Programmation Systèmes
Cours 2 — Process Management

Stefano Zacchiroli
zack@pps.univ-paris-diderot.fr

Laboratoire PPS, Université Paris Diderot

2013–2014

URL	http://upsilon.cc/zack/teaching/1314/progsyst/

Outline

1. Process startup and termination
2. Memory layout
3. Process control
4. Fork inheritance
5. Program execution
6. Simple UNIX architectures
7. Summary

Programs and processes — redux

Definition (programs and processes — 2nd approx.)

- 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.

Programs — details

A program contains all the 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

```
...
400730: push   %rbx
400731: callq  400700
400736: callq  400720
40073b: mov    %eax,%ebx
40073d: xor    %eax,%eax
40073f: callq  4006f0
400744: mov    %ebx,%eax
400746: pop    %rbx
400747: retq
400748: xor    %ebp,%ebp
40074a: mov    %rdx,%r9
40074d: pop    %rsi
40074e: mov    %rsp,%rdx
400751: and    $0xfffffff0,%rsp
400755: push   %rax
...
```

Processes — seen 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

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
  - on Linux, see `/proc/sys/kernel/pid_max`
- 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: (NULL-terminated) array of pointers to the actual arguments

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

Before main execution, a startup routine—inserted by the link editor at compile-time 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 — example

```c
#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);
}
```

```c
#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 different ways for a program to terminate.

Normal termination

1. return from `main` ("falls off the end")
2. `exit`
3. `_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 cleans up standard library resources before terminating the process:

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

Normal termination — abrupt shutdown

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

Returns: *does not return*

(for our purposes, the two are equivalent)

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.

UNIX convention

Programs terminating with a 0 exit status have terminated successfully; programs terminating with a ! = 0 exit status have failed.

The convention is heavily relied upon 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 explicitly return an integer value; barring standards evolution uncertainty:

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

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

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

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

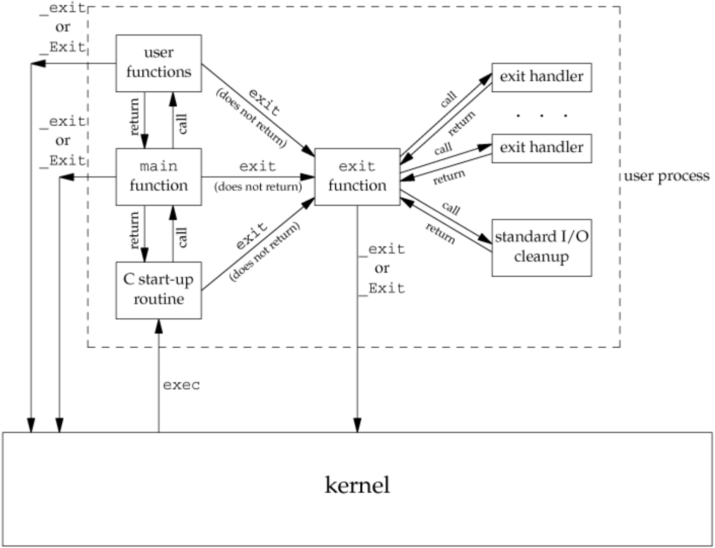
```
#include <stdio.h>
#include <stdlib.h>
#include "helpers.h"

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
```

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

Startup and termination — putting it all together

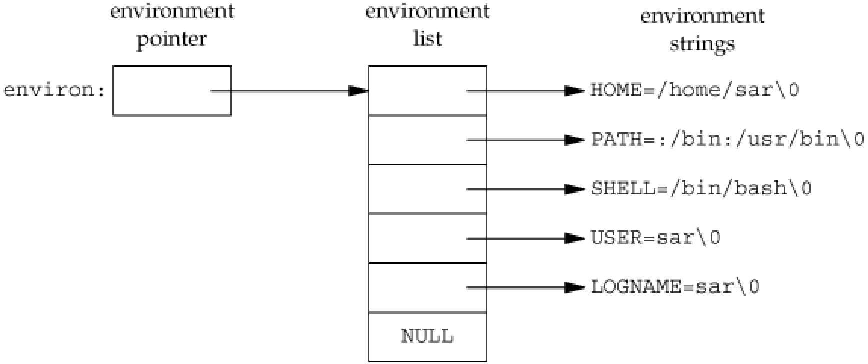


APUE, Figure 7.2

Environment list

Each process is also passed, upon startup, an environment list, i.e. a list of $\langle key, value \rangle$ 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 the 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

