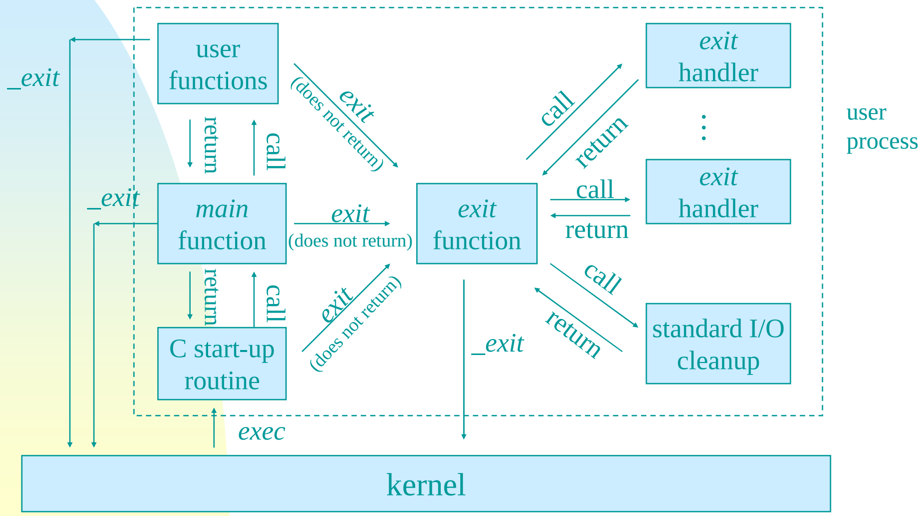
Contents

The Environment of a Unix Process

- Objective:
  - How a process is executed and terminates?
  - What the typical memory layout is?
  - Related functions and resource limits.
- int main(int argc, char *argv[])
  - A special start-up routine is called to set things up first before call main()
    - Set up by the link editor (invoked by the compiler)

How a C program is started and terminated.



Process Termination

- Five ways to terminate:
  - Normal termination
    - Return from main().
      - exit(main(argc, argv));
    - Call exit().
    - Call _exit()
  - Abnormal termination
    - Call abort()
    - Be terminated by a signal.

Process Termination

<stdlib.h>

```
#include <stdlib.h>
void exit(int status);
```

- ANSI C
- Perform cleanup processing
  - Close and flush I/O streams (fclose)

```
#include <unistd.h>
void _exit(int status);
```

- POSIX.1
- Undefined exit status:
  - Exit/return without status.
  - Main falls off the end.

```
#include <stdio.h>
main() {
   printf("hello world\n");
}
```


Process Termination

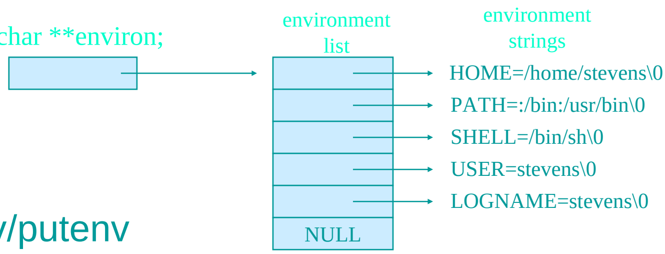
```
#include <stdlib.h>
int atexit(void (*func)(void));
```

- Up to 32 functions called by *exit* – ANSI C, supported by SVR4&4.3+BSD
- The same exit functions can be registered for several times.
- Exit functions will be called in reverse order of their registration.
- Program 7.1 – Page 165
  - Exit handlers

Command-Line Arguments & Environment Variables

- Command-Line Arguments
  - argv[argc] is NULL under POSIX.1 & ANSI C
  - Program 7.2 – Page 166
- Environment Variables
  - int main(int agrc, char **argv, char **envp);

extern char **environ;



