Programmation Systèmes
Cours 2 — Introduction to Process Management

Stefano Zacchiroli
zack@pps.jussieu.fr

Laboratoire PPS, Université Paris Diderot - Paris 7

6 Octobre 2011

Outline

Programs

Definition (programs and processes — 2nd approximation)

- A program is an executable file residing on the filesystem.
- A process is an abstract entity known by the kernel, to which system resources are allocated in order to execute a program.

A program contains all information needed to create a process at runtime:

- *binary format* (nowadays: ELF; once upon a time: a.out, COFF)
- machine *instructions*
- *entry-point*: address of the first instruction
- data
- symbol-/relocation- tables (for debugging, dynamic linking, etc.)
- shared library information

Processes — as viewed by the kernel

- A *process* is an abstract entity known by the kernel, to which system resources are allocated in order to execute a program.

From the point of view of the kernel, a process consists of:

- a portion of user-space memory
  - program code
  - variables accessed by the code
- kernel data structures to maintain state about the process, e.g.:
  - table of open *file descriptors*
  - *virtual memory* table
  - *signal* accounting and masks
  - process *limits*
  - current *working directory*
  - ...

Process IDs

How can the kernel index process-related data structures?

Definition (process ID)

Each process has a process ID (PID): a *positive integer* that uniquely identify processes on the system.

typical usages:

- *internal reference* by the kernel (e.g. indexing process-related data structures)
- *external reference* by other processes or the admin (e.g. `kill`)
- embedding in derived (unique) names, e.g. process-unique filenames

Process IDs — demo

1. (a view on) internal process reference: `/proc`
2. external reference: `ps(1)`, `kill(1)`
3. unique filenames, e.g.

```
$ ls /tmp | grep aptitude
aptitude-zack.20871:pUkqOd
$
```

Demo

Process ID reuse

Although unique, process IDs are reused. (why?)

- as soon as a process terminate, its process ID become *candidate* for reuse
- UNIX kernels implement algorithms to *delay reuse*
  - this prevents addressing by mistake new processes who took the place of recently terminated processes
  - the simplest effective algorithm is to allocate process IDs sequentially, wrapping around

Don't assume PIDs are stable forever.
With caution, you can assume they are stable "for a while".

Process ID reuse (cont.)

How long it is “for a while”? It depends on:

1. process creation ratio
2. PID max value

```
#include <stdio.h>
#include <sys/types.h>

int main()
{
    printf("pid_t:\t%ld\n", sizeof(pid_t)); // process IDs type
    printf("int:\t%ld\n", sizeof(int));
    printf("long:\t%ld\n", sizeof(long));
}
```

```
$ ./pid-size    # on a Linux, x86-64 bit system
pid_t:  4
int:    4
long:   8
$
```

getpid

Each process can retrieve *its own* PID at runtime using the syscall:

```
#include <unistd.h>
pid_t getpid(void);
```

Returns: *always return PID of calling process*

Accessing PID values:

- `pid_t` is an abstract type
- according to POSIX, process IDs shall be *signed integer types*
  - but they wrap to 0, according to PID definition
- we can use `pid_t` values as signed integers

getpid — demo

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char **argv) {
        printf("hello, world from process %d\n", getpid());
        exit(EXIT_SUCCESS);
}
```

```
$ gcc -Wall -o hello-pid hello-pid.c
$ ./hello-pid
hello, world from process 21195
$ ./hello-pid
hello, world from process 21196
$ ./hello-pid
hello, world from process 21199
```

Note: we print PIDs using %d conversion specifier

main

A C program starts with the execution of its `main` function:

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

- argc number of command line arguments
- argv array of pointers to arguments

It is the kernel who initiative program execution.[1]

Before main execution, a startup routine—inserted by the dynamic loader, or link editor, and specified in the binary program—is executed. The startup routine fills in:

- argc/argv (copying from exec arguments in kernel space)
- environment

[1] usually in response to an exec syscall

argv (by the standards)

not only:

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
        int i;
        for (i=0; i<argc; i++)
                printf("argv[%d] = %s\n", i, argv[i]);
        exit(EXIT_SUCCESS);
}
```

but also:

