

System-Programmierung

4: Prozesse und Signale

CC BY-SA, Thomas Amberg, FHNW
(soweit nicht anders vermerkt)

Ablauf heute

1/2 Vorlesung,

1/2 Hands-on,

Feedback.

Slides, Code & Hands-on: tmb.gr/syspr-4



Prozesse und Programme

Prozesse sind Instanzen eines laufenden Programms.

Ein *Programm* ist ein File mit Informationen, wie ein Prozess zur Laufzeit konstruiert wird.

Ein Programm kann mehrere Prozesse kreieren, und mehrere Prozesse können dasselbe Programm laufen lassen bzw. instanzieren.

Programm Binary Aufbau

Binärformat	z.B. Executable & Linking Format (ELF).
Instruktionen	Das "Programm" in Maschinensprache.
Eintrittspunkt	Instruktion bei der das Programm startet.
Programmdaten	Initialwerte, Konstanten, String-Literale.
Relokations- und Symboltabellen	Position und Name von Funktionen und Variablen im Programm, für Debugging.
Shared Libraries	Liste der Libraries und Pfad des Linkers.
Weitere Angaben	Informationen, wie Prozess gebaut wird.

Prozess aus Kernel Sicht

Der Kernel sieht Prozesse als User-Space Speicher mit Programmcode und Initialwerten von Variablen.

Zudem unterhält der Kernel Datenstrukturen, um den Zustand von Prozessen zu managen, z.b. Prozess IDs, Virtual Memory, Offene File Deskriptoren, Signal-Handling, Ressourcenverbrauch und -Limiten, das aktuelle Arbeitsverzeichnis, und vieles mehr.

Prozess IDs

Jeder Prozess hat eine *Prozess ID* (PID), eine ganze, positive Zahl die den Prozess im System identifiziert.

Der *getpid()* System Call liefert die PID des Callers:

```
pid_t getpid(void); // geht immer ohne Error
```

Der *init* Prozess mit dem Linux startet, hat die PID 1.

Die maximale Anzahl PIDs im System ist *PID_MAX*:

```
$ cat /proc/sys/kernel/pid_max
```

Prozess Baumstruktur

pid.c

Jeder Prozess hat einen Vorgänger-/Parent-Prozess:

```
pid_t getppid(void); // immer ohne Error - oder  
$ cat /proc/PID/status | grep PPid # z.B. PID=1
```

Die Prozesse bilden einen Baum, mit *init* als Wurzel:

```
$ pstree // zeigt den aktuellen Prozess-Baum
```

Wenn ein Prozess verwaist, wird er von *init* adoptiert, d.h. die *getppid()* Funktion gibt ab dann 1 zurück.

Prozess Speicherlayout

`segments.c`

Der Speicher jedes Prozesses ist in *Segmente* geteilt:
Text bzw. Programmcode, initialisierte Daten, nicht initialisierte Daten (bss), Stack und Heap.

Das Beispiel *segments.c* zeigt, welche Programmteile in welchem Segment alloziert werden, zusammen mit:

```
$ size segments // Grösse von text, data & bss  
$ cat /proc/PID/maps // zeigt Segment Adressen
```


0xFFFFFFFF:

Kernel Symbole (gemapped)

0xC0000000:

x86, 32-bit

argv, environ

Stack (wächst nach unten) ^{f()}_{g()}

Stack Top→

Nicht allozierter Speicher

Prg.Break→

Heap (wächst nach oben)

Adresse im
virtuellen
Speicher,
hex Format

Nicht initialisierte Daten⁰

←&end

Initialisierte Daten

←&edata

Text (Programmcode)

read
only

←&etext

0x08048000:

Virtueller Speicher

Linux managt Speicher als *Virtual Memory*, so wird eine effiziente Nutzung von CPU und RAM erreicht.