```
#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);
}
// end of getenv.c
```

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

1 Process startup and termination

2 Memory layout

3 Process control

4 Fork inheritance

5 Program execution

6 Simple UNIX architectures

7 Summary

Process address space

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.

Segments (1/2)

text segment machine instructions that the CPU executes. It is read from program (usually on disk) upon process creation. The instruction pointer points within this segment.

stack dynamically growing and shrinking segment made of stack frames (more on this in a bit).

heap dynamically growing and shrinking segment, for dynamic memory allocation (see `malloc` and friends). The top of the heap is called program break

Stack and stack frames

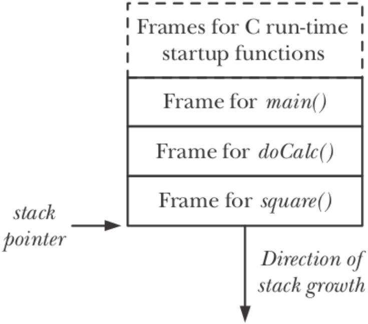
The stack pointer register always points to the top of the stack.

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 instruction pointer, 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.):

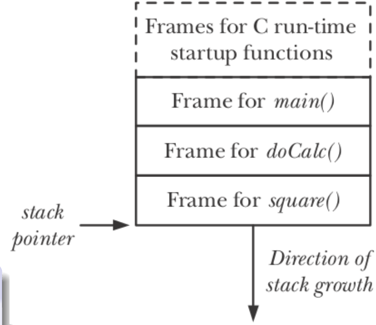
- automatic variables
  - function arguments
  - function return values
  - function local variables
  - variables allocated via `alloca`

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

Exercise

Stack frames are per-function-call, not per-function. Why?

(x86-32 bit Linux)



TLPI, Figure 6-3

Segments (2/2)

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

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

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

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

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

Segments (2/2)

initialized data segment (AKA “data segment”) global variables explicitly initialized, e.g.:

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

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

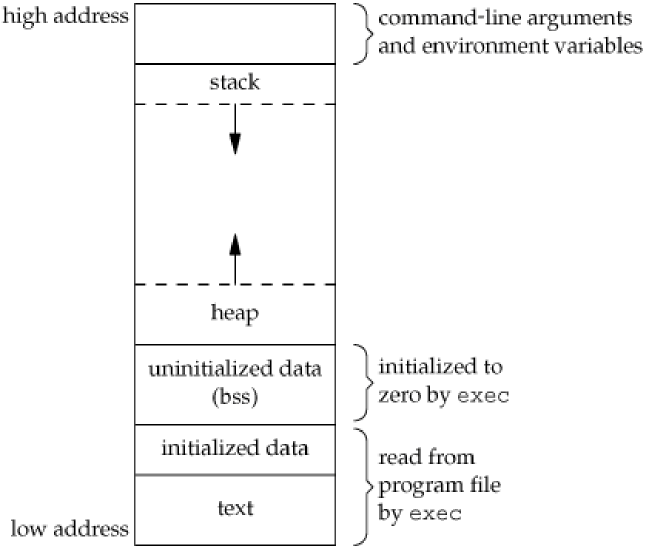
```
char crap[1024];   // outside any function
```

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

On “static memory”

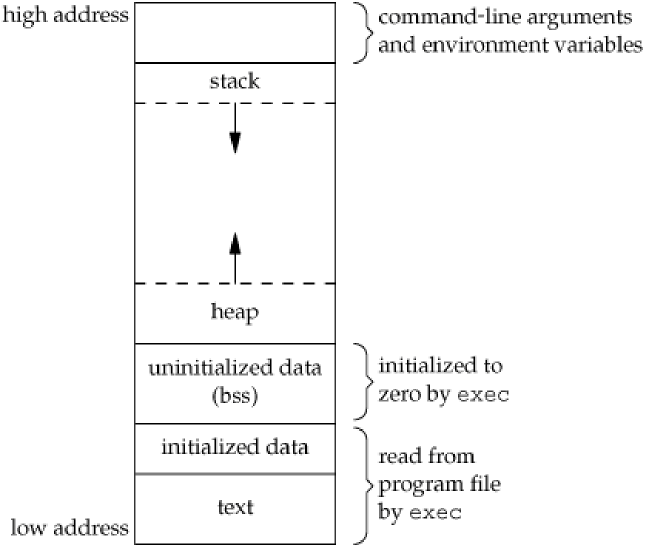
The imprecise expression “static memory” refers to memory allocated in the data or bss segments. Such memory is “static” wrt program execution (which is not the case for stack/heap-allocated memory).

Typical segment arrangement


high address
low address
stack
heap
uninitialized data (bss)
initialized data
text
command-line arguments and environment variables
initialized to zero by exec
read from program file by exec

APUE, Figure 7.6

Typical segment arrangement


high address
command-line arguments and environment variables
stack
heap
uninitialized data (bss)
initialized to zero by exec
initialized data
text
read from program file by exec
low address

APUE, Figure 7.6

Exercise

Experimentally verify that segment matches this diagram.

Segment arrangement — demo