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
        int i;
        for (i=0; argv[i] != NULL; i++)
                // POSIX.1 and ISO guarantee argv[argc] == NULL
                printf("argv[%d] = %s\n", i, argv[i]);
        exit(EXIT_SUCCESS);
}
```

Process termination

There are many ways for a program to terminate.

Normal termination

1. return from `main` ("falls off the end")
2. `exit`
3. `_exit` or `_Exit`
4. as (1) and (2), but for thread-related purposes

Abnormal termination

5. `abort` (signal-related)
6. receipt of a signal
7. fulfillment of a thread-cancellation request

Falling off the end implicitly invokes `exit`.
Intuition: it is as if the startup routine calls main as

```
exit(main(argc, argv));
```

Normal termination — clean shutdown

```
#include <stdlib.h>

void exit(int status);
```

Returns: *does not return*

Clean shutdown performs cleans up *standard library resources* before terminating the process:

- invoke `fclose` on all open streams
- invoke exit handlers

Normal termination — abrupt shutdown

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

Returns: *does not return*

Exit status

All exit-like functions expect an integer argument: the exit status.[2] The exit status provides a way to communicate to other processes *why* the process has (voluntarily) terminated.

Example

The UNIX convention is that programs terminating with a 0 exit status have terminated successfully; programs terminating with a $!= 0$ exit status have failed.
The convention is heavily used by shells.

To avoid magic numbers in your code:

```
#include <stdlib.h>

exit(EXIT_SUCCESS);
// or exit(EXIT_FAILURE);
```

[2] *exit* status ≠ *termination* status. The latter accounts for both normal and abnormal termination; the former only for normal termination

Exit status (cont.)

You shall always declare `main` of type `int` *and* return an integer value; barring standards uncertainty:

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

```
$ gcc -o fall-off fall-off.c
$ ./fall-off
hello, world!
$ echo $?
14
```

```
$ gcc -o fall-off -std=c99 fall-off.c
$ ./fall-off
hello, world!
$ echo $?
0
```

Exit handlers

A process can register handlers that will be executed upon *clean shutdown*:

```
#include <stdlib.h>

int atexit(void (*func)(void));
```

Returns: *0 if OK, nonzero on error*

Notes:

- handlers will be invoked last-registered-first
- ISO C guarantees that the system supports at least a maximum of 32 handlers

Exit handlers — example

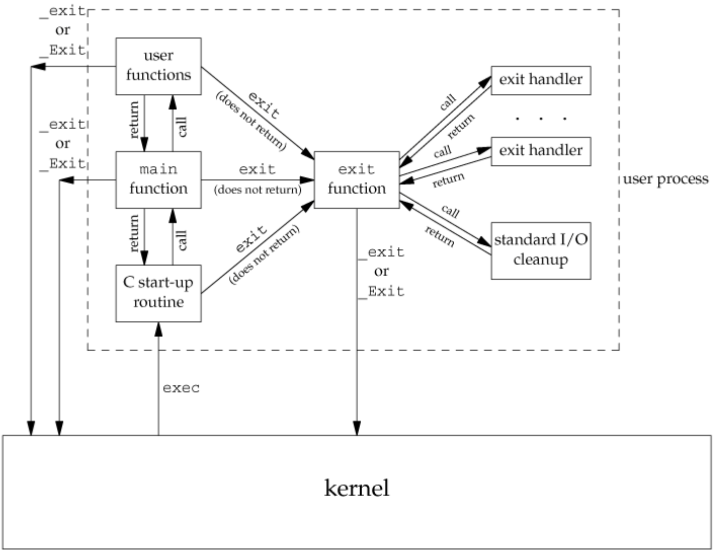
```
#include <stdio.h>
#include <stdlib.h>
#include "apue.h"

void my_exit1(void) { printf("first exit handler\n"); }
void my_exit2(void) { printf("second exit handler\n"); }

int main(void) {
        if (atexit(my_exit2) != 0)
                err_sys("can't register my_exit2");
        if (atexit(my_exit1) != 0)
                err_sys("can't register my_exit1");
        if (atexit(my_exit1) != 0)
                err_sys("can't register my_exit1");
        printf("main is done\n");
        return(0);
}
```

APUE, Figure 7.3

Exit handlers — example

