System-Programmierung 12: Terminals

Slides: tmb.gr/syspr-12

Überblick

Diese Lektion behandelt das Thema *Terminals*.

Wie Input vom Terminal zu Prozessen kommt.

Wie Steuerzeichen (um-)konfiguriert werden.

Sowie zeilen- und zeichenweise Verarbeitung.

Terminals

Terminals ermöglichen Input und Output von ASCII-Zeichen an der Schnittstelle von User und Computer.

Historisch erfolgte der Zugriff auf UNIX Systeme via Serial RS-232 Verbindung. Terminals waren Röhren-Bildschirme mit Tastatur, wie das DEC VT100. Diese waren typischerweise 24 Zeilen zu 80 Zeichen gross.

Heute emulieren *Pseudoterminals* diese Funktion.

TTY Devices

Schon auf frühen UNIX Systemen wurden "Teletype" Geräte, d.h. Fernschreiber, mit /dev/tty*n* bezeichnet.

Auf Linux sind /dev/tty*n* Devices virtuelle Konsolen.

Besonders am Anfang waren Terminal Geräte nicht standardisiert, Zeichenfolgen um z.B. den Cursor zu bewegen, waren je nach Gerät verschieden.

curses Library

Um geräteunabhängige Terminal-Programmierung zu ermöglichen, wurde die *curses* Library entwickelt.

Diese Bibliothek ist also eine Art Treiber für Terminal Geräte. Heute ist sie nützlich, um ASCII-basierte UIs zu entwickeln, z.B. für embedded Linux Computer.

Auf Linux heisst die (*new*) *curses* Library ncurses.

Hands-on, 15': *curses* Library

Kompilieren und testen Sie die Beispielprogramme:

```
$ sudo apt-get install libncurses5-dev
$ gcc -o NAME NAME.c -lcurses
```

Schreiben Sie ein eigenes Programm mit *ncurses*.

Mehr dazu in diesen Tutorials zur *curses* Bibliothek http://heather.cs.ucdavis.edu/~matloff/UnixAndC/CLanguage/Curses.pdf und Games mit ncurses.

Input Modes

Terminals arbeiten in einem von zwei *Input Modes*:

Im *Canonical Mode* wird Terminal Input zeilenweise verarbeitet, nach dem Drücken der ENTER Taste. Ein *read()* blockiert jeweils, bis eine ganze Zeile bereit ist.

Im *Noncanonical Mode* wird Terminal Input zeichenweise gelesen, ohne ENTER, z.B. in Editoren wie *vi*.