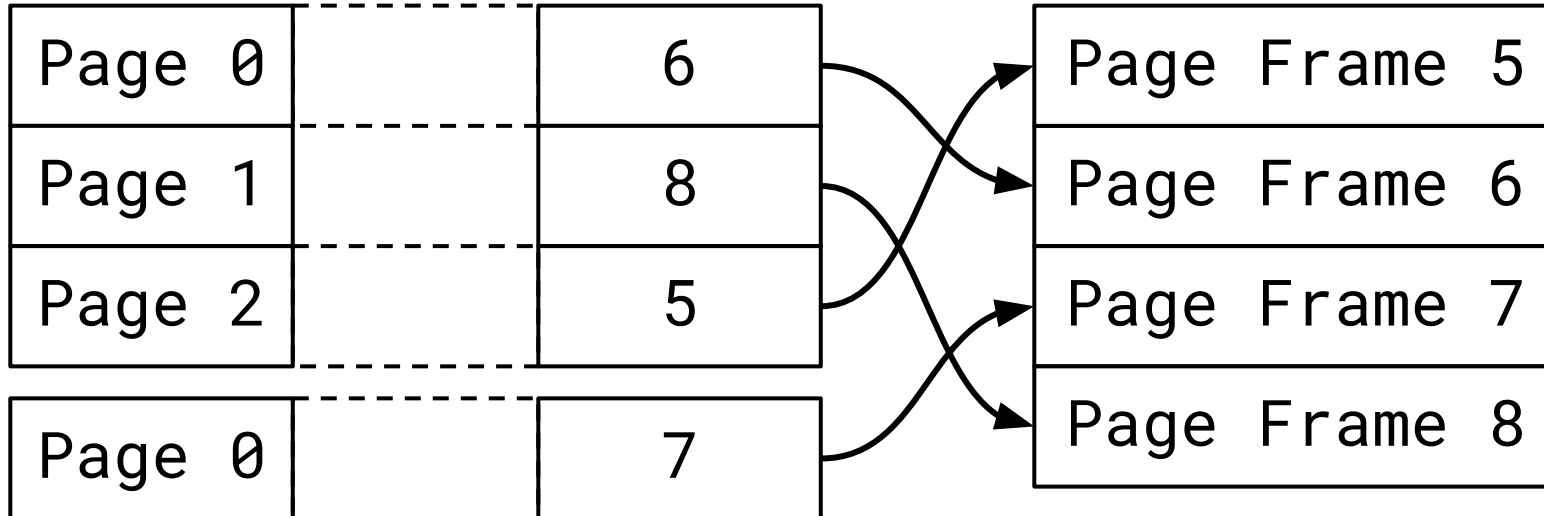
Das geht, weil aufeinanderfolgende Speicherzugriffe oft zeitlich und örtlich in der Nähe voneinander sind, z.B. in *while*-Schleifen, oder Array-Zugriffen: `a[i++]`

Dadurch muss nur ein Teil des Adressraums ins RAM geladen werden, eine *Page*, der Rest ist ausgelagert.

Prozess Page Table

Der Kernel führt für jeden Prozess eine *Page Table*:

Virtueller Adressraum Page Table / Prozess Physischer Speicher (RAM), Page Frames



Mapping auf physischen Speicher

Die Page Table beschreibt das Mapping von Pages im virtuellen Speicher auf physischen Speicher (RAM).

Unbenutzter virtueller *Adressraum* ist nicht gemappt.

Wenn ein Prozess auf ungemappten Speicher zugreift, gibt es einen *Segmentation Fault* (SIGSEGV).

Der gültige Adressraum kann sich ändern, wenn der Kernel zur Laufzeit Pages alloziert und dealloziert. 12

Vorteile von virtuellem Speicher

Prozesse sind voneinander und vom Kernel isoliert.

Programmtext kann (read-only) geshared werden.

Pages sind markierbar als read-/write-/executable.

Entwickler müssen physisches Layout nicht kennen.

Das Programm lädt/läuft schneller, und spart RAM.

Mehr Programme im RAM => CPU ist ausgelastet.

Stack und Stack Frames

Pro Funktionsaufruf wird ein *Stack Frame* alloziert, das nach dem *return* wieder vom *Stack* entfernt wird.