```
$ ./atexit
main is done
first exit handler
first exit handler
second exit handler
$
```

Startup and termination — putting it all together

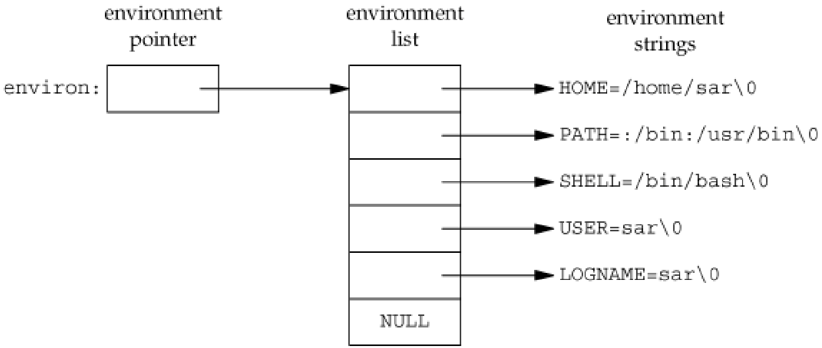


APUE, Figure 7.2

Environment list

Each process is passed, upon startup an environment list, i.e. a list of ⟨*key*, *value*⟩ pairs called environment variables.
The environment list can be accessed via the global variable:

```
extern char **environ;
```



APUE, Figure 7.5

getenv & putenv

Environment variables can also be accessed via specific functions from the standard library:

```
#include <stdlib.h>

char *getenv(const char *name);
                    Returns: pointer to value if name is found, NULL otherwise

int putenv(char *name);
                                        Returns: 0 if OK, nonzero on error
```

- getenv performs key-based lookup
- putenv adds a key/value pair given in "key=value" format, possibly overwriting previous values

The complete getenv family

```
#include <stdlib.h>

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

int unsetenv(const char *name);
```

Returns: *0 if OK, -1 on error*

- `setenv` is similar to `putenv`, but allows to tune its overwrite behavior
- `unsetenv` removes existing environment variables
  - relevant use case: cleaning up an environment before spawning a new process
- only `getenv` is ISO C and widely supported; support for the other functions varies

Note: `getenv` & friends are not expressive enough to browse the entire environment list; the only way to do that is via `environ`

A typical environment list

```c
#include <stdio.h>
#include <stdlib.h>

extern char **environ;

int main() {
        int i;
        for (i=0; environ[i] != NULL; i++)
                printf("%s\n", environ[i]);
        exit(EXIT_SUCCESS);
}
```

Demo

Standard environment variables

UNIX kernels ignore environment variables. Interpretation of the meaning of environment variables is left to applications.

POSIX.1 and SUS define some standard environment variables and their meaning. Some of them are:

- COLUMNS
- HOME
- LANG
- LC_ALL
- LC_COLLATE
- LC_CTYPE
- LC_MESSAGES
- LC_MONETARY
- LC_NUMERIC
- LC_TIME
- LINES
- LOGNAME
- PATH
- PWD
- SHELL
- TERM
- TMPDIR
- TZ

See APUE 7.7 and `environ(7)`.

Outline

Process address space — redux

Each process executes by default in its own address space and cannot access the address spaces of other processes — barring a segmentation fault error.

The memory corresponding to a process address space is allocated to the process by the kernel upon process creation. It can be extended during execution.

The address space of a program *in execution* is partitioned into parts called segments.

More on segments

text segment machine instructions that the CPU executes. It is read from disk upon process creation

initialized data segment ("data segment") global variables explicitly initialized, e.g.:

```c
int magic = 42;  // outside any function
```

uninitialized data segment ("bss segment") global variables not explicitly initialized, e.g.:

```c
char crap[1024];  // outside any function
```

- doesn't take any space in the on-disk binary
- it will be initialized by the kernel at 0 / NULL
- it can be initialized efficiently using *copy-on-write*

More on segments (cont.)

stack dynamically growing and shrinking segment made of stack frames. One stack frame is allocated for each currently called function. Each frame contains automatic variables, i.e. function's local variables, arguments, and return value.

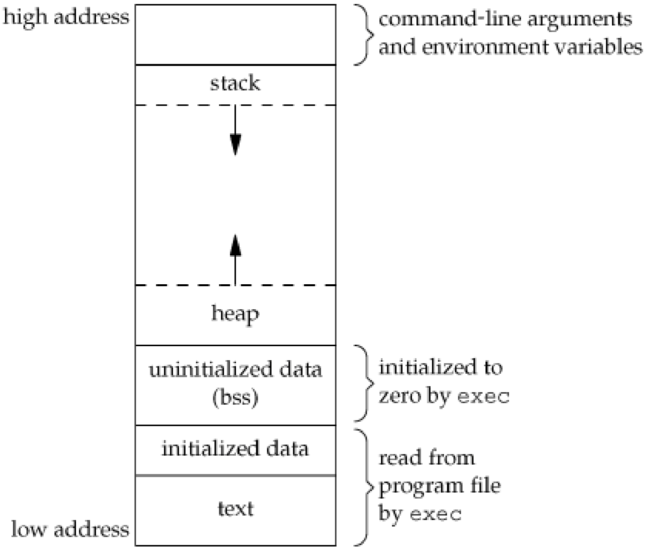
heap dynamically growing and shrinking segment, for dynamic memory allocation. The top of the heap is called program break

`size(1)` displays segment sizes for an on-disk binary

Static memory

The improper expression "static memory" refers to memory allocated in the data or bss segments. Such memory is static *wrt program execution* (which is not the case for stack and heap.)

Typical segment arrangement



APUE, Figure 7.6

Segment arrangement — demo

```
#include <stdio.h>
#include <stdlib.h>

int magic = 42;
char crap[1024];

void func(int arg) {
        printf("stack segment near\t%p\n", &arg);
}
int main(int argc, char **argv) {
        char *ptr;
        ptr = malloc(1);
        func(42);
        printf("heap segment near\t%p\n", ptr);
        printf("bss segment near\t%p\n", crap);
        printf("text segment near\t%p\n", &magic);

        free(ptr);
        exit(EXIT_SUCCESS);
}
```

Segment arrangement — demo (cont.)