- getenv/putenv

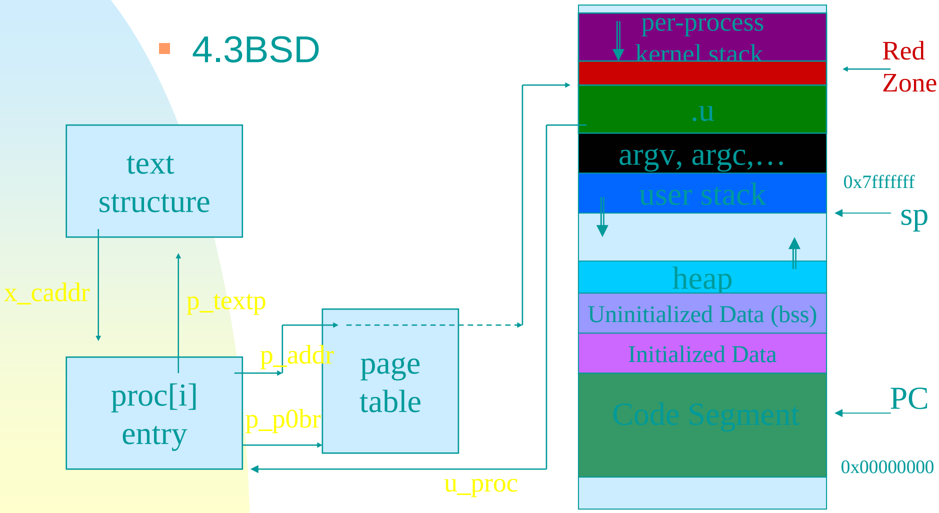
Memory Layout

- Pieces of a process
  - Text Segment
    - Read-only usually, sharable
  - Initialized Data Segment
    - int maxcount = 10;
  - Uninitialized Data Segment – bss (Block Started by Symbol)
    - Initialized to 0 by *exec*
    - long sum[1000];
  - Stack – return addr, automatic var, etc.
  - Heap – dynamic memory allocation

Read from program file by *exec*

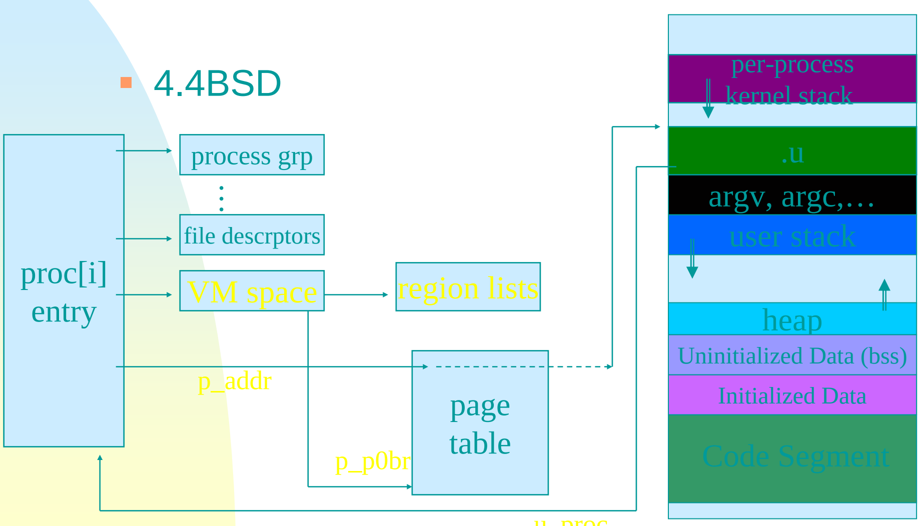
Memory Layout

- 4.3BSD



Memory Layout

- 4.4BSD



Memory Layout

- > size ls1 hole
  ls1: 6971 + 876 + 364 = 8211
  hole: 6995 + 896 + 368 = 8259

↑	↑	↑
text	data	bss

Shared Library

- Why a shared library?
  - Remove common library routines from executable files.
  - Have an easy way for upgrading
- Problems
  - More overheads
- Remark:
  - compiling options – gcc
  - Supported by many Unix systems

Memory Allocation

- Three ANSI C Functions:
  - malloc – allocate a specified number of bytes of memory. Initial values are indeterminate.
  - calloc – allocate a specified number of objects of a specified size. The space is initialized to all 0 bits.
  - realloc – change the size of a previously allocated area. The initial value of increased space is indeterminate.

Memory Allocation

```
#include <stdlib.h>
void *malloc(size_t size);
void *calloc(size_t nobj, size_t size);
void *realloc(void *ptr, size_t newsize);
```

- Suitable alignments for any data obj
- Generic void * pointer
- free(void *ptr) to release space to a pool.
- Leaking problem
- Free already-freed blocks or blocks not from alloc().
  - mallopt(M_GRAINSet, value), mallinfo

```
#include <stdio.h>
#include <stdlib.h>
main() {
char *ptr;

        ptr = malloc(100);
        free(ptr);
        free(ptr);
        ptr[0] = 'a'; ptr[1]=0;
        printf("%s - Done\n", ptr);
}
```

Memory Allocation

- Remark
  - realloc() could trigger moving of data → avoid pointers to that area!
    - *prt* == NULL → malloc()
  - sbrk() is used to expand or contract the heap of a process – a malloc pool
  - Record-keeping info is also reserved for memory allocation – do not move data inside.
  - alloca() allocates space from the stack frame of the current function!
    - No needs for free with potential portability problems

Environment Variables

- Name=value
  - Interpretation is up to applications.
    - Setup automatically or manually
    - E.g., HOME, USER, MAILPATH, etc.

setenv FONTPATH $X11R6HOME/lib/X11/fonts\:$OPENWINHOME/lib/fonts

```
#include <stdlib.h>
char *getenv(const char *name);
```

  - Figure 7.4 – environment variables
  - ANSI C function, but no ANSI C environment variable.