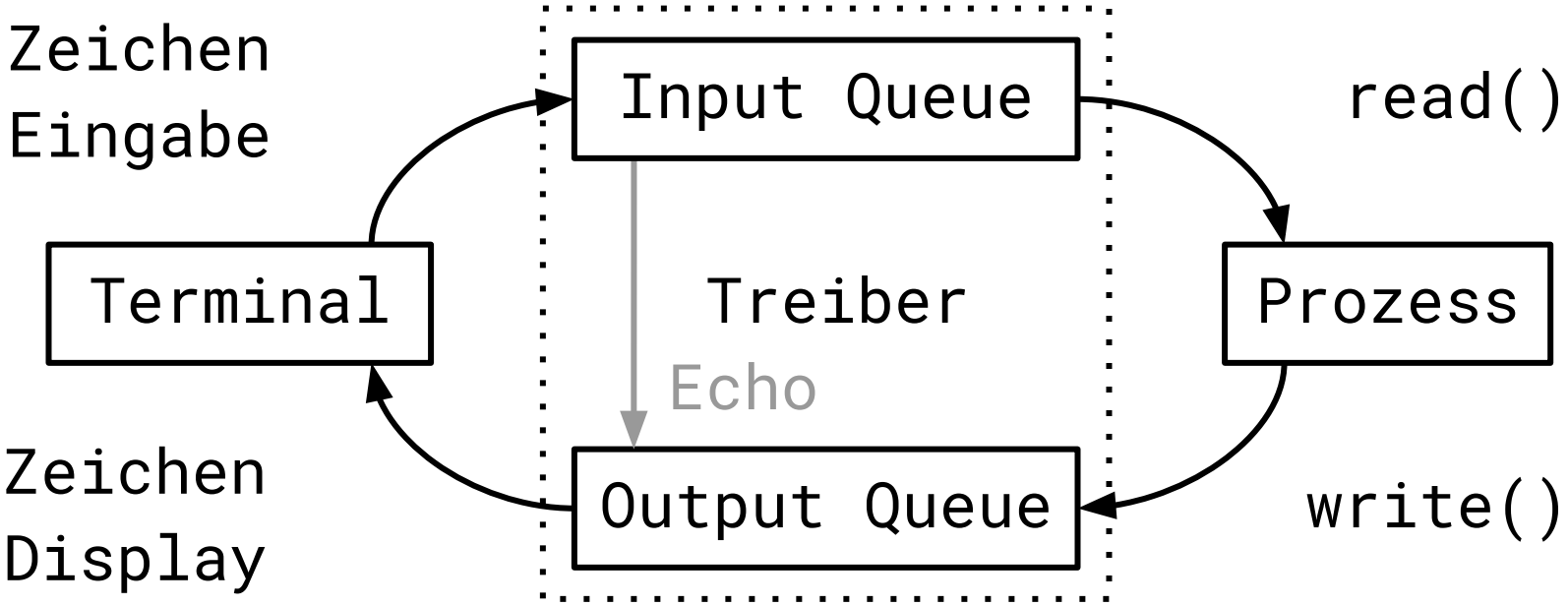
Terminal Treiber

Der Input Mode und die Interpretation von Steuerzeichen wie CTRL-C (*interrupt*) oder CTRL-D (*EOF*) werden im *Terminal Treiber* festgelegt.

Ein Terminal Treiber hat Queues für In- und Output.

Wenn die *Echo* Funktionalität eingeschaltet ist, wird jedes Zeichen Input auf die Output Queue kopiert.

Terminal Setup



Terminal Attribute lesen mit *tcgetattr()*

Terminal Attribute lesen mit dem *tcgetattr()* Call:

```
int tcgetattr(int tty_fd, struct termios *t);
```

Nach dem Aufruf stehen die aktuellen Attribute in *t*.

Im Fehlerfall ist der *return*-Wert *-1*, Fehler in *errno*.

Nach temporären Änderungen der Terminal Attribute sollte jeweils der ursprüngliche gelesene Wert wieder erstellt werden, siehe auch *tcsetattr()*.

Terminal Attribute in *struct termios*

Datenstruktur *struct termios* für Terminal Attribute:

```
struct termios {
  tcflag_t c_iflag; // Input Modes
  tcflag_t c_oflag; // Output Modes
  tcflag_t c_cflag; // Control Modes
  tcflag_t c_lflag; // Local Modes
  cc_t c_cc[NCCS]; // Special Characters
  … // Non-Standard Terminal Attribute
}
```

Terminal Attribute setzen mit *tcsetattr()*

Terminal Attribute setzen mit dem *tcsetattr()* Call:

```
int tcsetattr( // 0 oder 1, errno
  int tty_fd, // Terminal Device Deskriptor
  int optional_actions, // TSCANOW|DRAIN|FLUSH
  const struct termios *t); // von tcgetattr()
```

Der Parameter *optional_actions* bestimmt, wann die neuen Attribute angewendet werden. Der Parameter *t* sollte immer mit *tcgetattr()* initialisiert werden.

Verhalten von *tcsetattr()* bei Fehlern

Die *tcsetattr()* Funktion gibt `0` zurück, wenn eines der Attribute im *termios* Struct erfolgreich gesetzt wurde.

Ein Fehler bzw. -1 wird nur gemeldet, wenn keine der verlangten Änderungen durchgeführt werden konnte.

Es ist deshalb gut, Attribute nochmal mit *tcgetattr()* zu lesen, und die Werte mit dem Soll zu vergleichen.

Das *stty* Kommando

Das *stty* Kommando bietet dieselbe Funktionalität wie *tcgetattr()* und *tcsetattr()* auf der Command-Line:

```
$ stty -a # oder $ sudo stty -a -F /dev/ttyS0
speed 9600 baud; 24 rows; 80 columns; line = 0;
c_cc: intr = ^C; quit = ^\; erase = ^?; …
c_cflag: -parenb … c_iflag: -ignbrk …
c_oflag: opost … c_lflag: echoctl …
```

Ein '-' bedeutet, die Option ist nicht eingeschaltet.