Der Stack wächst mit jedem (verschachtelten) Aufruf und schrumpft beim *return* wieder um einen Frame.

Ein Stack Frame enthält Funktionsargumente, lokale "automatische" Variablen, und CPU Register Kopien.

Command Line Argumente

args.c⁰¹

Die *main()* Funktion eines Programms wird von der Shell aufgerufen mit den Argumenten *argc* und *argv*:

```
int main(int argc, char* argv[]);
```

Der Name des Programms ist in *argv[0]* enthalten.

Die Elemente von *argv* sind Null-terminierte Strings.

Command Line eines Programms in Linux anzeigen:

```
$ cat -v /proc/PID/cmdline # mit ^@ für '\0'
```

Umgebungsvariablen

`environ.c`

Environment, enthalten in einer globalen Variablen:

```
extern char **environ; // Liste von Strings
```

Auf einzelne Umgebungsvariable zugreifen:

```
char *getenv(const char *name); // oder NULL  
int putenv(char *string); // != 0 => Error
```

Umgebungsvar. eines Programms in Linux anzeigen:

```
$ sudo cat /proc/PID/environ
```


Dynamische Speicherallokation

Manchmal brauchen Programme neuen Speicher für dynamische Datenstrukturen (Listen, Bäume, etc.) deren Grösse erst zur Laufzeit bekannt wird.

Dieser Speicher kommt meistens vom Heap, dessen Grösse über System Calls verändert werden kann.

Wir betrachten nun Funktionen, um auf dem Heap (und auf dem Stack) neuen Speicher zu allozieren.

Heap Speicher allozieren

heap.c

Heap Speicher allozieren mit *malloc()*:

```
void *malloc(size_t size); // Zeiger oder NULL
```

Der resultierende *void*-Pointer passt für alle Typen.

Heap Speicher freigeben mit *free()*:

```
void free(void *p); // genau einmal pro Zeiger
```

Freigegebener Speicher wird in eine Liste freier Blöcke eingefügt, später von *malloc()* wiederverwendet.

Fehlerfälle *bei malloc()* und *free()*

Die *malloc()* Funktion liefert im Fehlerfall NULL:

```
void *p = malloc(size); // NULL => ENOMEM  
if (p == NULL) { printf("error %d\n", errno); }
```

Falls *free()* schon einmal aufgerufen wurde:

```
free(p); // Verhalten undefiniert, z.B. SIGSEGV
```

Falls *free(NULL)* aufgerufen wird:

```
free(NULL); // Kein Effekt
```

Heap Grösse setzen

`free_and_sbrk`^{TLPI}

Der *Program Break* markiert die Grenze des Heaps, mit `sbrk(0)` kann man den Program Break auslesen, mit `brk()` an eine neue Adresse im Speicher setzen:

```
void *sbrk(intptr_t increment); // alte Adresse  
int brk(void *addr); // LEGACY, sbrk auch
```

Diese System Calls braucht es vor allem, um Library Funktionen `malloc()` und `free()` zu implementieren.

Hands-on, 30': Eigenes *malloc()* *malloc.c*

Implementieren Sie ein Programm *my_malloc.c* das Funktionen *my_malloc()* und *my_free()* anbietet.

Nutzen Sie dazu die System Calls *sbrk()* oder *brk()*.

Eine Skizze des *malloc()* Algorithmus' gibt es unter <https://stackoverflow.com/a/31026883/3588>

Vereinfachung: Gerüst von *my_malloc.c* im Repo.

Implementierung von *malloc(n)*

Scanne Liste freier Blöcke, bis $\text{Block} \geq n$, 1st-/best-fit.

Grösse = n ? *return* Block, sonst den zu grossen Block aufteilen, in einen *return*- und einen freien Block.