```c
#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)

Also, `size(1)` displays the size of segments stored in program binaries:

```
$ size segments
   text    data     bss     dec     hex filename
   1657     596    1056    3309     ced segments
$
```

Virtual memory

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

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

- *spatial locality*: tendency to reference memory addresses near recently addressed ones
- *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

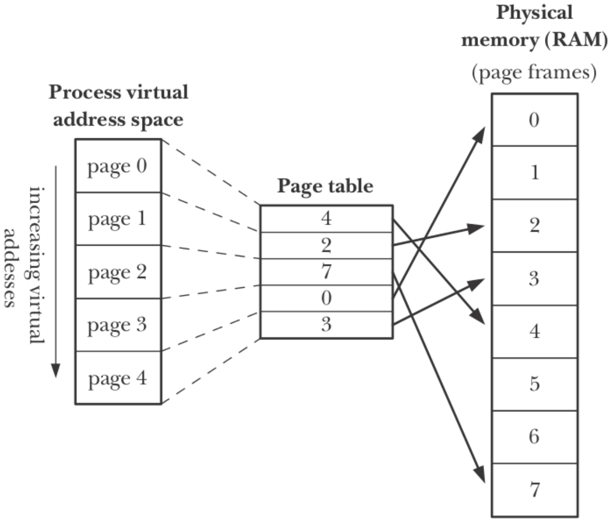
For each process, we maintain a mapping among the two sets.

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


Process virtual address space
increasing virtual addesses
page 0
page 1
page 2
page 3
page 4
Page table
4
2
7
0
3
Physical memory (RAM) (page frames)
0
1
2
3
4
5
6
7

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 don't even have page table entries
  - e.g.: the huge gap between 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

Benefits of virtual memory

As long as swap out chooses pages that fail locality of reference, *physical* memory is used (space-)efficiently.

Other effects:

- processes are isolated from one another and from the kernel
- memory access control is easy: capabilities can be attached to page table entries and verified at each access
- processes can share memory
  - processes can share read-only frames (e.g. text segment)
  - processes can share arbitrary frames (see `mmap`, `shmget`)
- developers (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)

Outline

1 Process startup and termination

2 Memory layout

3 Process control

4 Fork inheritance

5 Program execution

6 Simple UNIX architectures

7 Summary

Process families

Processes on UNIX systems are arranged in a tree structure:

- each process—other than the root—has exactly one parent process
- each process can have 0 or more child processes
- the root process—usually called `init` and associated to PID 1—has no parent and it is the first process, created by the kernel at boot time

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-gnom
     |      |                 |                 |               |-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:

- you almost always want to *differentiate* parent and child behaviors; the different return values allow to do that
- child can retrieve parent pid with getppid

fork — example

```
#include <unistd.h>
#include "helpers.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 common fork pattern.

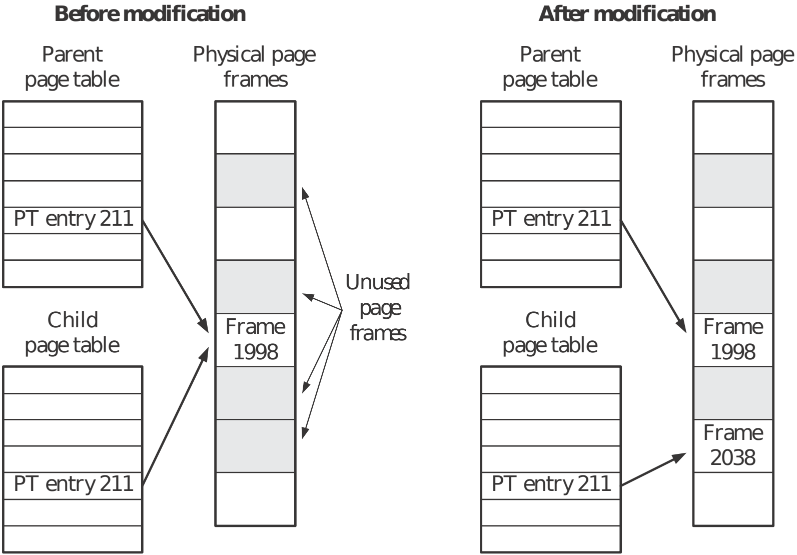
fork — example (cont.)

```
$ ./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

Copy-on-write — example



TLPI, Figure 24-3

Memory after fork — example

```
#include <unistd.h>
#include "helpers.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);
}
```

Memory after 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?

Not necessarily.
Parent processes might 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: *child 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, collect it 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 "helpers.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 (cont.)

```
$ ./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 "helpers.h" */
```

wait — example

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

Observation

(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.

Zombie

Observation

(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 (zombie process)

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 + entries in the process table.

- if you write a long running program that forks 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 "helpers.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(60); /* time window to observe zombies */
    printf("bye from parent\n");
    exit(EXIT_SUCCESS);
}
```

Zombie — example (cont.)

Using the previous example, ps, and shell job control we can observe 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

```c
#include <unistd.h>

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

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

Outline

1. Process startup and termination
2. Memory layout
3. Process control
4. Fork inheritance
5. Program execution
6. Simple UNIX architectures
7. Summary

forking & flushing

Many system resources associated to processes are duplicated and inherited by child processes

- memory copying (on write) is just an instance of that

Memory copy has an effect on buffers that are stored in memory:

- most notably: standard I/O buffers
- but also: any other user-space buffer layer you might be using

forking & flushing (cont.)

```
#include <unistd.h>
#include "helpers.h"
#include <string.h>

char buf[] = "write to stdout\n";
int main(void) {
        pid_t pid;
        if (write(STDOUT_FILENO, buf, strlen(buf)) != strlen(buf))
                err_sys("write error");
        printf("printf by %d: before fork\n", getpid());
        if ((pid = fork()) < 0) {
                err_sys("fork error");
        } else if (pid == 0) {  /* child */
                printf("printf by %d: hi from child!\n", getpid());
        } else {                /* parent */
                printf("printf by %d: hi from parent!\n", getpid());
        }
        printf("printf by %d: bye\n", getpid());
        exit(EXIT_SUCCESS);
}
```

What do you expect from the above code? (up to interleaving)

forking & flushing (cont.)

```
$ ./fork-flush
write to stdout
printf by 13495: before fork
printf by 13495: hi from parent!
printf by 13495: bye
printf by 13496: hi from child!
printf by 13496: bye
$
```

forking & flushing (cont.)

```
$ ./fork-flush
write to stdout
printf by 13495: before fork
printf by 13495: hi from parent!
printf by 13495: bye
printf by 13496: hi from child!
printf by 13496: bye
$

$ ./fork-flush > log
$ cat log
write to stdout
printf by 10758: before fork
printf by 10758: hi from parent!
printf by 10758: bye
printf by 10758: before fork
printf by 10759: hi from child!
printf by 10759: bye
$
```

why output redirection changes the result?

forking & flushing (cont.)