Terminal Steuerzeichen

CR	Carriage Return	^M	ICANON, IGNCR, ICRNL, OPOST, OCRNL, ONOCR
DISCARD	Discard output	^O	(not implemented)
EOF	End-of-File		ICANON
EOL	End-of-Line		ICANON
EOL2	Alt. End-of-Line	^D	ICANON, IEXTEN
ERASE	Erase character	^?	ICANON
INTR	Interrupt (SIGINT)	^C	ISIG

KILL	Erase line	^U	ICANON
LNEXT	Literal next	^V	ICANON, IEXTEN
NL	Newline	^J	ICANON, INLCR, ECHONL, OPOST, ONLCR, ONLRET
QUIT	Quit (SIGQUIT)	^\	ISIG
REPRINT	Reprint input line	^R	ICANON, IEXTEN, ECHO
START	Start output	^Q	IXON, IXOFF
STOP	Stop output	^S	IXON, IXOFF
SUSP	Suspend (SIGTSTP)	^Z	ISIG
WERASE	Erase word	^W	ICANON, IEXTEN

Interrupt Character ändern new_intr.c TLPI

Beispiel, wie Steuerzeichen geändert werden kann:

```
struct termios t;
int intr_char;
…
tcgetattr(STDIN_FILENO, &t); // STDIN ist tty
tp.c_cc[VINTR] = intr_char; // V{CHAR_NAME}
tcsetattr(STDIN_FILENO, TCSAFLUSH, &t);
```

Danach Default wiederherstellen: `$ stty sane`

Terminal Flag ECHO aus no_echo.c TLPI

Beispiel, wie das *ECHO* Flag disabled werden kann:

```
struct termios tp, save;
tcgetattr(STDIN_FILENO, &tp);
save = tp; // Am Schluss wiederherstellen
tp.c_lflag &= ~ECHO; // Andere Bits ungeändert
tcsetattr(STDIN_FILENO, TCSAFLUSH, &tp);
… // Echo ist ausgeschaltet
tcsetattr(STDIN_FILENO, TCSANOW, &save);
```

Terminal I/O Modes

Canonical und *Noncanonical Mode* ermöglichen eine zeilen- und zeichenweise Verarbeitung von Input.

Cooked, *Cbreak*, und *Raw Mode* sind eine früher verwendete Aufteilung von Terminal I/O Modes, die mit den obigen Modes umgesetzt werden kann.

Canonical Mode

Canonical Mode wird mit dem *ICANON* Flag gesetzt und steht für zeilenweise Verarbeitung von Input:

Input ist editierbar, bis eine Zeile abgeschlossen wird.

Eine Zeile endet mit *NL*, *EOL*, *EOL2*, *EOF* oder *CR* (falls *ICRNL*), wobei dieses Zeichen (ausser *EOF*) an den Aufrufer von *read()* mit übergeben wird.

Noncanonical Mode

Noncanonical Mode, *~ICANON*, ist für zeichenweise Verarbeitung, wobei die Attribute *TIME** und *MIN*** das Verhalten von *read()* genauer festlegen:

```
MIN == 0, TIME == 0 // Polling read()
MIN > 0, TIME == 0 // Blocking read()
MIN == 0, TIME > 0 // read() mit Timeout
MIN > 0, TIME > 0 // read(), per Byte Timeout
```

* Timeout in 1/10 s, ** min. Anzahl gelesene Bytes.

Cooked, Cbreak und Raw Mode

Cooked Mode ist im wesentlichen Canonical Mode, wobei die default Steuerzeichen eingeschaltet sind.

Raw Mode ist das Gegenteil, Noncanonical Mode mit jeglicher Verarbeitung von In-/Output abgeschaltet.

Cbreak Mode ist dazwischen, Noncanonical Mode mit Verarbeitung von Signal-erzeugenden Input-Zeichen. In *curses* gibt es *cbreak()* und *raw()* Funktionen.

Terminal Line Speed (Bit Rate)

Verschiedene Terminals und Serial Devices übertragen Daten mit verschiedenen Speeds, in *bit/s* oder *baud*.

Diese Calls lesen bzw. setzen die In-/Output Bitrate:

```
speed_t cfgetispeed(const struct termios *t);
speed_t cfgetospeed(const struct termios *t);
int cfsetispeed(struct termios *t, speed_t s);
int cfsetospeed(struct termios *t, speed_t s);
```

Struct *t* wie vorher, Bit-/Baud-Raten siehe termios.

Terminal Line Control

`0` (BREAK) an *fd* für *duration* Millisekunden senden:

```
int tcsendbreak(int fd, int duration_ms);
```

Blockieren, bis Terminal Output Queue gesendet ist:

```
int tcdrain(int fd);
```

Rest-Inhalt der Input und Output Queue verwerfen:

```
int tcflush(int fd, int queue_selector);
```

Flow-Control regeln, Action `TCO`**`ON`**`|OFF`, `TCI`**`ON`**`|OFF`

```
int tcflow(int fd, int action);
```

Terminal Fenstergrösse

demo.c TLPI

Bei Änderungen der Terminal Fenstergrösse wird das SIGWINCH Signal an den Prozess gesendet.