```
$ ./segments
stack segment near 0x7ffff53ecccc
heap segment near  0x     1c52010
bss segment near   0x      600b00
text segment near  0x      600ad0
$
```

(output edited for alignment)

Virtual memory

Segments are conceptual entities not necessarily corresponding to physical memory layout. In particular, segments are about the layout of *virtual* memory.

Virtual Memory Management (VMM) is a technique to make efficient of physical memory, by exploiting locality of reference that most programs show:

- *spatial locality*: tendency to reference memory addresses near recently addressed addresses
- *temporal locality*: tendency to reference in the near feature memory addresses that have been addressed in the recent past

Virtual Memory Management in a nutshell

We partition:

- address space of each process in fixed-size units called pages
- physical memory in frames of the same size

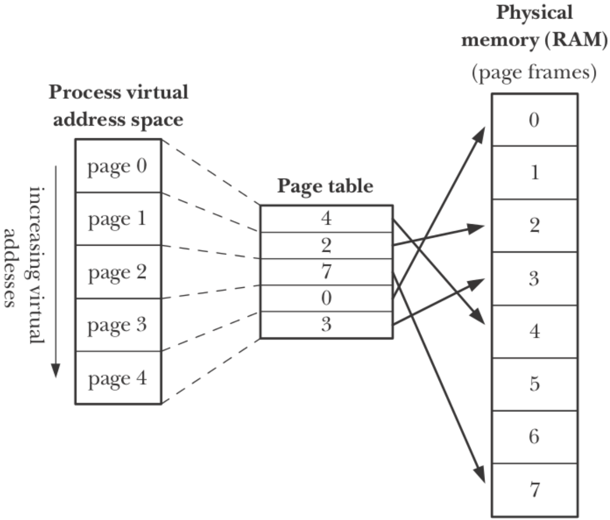
For each process, we maintain a mapping among the two sets.

At any given time, only *some* of the pages of a program (the resident set) need to be present in physical frames. Hence we can:

- swap out unused pages to a *swap area* (usually on disk)
- when a page fault—i.e. access to page $p \notin$ *resident set*—occurs
  1. suspend process execution
  2. *swap in* the corresponding frame
  3. resume process execution

Virtual memory on UNIX

The kernel maintains a page table for each process:



TLPI, Figure 6-2

Virtual memory on UNIX (cont.)

- each entry describes a page of the process virtual address space
- each entry either points to a physical frame, or indicates that the page has been swapped out
- usually, many pages are unused and lack page table entries
  - think about the huge gap among stack and heap addresses

  accessing unused pages terminates a process delivering a SIGSEGV signal

The range of valid virtual pages can change overtime:

- stack grows past previous limits
- memory is (de)allocated by moving the program break
- shared memory is attached/detached
- memory mappings are established/canceled

Effects of virtual memory

As long as swap in / swap out choose pages that fail locality of reference, physical memory is used more (space-)efficiently.

Other effects:

- processes are isolated from one another and from the kernel
- processes can share memory
  - processes can share read-only frames (e.g. text segment)
  - processes can share arbitrary frames (e.g. `mmap`, `shmget`)
- memory access control is easy: capabilities can be attached to page table entries and verified at each access
- programmers (and some toolchain programs—compiler, linker, etc.) can ignore memory physical layout
- lazy loading of programs is possible (and faster)
- virtual memory size can exceed RAM capacity
- CPU efficiency (thanks to swap out, more processes can stay in memory, increasing the likelihood that one is runnable)

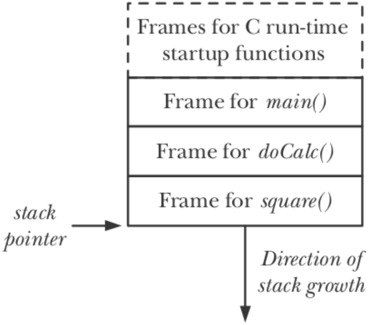
Stack and stack frames

The stack pointer register always points to the top of the stack. Each time a function is called a new frame is allocated; each time a function returns, one is removed.

Each stack frame contains:

- call linkage information: saved copies of various CPU registers. In particular: the program counter, to know where to resume execution of the previous function in the call stack

(x86-32 bit Linux)



TLPI, Figure 6-3

Stack and stack frames (cont.)

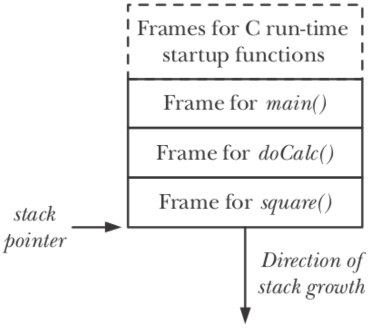
Each stack frame contains (cont.):

- automatic variables
  - function arguments
  - function return values
  - function local variables
  - variables allocated via `alloca`

Automatic variables disappear shortly after the function *call* corresponding to the containing stack frame returns.

Note: stack frames are per-function-*call*, not per-function. Why?

(x86-32 bit Linux)



TLPI, Figure 6-3

Outline

Process families

Processes on UNIX systems form a tree structure:

- each process—other than PID 1—has exactly one parent process
- each process can have 0 or more child processes
- the process with PID 1—usually `init`—has no parent and sits at the root of the process tree

Process families — example