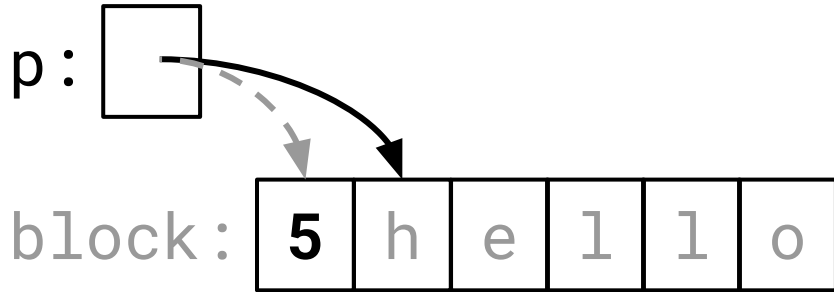
Kein freier Block? Heap mit *sbrk()* um $N \geq n$ Bytes vergrössern, mit N = ein Vielfaches der Page-Grösse, *return* Block, Rest als Block in die Liste freier Blöcke.

Implementierung von *free()*

size.c

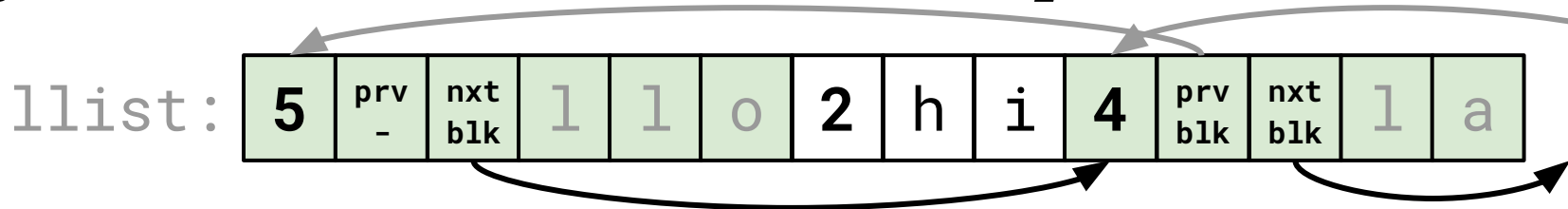
Bei *malloc()* hat's jeweils ein Byte für die Block Länge:

```
void *p = malloc(n); // => *((int *)p-1) ≈ n8+1
```



Achtung: free() nur auf Pointer die von malloc() kommen

free() nutzt den Block zudem für *prev*, *next* Pointers:



Stack Speicher allozieren

Mit *alloca()* alloziert man Speicher auf dem Stack:

```
void *alloca(size_t size); // sparsam nutzen
```

Da der Aufrufer zuoberst auf dem Stack ist, kann die Library in *alloca()* einfach den Stack Pointer erhöhen.

Falls Speicher voll: Segmentation Fault, nicht NULL.

Stack Speicher wird nach *return* frei, wenn der Stack Frame abgeräumt wird, es gibt hier kein *free()*.

Signale

Ein *Signal* sagt dem Prozess, dass etwas passiert ist.

Signale sind eine Art Software Interrupts, da sie wie echte Interrupts den Programmablauf unterbrechen. Man weiss oft nicht genau, wann ein Signal kommt.

Prozesse können Signale an andere Prozesse senden, und an sich selbst, die meisten Signale kommen aber vom Kernel (HW Exceptions, Input, SW Events).

Signal Ursachen

Im Kernel ist eine *Hardware Exception* aufgetreten, z.B. Division durch Null oder Segmentation Fault.

Bei *User Input*, z.B. *CTRL-C* oder *CTRL-Z* gedrückt.

Software Events, z.B. wenn ein Timer abgelaufen ist, oder Input verfügbar wird an einer File Schnittstelle, oder wenn ein Prozess ein Signal verschickt hat.

Signal Ablauf

Eine Quelle (Kernel, Prozess) *generiert* ein Signal.

Bis zur Auslieferung ist das Signal *pending* (hängig).

Sobald der Prozess an der Reihe ist, wird es *geliefert*.

Der Prozess kann nun (mit Core Dump) terminieren, stoppen, wieder weiterlaufen, das Signal ignorieren, oder das Signal behandeln, mit einem *Handler*.