Die aktuelle Fenstergrösse wird mit *ioctl()* abgefragt:

```
int res = ioctl(fd, TIOCGWINSZ, &ws);

struct winsize {
    unsigned short ws_row, ws_col; // Linux
    unsigned short ws_xpixel, ws_ypixel;
};
```

Terminal Identifizierung tty_id.c

Die Funktionen *isatty()* und *ttyname()* identifizieren File Deskriptoren als Terminals, oder geben `0`, *NULL*.

isatty() prüft, ob File Deskriptor *fd* ein Terminal ist:

```
int isatty(int fd); // 1, falls offener TTY FD
```

ttyname() liefert den TTY Namen des Deskriptors *fd*:

```
char *ttyname(int fd); // z.B. "/dev/pts/0"
```

Hands-on, 15': Kilo.c Revisited

Analysieren Sie den Source Code dieses Programms:
https://github.com/antirez/kilo/blob/master/kilo.c

Welche Terminal-spezifischen Calls werden im Code verwendet und wozu?

@antirez ist auch der Autor von Redis.

Pseudoterminals

Ein *Pseudoterminal* (*PTY*) besteht aus einem *master* Device und einem *subordinate* Device, bidirektional verbunden durch einen IPC Kanal.

Die Terminal "Emulation" geschieht im User-Space.

Dadurch kann ein Terminal-orientiertes Programm auch remote, über ein Netzwerk benutzt werden.

Master und subordinate Device

Historisch wurden im Zusammenhang mit Pseudo-terminals die Begriffe "master" und "slave" benutzt.

Wir verwenden stattdessen *master/subordinate*, als Adjektive, wie in diesem Style Guide erläutert.

Eine ähnliche Konvention gab es im Zusammenhang mit Datenbanken, dort sagt man neu master/replica, primary/secondary, oder leader/follower.