The `write` syscall is not (user-space) buffered, executing it before forking ensures that data is written exactly once.
The standard I/O library is buffered, if buffers are not flushed *before* `fork`, multiple writes can ensue.

- when stdout is connected to a terminal (the case with no redirection) the STDOUT stream is line-buffered
  - each newline triggers a flush
  - hence `printf` content gets flushed *before* fork and is delivered only once
- otherwise (the redirection case), stdout is fully-buffered
  - flushs are delayed past fork, hence `printf` content might get duplicated

See: `setvbuf(3)`

Similar issues might affect any other user-space buffering layer. . .

File sharing

Another relevant resource inherited by child processes are file descriptors.

Upon fork all file descriptors of the parent are duplicated into the the child.

- duplication is performed by the kernel, as if dup had been called on each of them
- as a consequence, parent and child share a file table entry (in kernel space) for every file descriptor that was open at the time of fork

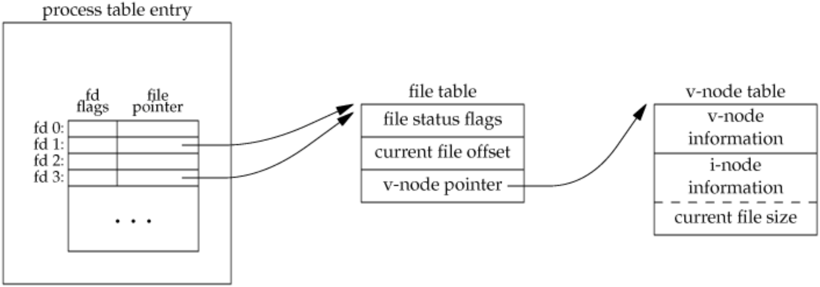
Reminder — dup

```
#include <unistd.h>

int dup(int filedes);

int dup2(int filedes, int filedes2);
```

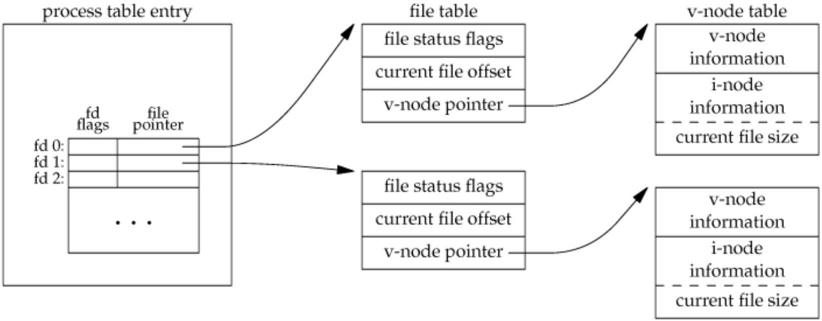
Returns: *new file descriptor if OK, -1 on error*



APUE, Figure 3.8

sample situation after `dup(STDIN_FILENO);`

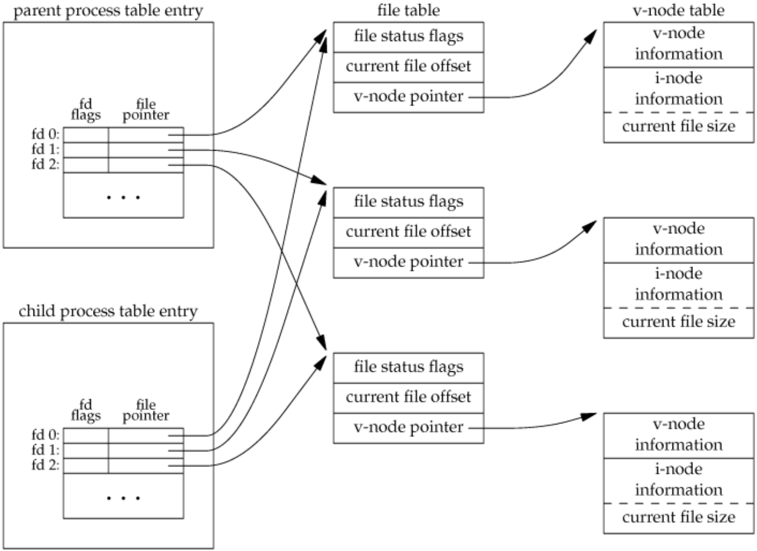
File sharing — before fork



APUE, Figure 3.6

(same for standard error, not shown)

File sharing — after fork



APUE, Figure 8.2

File sharing — offsets

File offsets are stored in the file table, therefore after `fork` file offsets are shared among parent and child

- if a process moves the offset, the other will see the displacement
- remember: movements also happen *implicitly* at each read/write on a file descriptor

This features helps creating software architectures where related processes collaborate on the same open files.