```
$ pstree                          # output trimmed
init -+-NetworkManager-+-dhclient
      |                `-2*[{NetworkManager}]
      |-acpid
      |-atd
      |-chromium-+-2*[chromium]
      |          |-2*[chromium---{chromium}]
      |          `-27*[{chromium}]
      |-cpufreq-applet---{cpufreq-applet}
      |-cron
      |-2*[dbus-daemon]
      |-dconf-service---{dconf-service}
      |-dhclient---dhclient-script---ping
      |-emacs23-+-aspell
      |         `-{emacs23}
      |-emacsclient
      |-gdm3-+-gdm-simple-slav-+-Xorg
      |      |                 |-gdm-session-wor-+-gnome-session-+-awesome
      |      |                 |                 |               |-evolution-alarm---{evolution-a
      |      |                 |                 |               |-gnome-panel---2*[{gnome-panel}]
      |      |                 |                 |               |-gnome-power-man---{gnome-power-
      |      |                 |                 |               |-nautilus---{nautilus}
      |      |                 |                 |               |-nm-applet---{nm-applet}
      |      |                 |                 |               |-notification-da---{notification
      |      |                 |                 |               |-polkit-gnome-au---{polkit-gnome
      |      |                 |                 |               |-ssh-agent
      |      |                 |                 |               `-2*[{gnome-session}]
      |      |                 |                 `-{gdm-session-wo}
      |      |                 `-{gdm-simple-sla}
      |      `-{gdm3}
      |-6*[getty]
$
```

Knowing your family

How can a process know the (PID of) processes in its own family?

Self

`getpid` (already seen)

Parent

```
#include <unistd.h>

pid_t getppid(void);
```

Returns: *parent process ID of calling process*

Children

The PID of children processes is usually retrieved at creation time. . .

fork

An existing process can create a new child process using fork:

```
#include <unistd.h>

pid_t fork(void);
```

Returns: *0 in child, process ID of child in parent, -1 on error*

> *This function is called once but returns twice.*
>
> — *W. Richard Stevens*

1. child process starts execution just after fork
2. parent process continues execution just after fork

fork

An existing process can create a new child process using fork:

```
#include <unistd.h>

pid_t fork(void);
```

Returns: *0 in child, process ID of child in parent, -1 on error*

> *This function is called once but returns twice.*
>
> — *W. Richard Stevens*

1. child process starts execution just after fork
2. parent process continues execution just after fork

Notes:

- often, you want to *differentiate* parent and child behaviors; the difference in return values allows to do that
- child can retrieve parent pid with getppid

fork — example

```c
#include <unistd.h>
#include "apue.h"

int main(void) {
	pid_t pid;

	printf("before fork (%d)\n", getpid());
	if ((pid = fork()) < 0) {
		err_sys("fork error");
	} else if (pid == 0) {  /* child */
		printf("hi from child! (%d -> %d)\n",
			getpid(), getppid());
	} else {		/* parent */
		printf("hi from parent! (%d)\n", getpid());
	}
	printf("bye (%d)\n", getpid());
	exit(EXIT_SUCCESS);
}
```

Note: the above if/else-if/else is a classic fork pattern.

fork — example (cont.)