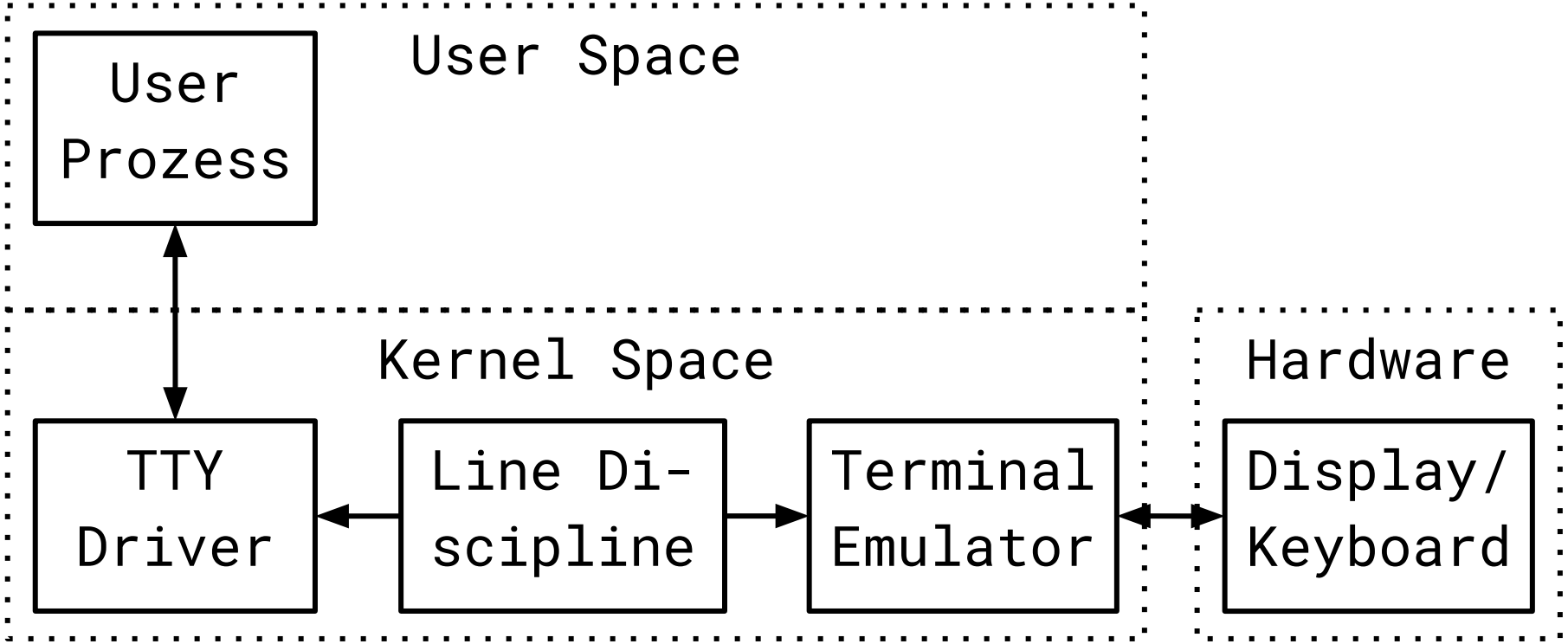
Terminal-orientierte Programme

Ein *Terminal-orientiertes* Programm erwartet vom Terminal Driver eine gewisse Input-Vorverarbeitung und Output-Nachbearbeitung (*Line Discipline*).

Und es braucht ein kontrollierendes Terminal, dessen File Deskriptor via */dev/tty* geöffnet werden kann.

Es geht also um Programme, die normalerweise in einer (lokalen) Terminal Session laufen würden.

Terminal Emulator Setup



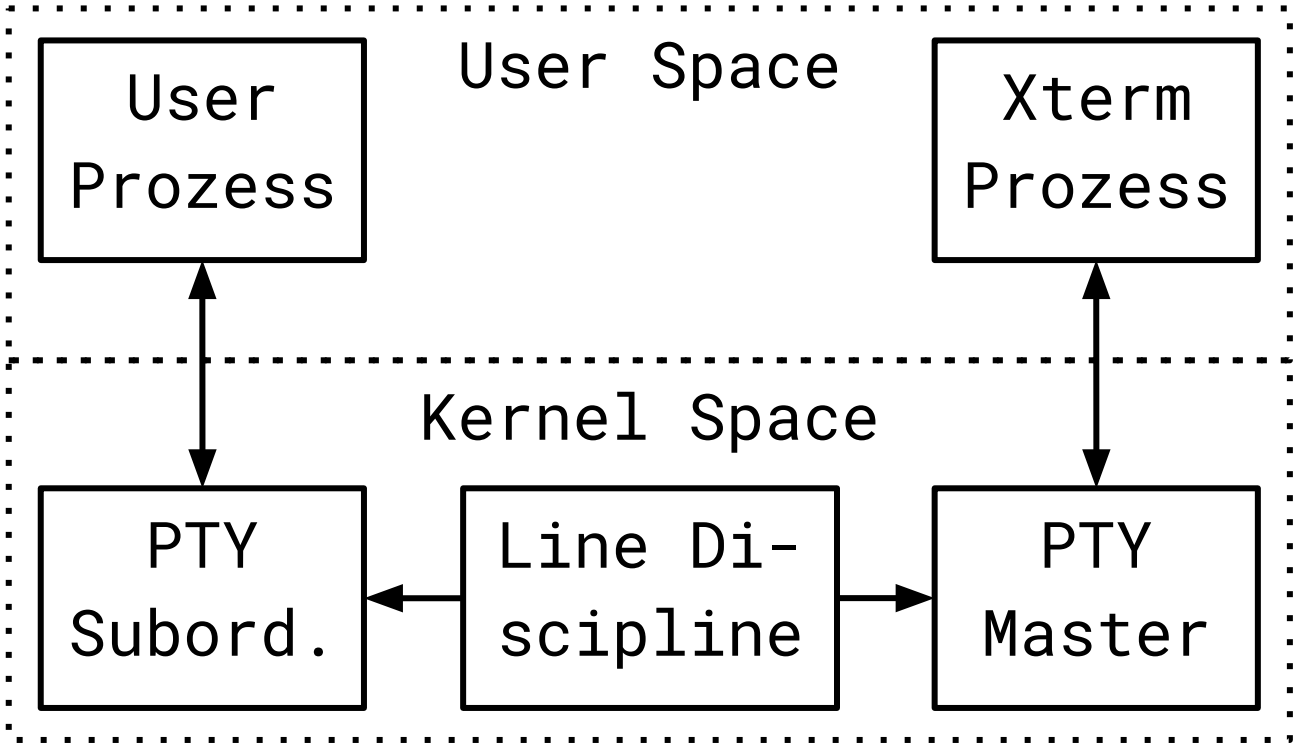
Auslagerung in User-Space

Ein Prozess, der ein Terminal erwartet, kann sich mit dem subordinate Device auf dieselbe Art verbinden, durch öffnen von TTY File Deskriptoren.