Example (fork-based architecture)

1. parent `fork` a child to do some task and `wait` for it to complete
2. child uses standard file descriptors as part of his task
3. upon child exit, parent can resume using standard file descriptors without having to fiddle with offsets[a]

[a] you just need to be careful with buffering

File sharing — interleaving

The previous architecture works because the parent ensures child goes first, waiting for it. In the general case, parent and child should not use shared files “at the same time”. Doing so would result in garbled I/O due to interleaving issues.

There are 3 main approaches to file sharing after fork:

1. the parent waits for child to complete (previous example) and do nothing with its file descriptors in the meantime
2. parent and child go different ways; to avoid interleaving issues each process closes the file descriptors it doesn’t use (the set of shared files should be empty after closes)
3. parent and child maintain a set of shared files and synchronize access to them; *goal*: ensure that at any given time only one process is acting on a shared file

Common traits after fork

In addition to memory and file descriptors, many other resources are inherited by child processes:[4]

- user & group IDs (real, effective)
- process group & session IDs
- set-(user/group)-IDs flags
- controlling terminal
- current working & root directory
- umask
- signal mask
- close-on-exec flags
- environment variables
- shared memory & memory mappings
- resource limits

[4] we've discussed only some of them up to now

Distinguishing traits after fork

On the other hand, differences remain among parent and child:[5]

- fork return value
- process and parent process IDs
- time accounting counters are set to 0 in the child
- file locks are not inherited
- pending alarms (cleared for the child)
- pending signals (set to the empty set for the child)

[5] ditto: we've discussed only some of them up to now

fork failures

fork can essentially only fail due to resource exhaustion

1. if the maximum number of processes in the system has been reached, or if there is not enough memory to create a new process table entry (ENOMEM)
2. if the maximum number of processes *allocated to the current user* has been reached
    - this is when the fork bomb will stop (creating new processes. . .)

      ```
      #include <unistd.h>

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

    - related trivia, what does the following shell code do?
      `:(){ :|:& };:`

Outline

1. Process startup and termination
2. Memory layout
3. Process control
4. Fork inheritance
5. Program execution
6. Simple UNIX architectures
7. Summary

Fork use cases

There are two main use cases for fork:

1. a process wants to duplicate itself to concurrently execute (possibly different) parts of the same program
   - e.g. network/system daemons and other services that want to serve multiple requests concurrently
   - the parent idles waiting for requests; it forks a new child to handle each incoming request
2. a process wants to execute a different program
   - e.g. shells
   - e.g. software architectures organized into a set of independent executables (typical products of the UNIX philosophy)

We've seen how fork deals with (1).

UNIX's recipe for (2) is to first fork and then, in the child process, use a syscall of the exec family to execute a new program.

Exec

In the UNIX tradition, creating a new process and executing a program are separate actions.

Therefore exec does *not* create a new process. Rather:

- it replaces the program run by the invoking process with a new one, obtained reading a binary from the filesystem
- all segments—text, data, bss, heap, and stack—are re-initialized as if the program were being executed from scratch
  - in fact *it is*, that's why we've seen exec in `strace` logs before
- the usual startup routine is executed up to handing control over to `main()` of the new program

Note: if you plainly exec something (without `fork`), you "throw away" your program.

The exec family

We often write “exec” for short, but there are 6 different exec syscalls:

```
#include <unistd.h>
int execl(const char *pathname, const char *arg0, ...);
int execv(const char *pathname, char *const argv[]);
int execle(const char *pathname, const char *arg0, ..., char *const envp[]);
int execve(const char *pathname, char *const argv[], char *const envp[]);
int execlp(const char *filename, const char *arg0, ...);
int execvp(const char *filename, char *const argv[]);
```

Returns: *all return -1 on error, don’t return on success*

All execute an external program replacing the current one.