```
$ ./fork
before fork (16804)
hi from parent! (16804)
bye (16804)
hi from child! (16805 -> 16804)
bye (16805)
$
```

fork and (virtual) memory

- child is a copy of parent
  - child process gets copies of data, heap, and stack segments
  - again: they are copies, not *shared* with the parent
- the text segment is shared among parent and child
  - virtual memory allows to have real sharing (hence reducing memory usage)
  - it is enough to map pages of the two processes to the same frame (which is read-only, in the text segment case)
- no upfront copy is needed, copy-on-write (COW) to the rescue!
  - initially, all pages are shared as above, as if they were read-only
  - if either process writes to these pages, the kernel copies the underlying frame and update the VM mapping before returning

Memory after fork — example

```c
#include <unistd.h>
#include "apue.h"
int     glob = 42;      /* initialized data */

int main(void) {
    int var;              /* automatic variable */
    pid_t pid;
    var = 88;
    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {  /* child */
        printf("child pid: %d\n", getpid());
        glob++; /* modify variables */
        var++;
    } else {              /* parent */
        printf("parent pid: %d\n", getpid());
        sleep(1);
    }
    printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
    exit(EXIT_SUCCESS);
}
```

fork — example (cont.)

```
$ ./fork-2
child pid: 19502
pid = 19502, glob = 43, var = 89
parent pid: 19501
pid = 19501, glob = 42, var = 88
$
```

Termination

Upon process termination (no matter if normal/abnormal, clean/abrupt), the kernel:

- closes all open file descriptors (! = I/O streams)
- releases the process memory

No matter the kind of termination, we want a mechanism to communicate *how* a process terminates to its parent.

- for normal termination → we have `exit`(*status*) & co.
- for abnormal termination → the kernel prepares a termination status

Either way, the kernel stores the termination status—which might contain an exit status or not—*until the parent collects it*.

Reparenting

We've implicitly assumed that there is always a parent process to collect the termination statuses of its children.

Is it a safe assumption?

[3] Yes, upon `init` termination the system crashes

Reparenting

We've implicitly assumed that there is always a parent process to collect the termination statuses of its children.

Is it a safe assumption?

No. Because parent processes can terminate before their children.

Upon termination of a process, the kernel goes through active processes to check if the terminated process had children.
If so, `init` becomes the parent of orphan children.

This way the assumption is made safe.[3]

[3] Yes, upon `init` termination the system crashes

wait

The main facility to retrieve termination status of a child process is:

```
#include <sys/wait.h>
pid_t wait(int *statloc);
```

Returns: *process ID if OK, -1 on error*

upon invocation wait:

- if no children has recently terminated, blocks until one terminates
- if a children has terminated and its termination status has not been collected yet, returns immediately filling *statloc*
- return an error if the calling process has no children

wait — inspecting termination status

The various cases of termination can be inspected applying suitable <sys/wait.h> macros to the integer filled by wait.

- WIFEXITED(*status*) true for normal termination
  - WEXITSTATUS(*status*) can *then* be used to retrieve the exit status
- WIFSIGNALED(*status*) true for abnormal termination due to uncatched signal, then:
  - WTERMSIG(*status*) gives the signal number

Other macros are available for job control.

wait — example

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include "apue.h"

int main(void) {
        pid_t pid;
        int status;
        if ((pid = fork()) < 0)
                err_sys("fork error");
        else if (pid == 0) {    /* child */
                printf("hi from child\n");
                exit(7);
        } else {                /* parent */
                if (wait(&status) != pid)
                        err_sys("wait error");
                printf("hi from parent\n");
                if (WIFEXITED(status))
                        printf("normal termination, exit status = %d\n",
                                WEXITSTATUS(status));
                else if (WIFSIGNALED(status))
                        printf("abnormal termination, signal number = %d\n",
                                WTERMSIG(status));
        }
        exit(EXIT_SUCCESS);
}
```

wait — example

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include "apue.h"

int main(void) {
        pid_t pid;
        int status;
        if ((pid = fork()) < 0)
                err_sys("fork error");
        else if (pid == 0) {    /* child */
                printf("hi from child\n");
                exit(7);
        } else {                /* parent */
                if (wait(&status) != pid)
                        err_sys("wait error");
                printf("hi from parent\n");
                if (WIFEXITED(status))
                        printf("normal termination, exit status = %d\n",
                                WEXITSTATUS(status));
                else if (WIFSIGNALED(status))
                        printf("abnormal termination, signal number = %d\n",
                                WTERMSIG(status));
        }
        exit(EXIT_SUCCESS);
}
```