Der Prozess kann dann von einem 2. Prozess benutzt werden, der zum PTY master Device verbunden ist.

Beide Prozesse befinden sich im User-Space.

Pseudoterminal Setup mit Xterm



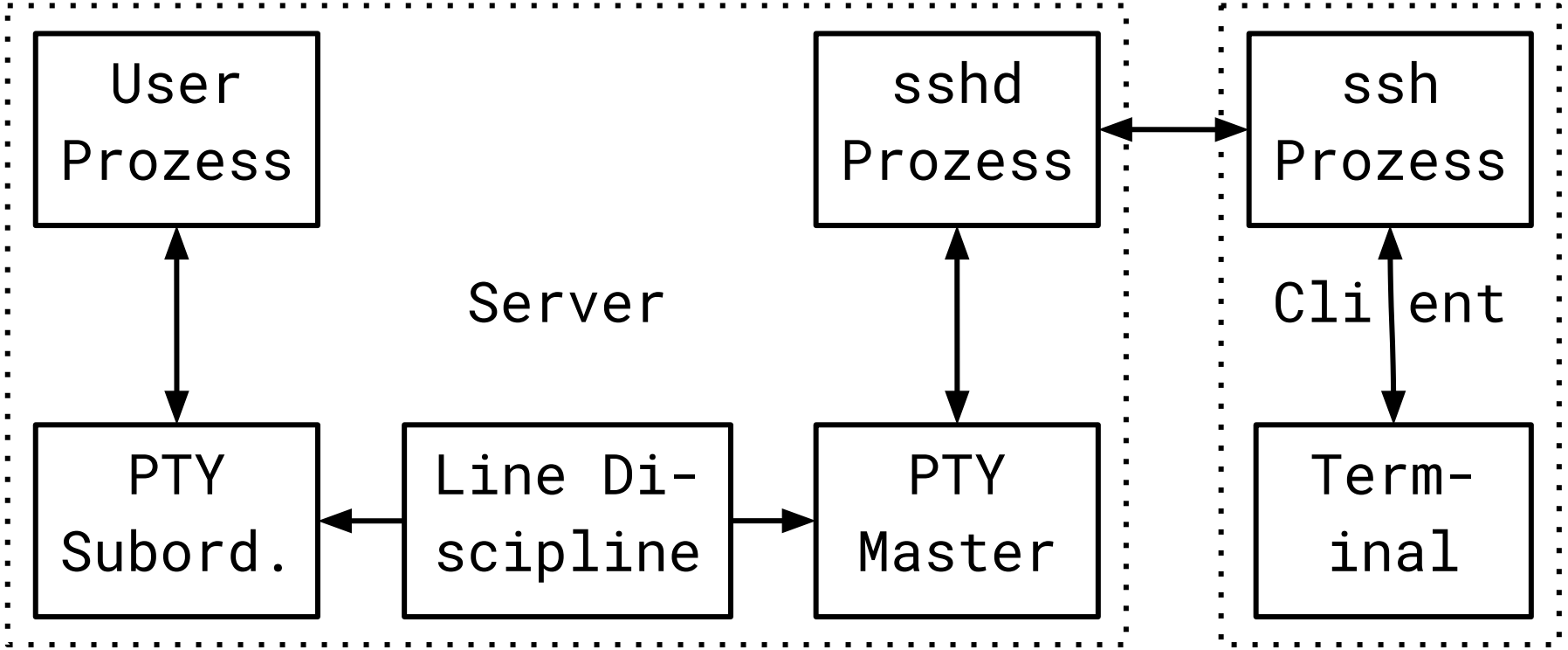
Zugriff über ein Netzwerk

Beim Zugriff über ein Netzwerk ist das Terminal nicht auf demselben Rechner, wie das Ziel-Programm.

Die Verbindung kann nur via Sockets geschehen, aber Terminal-orientierte Programme erwarten ein TTY.

Deshalb braucht es stellvertretend für das Terminal einen Proxy, das subordinate Device bzw. PTY.

Pseudoterminal Setup mit SSH



Feedback oder Fragen?

Gerne im Slack https://fhnw-syspr.slack.com/

Oder per Email an thomas.amberg@fhnw.ch

Danke für Ihre Zeit.