Note: on success they *do not return*.

exec differences

Don't panic!

Differences among execs can be organized along 3 axes.

1. "Command line" argument specification: `argv` for the new program can be given either as *lists* or *arrays* (or "vectors") of pointers.
    - list of pointers must be NULL-terminated
        - as, no wonder, it happens for `argv` in `main()` calling convention
    - note: the calling convention allows to have `basename`(*path*) ≠ `argv[0]`

Mnemonic

execs containing the '`l`' character (3 of them) take list of pointers; execs containing the '`v`' character (3 of them) take vectors.

exec differences (cont.)

Don’t panic!

Differences among execs can be organized along 3 axes.

2. Program specification: the program to be executed can be given either as a *pathname* or as a *filename* argument.
    - a filename argument that does not contain "/" will be searched according to the PATH environment variable

   e.g.

   ```
   $ echo $PATH
   /home/zack/bin:/usr/local/bin:/usr/bin:/bin:/sbin/:/usr/sbin/
   ```

Mnemonic

execs containing the ‘p’ character (2 of them) take filename arguments (i.e. they *search in the path*); other execs (4 of them) take pathname arguments.

exec differences (cont.)

Don't panic!
Differences among execs can be organized along 3 axes.

3. Environment list specification: `*environ` for the new program can be given either implicitly or explicitly
   - in the implicit case, the environment list is copied from the calling process
   - in the explicit case, a list of pointers to `"key=value"` strings shall be given
     - as, no wonder, it happens for `*environ` in `main()` calling convention

Mnemonic

execs containing the 'e' character (2 of them) take an environment list argument; other execs don't.

exec relationships

Ultimately:

mnemonics and `man 3 exec` are your friends.

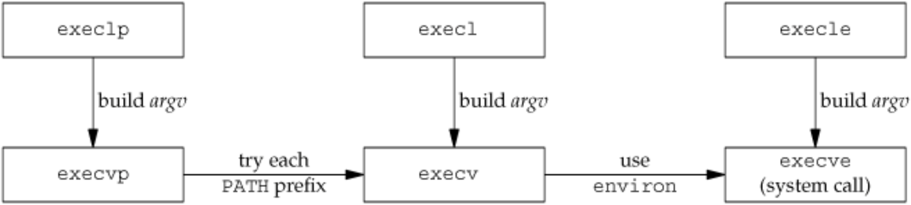
(... but why section 3?)

exec relationships

Ultimately:

mnemonics and `man 3 exec` are your friends.

Note: on most UNIX the only syscall is execve (documented in section 2), whereas other execs are wrapper functions:



APUE, Figure 8.15

Helper — echoall

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

extern char **environ;

int main(int argc, char *argv[])
{
        int i;
        char **ptr;

        for (i = 0; i < argc; i++)      /* dump argv */
                printf("argv[%d]: %s\n", i, argv[i]);
        for (ptr = environ; *ptr != NULL; ptr++) /* dump env */
                printf("%s\n", *ptr);