Signal Nr. und Symbole

Jedes Signal hat eine Nr. und ein *SIGxxxx* Symbol:

```
#define SIGINT 2 // in signal.h
```

Standard Signale vom Kernel an den Prozess: 1 - 31.

Daneben gibt es in Linux *real-time* Signale: 32 - 64,
für Anwendungs-spezifische Use Cases: SIGRTMIN+n

Auf Signale sollte man immer per Symbol verweisen,
weil sie je nach System verschiedene Nr. haben.

Signale Maskieren

Darf ein Prozess nicht unterbrochen werden, setzt man eine *Maske* um einzelne Signale abzublocken.

Die blockierten Signale eines Prozesses anzeigen:

```
$ cat /proc/PID/status # SigBlk
```

Signale bleiben *pending* bis sie entblockt werden:

```
$ cat /proc/PID/status # SigPnd, ShdPnd
```

Es gibt keine Queue, Signal-Masken sind nur Sets.

Signal Handler installieren

Der *signal()* Call setzt für ein Signal *s* den Handler *h*:

```
void (*signal(int s, void (*h)(int))) (int);
```

Ein *Signal Handler* hat demnach die folgende Form:

```
void handle(int signal) { ... }
```

Der *return*-Wert ist die vorherige Handler-Funktion:

```
old_handle = signal(SIGINT, handle); // save  
// do something else, handle handles SIGINT  
... signal(SIGINT, old_handle); // restore
```

Signal Handler Konstanten

Bei einem Fehler ist der *return*-Wert *SIG_ERR*:

```
result = signal(SIGINT, handle);  
if (result == SIG_ERR) { ... }
```

Für default Signal Handler, *SIG_DFL* installieren:

```
result = signal(SIGINT, SIG_DFL);
```

Signal ignorieren, d.h. Handler *SIG_IGN* installieren:

```
result = signal(SIGINT, SIG_IGN);
```

Hands-on, 15': Sig. Handler `sigint.c`, `sigint.png`

Ihr nächstes Programm `my_sigint.c`, soll das Signal `SIGINT` mit einer Funktion `handle()` behandeln.

Schicken Sie ihrem Programm `SIGINT` mit `CTRL-C`.

Welche Rolle spielen User, Shell, Kernel, Prozess?

Zeichnen Sie ein Sequenzdiagramm des Aufrufs,
z.B. mit <https://www.websequencediagrams.com/>

Signal Handler Design

Signal Handler sollten so einfach wie möglich sein.

Oft setzt der Handler einfach nur ein globales Flag:

```
volatile int flag; // portabel: sig_atomic_t
```

Bei *Reentrance* (Wiedereintritt) wird Code mehrfach ausgeführt, Race Conditions können entstehen. Es ist z.B. unsafe, *stdio*-Funktionen wie *printf()* aufzurufen.

Bzw. man muss mit seltsamem Output rechnen.

Signale senden mit *kill()*

Ein Signal *sig* an den Prozess *pid* senden, mit *kill()*:

```
int kill(pid_t pid, int sig); // or -1, errno
```

Falls *pid* = 0 ist, geht das Signal an alle in derselben Prozess-Gruppe, wie der aufrufende Prozess.

Mit *pid* = -1 geht das Signal an alle Prozesse, an die der Aufrufer Signale senden darf, ausser an *init*.

Mit *sig* = 0 wird geprüft, ob Senden möglich ist.

Signale senden mit *raise()*

raise() sendet ein Signal *sig* an den eigenen Prozess:

```
int raise(int sig); // != 0 bei Error EINVAL
```

Das Aufrufen von *raise()* hat denselben Effekt wie:

```
kill(getpid(), sig);
```

Oder, in einem Programm mit mehreren Threads:

```
pthread_kill(pthread_self(), sig);
```

Der einzige Fehler ist *EINVAL*, falls *sig* ungültig.