Environment Variables

#include <stdlib.h>

int putenv(const char *name-value);

- Remove old definitions if they exist.

int setenv(const char *name, const char *value, int rewrite);

- rewrite = 0 → no removing of existing names.

int unsetenv(const char *name);

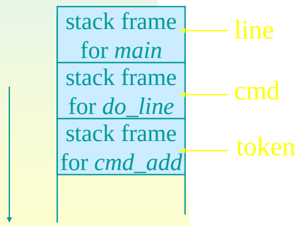
- Remove any def of the *name*
- No error msg if no such def exists.

Environment Variables

- Adding/Deleting/Modifying of Existing Strings
  - Modifying
    - The size of a new value <= the size of the existing value → overwriting
    - Otherwise; malloc() & redirect the ptr
  - Adding
    - The first time we add a new name → malloc of pointer-list's room & update *envron*
    - Otherwise; copying. realloc() if needed.
  - The heap segment could be involved.

setjmp and longjmp

- Objective:
  - goto to escape from a deeply nested function call!
  - Program 7.3 – Program Skeleton
    - What if cmd_add() suffers a fatal error? How to return to main() to get the next line?



Note: Automatic variables are allocated within the stack frames!

setjmp and longjmp

#include <setjmp.h>

int setjmp(jmp_buf env);

int longjmp(jmp_buf env, int val);

- Return 0 if called directly; otherwise, it could return a value *val* from longjmp().
- *env* tends to be a global variable.
- longjmp() unwinds the stack and affect some variables.

- Program 7.4 – Page 178
  - setjmp and longjmp

setjmp and longjmp

- Automatic, Register, and Volatile Variables
  - Compiler optimization
    - Register variables could be in memory.
  - Values are often indeterminate
    - Normally no roll back on automatic and register variables
      - Shown in Program 7.5 later
    - Global and static variables are left alone when longjmp is executed.
  - Portability Issues!

setjmp and longjmp

- Program 7.5 – Page 179
  - Effects of longjmp
  - Variables stored in memory have their values unchanged – no optimization…
- Potential Problems with Automatic Variables – Program 7.6 (Page 180)
  - Never be referenced after their stack frames are released.

getrlimit and setrlimit

```
struct rlimit {
 rlim_t rlim_cur; /* soft limit */
 rlim_t rlim_max; /* hard limit *
 }
```

#include <sys/time.h>

#include <sys/resource.h>

int getrlimitint resource, struct rlimit *rlptr);

int setrlimitint resource, const struct rlimit *rlptr);

- Not POSIX.1, but supported by SVR4 and 4.3+BSD
- Rules:
  - Soft limit <= hard limit, the lowering of hard limits is irreversible.
  - Only a superuser can raise a hard limit.

getrlimit and setrlimit

- Resources (SVR4&4.3BSD)
  - RLIMIT_CORE (both), RLIMIT_CPU (both, SIGXCPU), RLIMIT_DATA (both), RLIMIT_FSIZE (both , SIGXFSZ), RLIMIT_MEMLOCK (4.3+BSD, no implementation), RLIMIT_NOFILE (SVR4, _SC_OPEN_MAX), RLIMIT_NPROC (4.3+BSD, _SC_CHILD_MAX), RLIMIT_OFILE (4.3+BSD=RLIMIT_NOFILE), RLIMIT_RSS (4.3+BSD, max memory resident size), RLIMIT_STACK (both), RLIMIT_VMEM (SVR4)

* All rights reserved, Tei-Wei Kuo, National Taiwan University, 2003.

getrlimit and setrlimit

- Resource Limits → inheritance by processes
  - Built-in commands in shells
    - umask, chdir, limit (C shall), ulimit –H and –S (Bourne shell and KornShell)
- Program 7.7 – Page 183
  - Resource limits
    - #define RLIM_NLIMITS 7
  - doit(RLIMIT_CORE) = pr_limits(“RLIMIT_CORE”, RLIMIT_CORE)
    - #define doit(name) pr_limits(#name, name)