);
FD_SET (STDIN_FILENO, &rset);
```

We define the set

We zero the set

We set bits for the descriptor we are interested in

- Once, select returns, we tests whether a given bit is still set

```
if (FD_ISSET(fd, &rset)) {
    ...
}
```

The select function

- The first argument to select stands for
  - The "maximum file descriptor plus 1."
  - We compute the highest descriptor that we are interested in, considering all three of the descriptor sets, add 1
    - We could just set the first argument to FD_SETSIZE, a constant in <sys/select.h> that specifies the maximum number of descriptors (often 1,024), but this value is too large for most applications

```
int select (
  int maxfdp1,
  fd_set *restrict readfds,
  fd_set *restrict writefds,
  fd_set *restrict exceptfds,
  struct timeval *restrict tvptr
);
```

The select function

- There are three possible return values
  - A value of −1 means that an error occurred
  - A return value of 0 means that no descriptors are ready
    - This happens if the time limit expires before any of the descriptors are ready
    - When this happens, all the descriptor sets will be zeroed out

```
int select (
  int maxfdp1,
  fd_set *restrict readfds,
  fd_set *restrict writefds,
  fd_set *restrict exceptfds,
  struct timeval *restrict tvptr
);
```

The select function

- A positive return value specifies the number of descriptors that are ready
  - This value is the sum of the number of descriptors ready in all three sets
    - If the same descriptor is ready to be read and written, it will be counted twice in the return value
  - The only bits left on in the three descriptor sets are the bits corresponding to the descriptors that are ready

```
int select (
  int maxfdp1,
  fd_set *restrict readfds,
  fd_set *restrict writefds,
  fd_set *restrict exceptfds,
  struct timeval *restrict tvptr
);
```

The select function

- A descriptor is consider as "ready"
  - If a read from that descriptor won't block (if it is in the read set, readfds)
  - If a write from that descriptor won't block (if it is in the write set, writefds)
  - If an exception condition is pending on that descriptor (if it is in the exception set, exceptfds)
    - File descriptors for regular files always return ready for reading, writing, and exception conditions

```
int select (
  int maxfdp1,
  fd_set *restrict readfds,
  fd_set *restrict writefds,
  fd_set *restrict exceptfds,
  struct timeval *restrict tvptr
);
```