Signal Beschreibung ausgeben

strsignal() liefert die Beschreibung des Signals *sig*:

```
char *strsignal(int sig); // oder NULL
```

Der String ist nur bis zum nächsten Aufruf gültig.

psignal() druckt eine Fehlermeldung *msg*, gefolgt von der Beschreibung von *sig* und *\n* auf *stderr*:

```
void psignal(int sig, const char *msg);
```

Sets von Signalen

Für Sets von Signalen gibt's den System Typ *sigset_t*.

Mit *sigemptyset()* oder *sigfillset()* wird ein Set kreiert:

```
int sigemptyset(sigset_t *set); // leeres Set  
int sigfillset(sigset_t *set); // alle im Set
```

sig{add|del}set() schliesst das Signal *sig* ein oder aus:

```
int sigaddset(const sigset_t *set, int sig);  
int sigdelset(const sigset_t *set, int sig);
```

Signale maskieren mit *sigmask()*

Signal Set *old* durch *new* ersetzen mit *sigprocmask()*:

```
int sigprocmask(int how, // e.g. SIG_SETMASK
                const sigset_t *new, // e.g. NULL = get old
                sigset_t *old); // ergibt 0 oder -1, errno
```

Der *how* Parameter kann folgende Werte annehmen:

```
SIG_BLOCK; // Signale in new werden hinzugefügt
SIG_UNBLOCK; // Signale in new werden entfernt
SIG_SETMASK; // Signale in new werden gesetzt
```

Warten auf Signale `pause.c`, `intquit.c`^{TLPI}

Die *pause()* Funktion suspendiert den Prozess, bis ein Signal eintrifft, danach wird -1 zurückgegeben:

```
int pause(void); // -1, errno = EINTR
```

Prüfen ob ein vom Handler gesetztes Flag aktiv ist:

```
pause(); // wartet auf das nächste Signal  
// installierter Handler wird aufgerufen  
// der Handler setzt ein globales Flag  
if (flag) { ... } // siehe auch pause.c
```

Selbststudium, 3h: Prozess Kreation

Als Vorbereitung auf die nächste Lektion, lesen Sie [\[TLPI\]](#) *Chapter 24: Process Creation*.

Das [PDF des Kapitels 24](#) ist verfügbar als Leseprobe.

Die nächste Lektion fasst den Lesestoff zusammen, ohne Selbststudium wird das Tempo eher hoch sein.

PS. Eine gute Übersicht zu Signals finden Sie [hier](#).

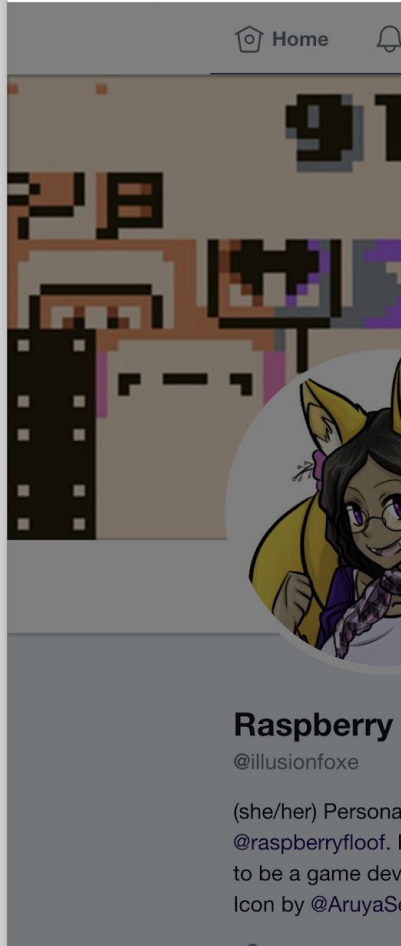
Feedback oder Fragen?