```
$ ./wait
hi from child
hi from parent
normal termination, exit status = 7
$
```

Helper — pr_exit

```
void pr_exit(int status) {
    if (WIFEXITED(status))
        printf("normal termination, exit status = %d\n",
                WEXITSTATUS(status));
    else if (WIFSIGNALED(status))
        printf("abnormal termination, signal number = %d\n",
                WTERMSIG(status));
}

/* defined from now on in "apue.h" */
```

wait — example

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include "apue.h"

int main(void)
{
        pid_t   pid;
        int     status;

        if ((pid = fork()) < 0)
                err_sys("fork error");
        else if (pid == 0)                      /* child */
                exit(7);
        if (wait(&status) != pid)               /* wait for child */
                err_sys("wait error");
        pr_exit(status);                        /* and print its status */
        if ((pid = fork()) < 0)
                err_sys("fork error");
        else if (pid == 0)                      /* child */
                abort();                        /* generates SIGABRT */
        if (wait(&status) != pid)               /* wait for child */
                err_sys("wait error");
        pr_exit(status);                        /* and print its status */
        if ((pid = fork()) < 0)
                err_sys("fork error");
        else if (pid == 0)                      /* child */
                status /= 0;                    /* divide by 0 generates SIGFPE */
        if (wait(&status) != pid)               /* wait for child */
                err_sys("wait error");
        pr_exit(status);                        /* and print its status */
        exit(EXIT_SUCCESS);
}
```

wait — example (cont.)

```
$ ./wait-2
normal termination, exit status = 7
abnormal termination, signal number = 6
abnormal termination, signal number = 8
$
```

Zombie

(i) Process termination and (ii) collection of termination status are not synchronized actions. They are mediated by the kernel that stores the termination status until it is collected.

Definition

A process that has terminated but whose termination status has not yet been collected is called a zombie process.

Large amounts of zombie processes are undesirable, as they consume resources—the (small) amounts of memory for termination status and entries in the process table.

- if you write a long running program that `fork`s a lot, you should take care of `wait`ing a lot
  - if you don't care about termination status, pass *statloc*=NULL
- `init` automatically collects termination statuses of its children

Zombie — example

```c
#include <stdio.h>
#include <unistd.h>
#include "apue.h"

int main(void) {
    pid_t pid;
    int i;

    for(i = 0; i<5; i++) {
        if ((pid = fork()) < 0) {
                err_sys("fork error");
        } else if (pid == 0) {  /* i-th child*/
                printf("bye from child %d: %d\n", i, getpid());
                exit(EXIT_SUCCESS);
        }
        /* parent does nothing */
    }
    sleep(10);
    printf("bye from parent\n");
    exit(EXIT_SUCCESS);
}
```

Zombie — example (cont.)

Using the previous example, ps, and shell job control we can "appreciate" zombie processes:

```
$ ./zombie &
[1] 4867
$ bye from child 0: 4868
bye from child 2: 4870
bye from child 3: 4871
bye from child 4: 4872
bye from child 1: 4869

$ ps
  PID TTY          TIME CMD
 2597 pts/3    00:00:00 bash
 4867 pts/3    00:00:00 zombie
 4868 pts/3    00:00:00 zombie <defunct>
 4869 pts/3    00:00:00 zombie <defunct>
 4870 pts/3    00:00:00 zombie <defunct>
 4871 pts/3    00:00:00 zombie <defunct>
 4872 pts/3    00:00:00 zombie <defunct>
 4876 pts/3    00:00:00 ps
$
bye from parent
```

Trivia

```
#include <unistd.h>

int main() {
    while(1)
        fork();
}
```

What happens when you run the above program?
Try it out! (or not)