Gerne auf <https://fhnw-syspr-fs20.slack.com/>

Oder per Email an thomas.amberg@fhnw.ch

Slides, Code & Hands-on: tmb.gr/syspr-4





Raspberry

@illusionfoxe

Follow

I love that the default Emacs scratch buffer text got into Mario Golf (GBC) somehow. I wonder if this was a mis-copy-and-paste from a developer, or if it was another case of a compiler taking random memory contents and using them for padding in the ROM...

```
[D:\Roms\Game Boy Color\Mario Golf (USA).gbc] - Translhexion
File Selection Offset Insert Search Bookmarks Script Options View Help
000C8780 3B 48 56 48 AE 48 00 00 00 00 00 00 00 00 00 00 00 FF 00 :HVH0H.....y.
000C8790 00 00 09 00 09 01 80 00 01 00 01 02 C0 00 1E 00 :.....A.
000C87A0 01 03 C0 00 1E 00 31 FF 01 FF 00 00 00 00 00 03 01 :.A..ly.y.
000C87B0 FF DF 02 44 C9 21 72 01 DF 46 05 C9 02 FF 00 00 :yB.DElr.BF.E.y.
000C87C0 6B 01 03 00 03 FF 00 00 6C 01 03 00 04 FF 00 00 :k...y..l...y.
000C87D0 6D 01 03 00 05 FF 00 00 6E 01 03 00 06 FF 00 00 :m...y..n...y.
000C87E0 6F 01 03 00 07 FF 00 00 70 01 00 00 08 FF 00 00 :o...y..p...y.
000C87F0 B1 47 01 00 FF 02 10 00 00 6B 01 01 00 03 80 00 :tG.y...k.
000C8800 00 6C 01 01 00 04 20 00 00 6D 01 01 00 05 40 00 :l...m...@.
000C8810 00 6E 01 01 00 06 FF 65 80 6F 01 01 00 07 30 65 :n...ye o...0e
000C8820 00 70 01 01 00 FF DF 00 0A 21 72 01 DF 0E 0A 3E :p...yB..lr.B.>
000C8830 00 DF 08 0A DF 02 0A EF 40 02 C9 01 FF 00 00 67 :.B..B..i@.E.y..g
000C8840 7E 06 00 FF DF 00 0A 21 72 01 DF 0E 0A 3E 00 DF :~.yB..lr.B.>.B
000C8850 08 0A DF 02 0A C9 01 FF 00 00 44 48 00 00 FF 54 :.B.E.y.DH..yT
000C8860 68 69 73 20 62 75 66 65 72 20 69 73 20 66 6F :has buffer is fo
000C8870 72 20 6E 6F 74 65 73 79 6F 75 20 64 6F 6E 27 74 :r notesyou don't
000C8880 20 77 61 6E 74 20 74 6F 20 73 61 76 65 2C 61 6E :want to save an
000C8890 64 20 66 6F 72 20 4C 69 73 70 20 65 76 61 6C 75 :d for Lisp evalu
000C88A0 61 74 69 6F 6E 2E 41 42 43 44 45 46 47 00 FA 6A :ation ABCDEFGH ij
000C88B0 C4 FE 04 20 34 0E 04 CD AF 25 CD 25 26 21 1C 01 :Ap. 4..I%X!&l.
000C88C0 3E 03 DF 50 05 FE FF 28 EC EA 90 C8 F6 10 EA D0 :> AP by(i@.E@.e@
000C88D0 C8 3E 01 EA 6C C9 3E 00 DF 18 0B 3E 00 01 00 01 :E>.eIE>.B..>...
000C88E0 11 00 01 DF 20 0A 18 CD C9 FE 05 20 22 3E 03 EA :>.B..IEp>..>.e
000C88F0 36 C8 3E 0A EA 90 C8 3E 08 EA D0 C8 3E 03 EA 6C :6E>.eE>.eE>.eI
000C8900 C9 3E 02 DF 18 0B 0E 04 CD AF 25 CD 25 26 C9 FE :E>.B..I%X!&Ep
000C8910 06 20 1D 3E 00 EA 90 C8 3E F6 EA BE C8 3E 04 EA :>.eE>.eE>.e
000C8920 6C C9 3E 01 DF 18 0B 0E 04 CD AF 25 CD 25 26 C9 :IE>.B..I%X!&E
000C8930 FE 07 20 22 3E 06 EA 90 C8 3E 02 EA D0 C8 3E 03 :b..>.eE>.eE>..
```