        exit(EXIT_SUCCESS);
}
// end of echoall.c
```

exec — example

```c
#include <unistd.h>
#include <sys/wait.h>
#include "helpers.h"
char    *env_init[] = { "USER=unknown", "TERM=xterm", NULL };
int main(void) {
        pid_t pid;
        if ((pid = fork()) < 0) {
                err_sys("fork error");
        } else if (pid == 0) {  /* child */
                if (execle("/tmp/echoall",      /* path, env */
                           "echoall", "foo", "BAR", NULL,
                           env_init) < 0)
                        err_sys("execle error");
        }
        if (wait(NULL) < 0)
                err_sys("wait error");
        if ((pid = fork()) < 0) {
                err_sys("fork error");
        } else if (pid == 0) {  /* file, no env */
                if (execlp("echoall",
                           "echoall", "only 1 arg", NULL) < 0)
                        err_sys("execlp error");
        }
        exit(EXIT_SUCCESS);
} // end of exec.c
```

exec — example (cont.)

Demo

exec inheritance

Similarly to the fork case, the exec-uted program inherits some traits from the former process:

- process and parent process ID
- real user & group ID
- process group and session ID
- file descriptors
- controlling terminal
- pending alarm (not reset)
- current working & root directory
- umask
- file locks
- signal mask and pending signals
- resource limits
- time accounting counters
- environment list

for non-‘e’ exec

exec limits and xargs

Part of the communication “bandwidth” among parent and child processes is related to the maximum size of the argument and environment lists.
POSIX.1 guarantees that such a limit is at least 4096 arguments / environment variables.

Even if such a limit is pretty large, one way to hit it is playing with shell globbing, e.g. `grep execve /usr/share/man/*/*`, depends on your system. . . .

As a solution for shell globbing, you might resort to xargs(1).

system

As a shorthand to execute a system program from a running process, the libc offers:

```
#include <stdlib.h>

int system(const char *cmdstring);
```

On UNIX systems, system takes care of fork, exec, wait, and return its termination status.

- process management is hidden from the invoking process
  - the function is both ISO C and POSIX, but in the ISO standard it's heavily implementation-dependent
- the command is given as a string that will be interpreted by the system shell
  - you need a standard shell for this, according to POSIX it's /bin/sh

system (cont.)

As a shorthand to execute a system program from a running process, the libc offers:

```
#include <stdlib.h>

int system(const char *cmdstring);
```

Return value:

- if either fork or wait fail → -1
- if the shell cannot be executed (exec failure) → 127
- otherwise → shell termination status

- special case: passing NULL as an argument should return non-zero if system is supported—which is always the case on UNIX—; 0 otherwise

system — example

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/wait.h>
#include "helpers.h"

char *cmd = "grep zack /etc/passwd | cut -f 5 -d:";

int main(void) {
        int status;
        if ((status = system(cmd)) == -1)
                err_sys("system error");
        pr_exit(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)); */
        exit(EXIT_SUCCESS);
}
```

system — example (cont.)

```
$ ./system
Stefano Zacchiroli,,,
normal termination, exit status = 0
$
```

Notes:

- the command makes use of shell meta characters, pipelines, etc.
- fork-based file sharing is implicitly used by system
- termination status can be inspected via the usual `wait` macros

Exercise: `system` implementation

We want to provide an implementation of `system`.

Given that it is a regular function rather than a syscall, we can provide such an implementation in user space.

Requirements:

1. the implementation shall match `system` prototype
2. the implementation shall obey `system` return convention
3. the implementation shall not interfere with the calling process

system — implementation

```c
#include <sys/wait.h>
#include <errno.h>
#include <unistd.h>

int system(const char *cmd) {
        pid_t pid;
        int status;
        if (cmd == NULL)
                return(1);      /* system is available */
        if ((pid = fork()) < 0) {
                status = -1;    /* fork error */
        } else if (pid == 0) {  /* child */
                execl("/bin/sh", "sh", "-c", cmd, NULL);
                _exit(127);     /* exec error */
        } else {                /* parent */
                wait(&status);
        }
        return(status);
}
```

system — implementation

```
#include <sys/wait.h>
#include <errno.h>
#include <unistd.h>

int system(const char *cmd) {
        pid_t pid;
        int status;
        if (cmd == NULL)
                return(1);      /* system is available */
        if ((pid = fork()) < 0) {
                status = -1;    /* fork error */
        } else if (pid == 0) {  /* child */
                execl("/bin/sh", "sh", "-c", cmd, NULL);
                _exit(127);     /* exec error */
        } else {        /* parent */
                wait(&status);
        }
        return(status);
}
```

Is this solution correct?

system — (*buggy*) implementation

```
#include <sys/wait.h>
#include <errno.h>
#include <unistd.h>

int system(const char *cmd) {
        pid_t pid;
        int status;
        if (cmd == NULL)
                return(1);      /* system is available */
        if ((pid = fork()) < 0) {
                status = -1;    /* fork error */
        } else if (pid == 0) {  /* child */
                execl("/bin/sh", "sh", "-c", cmd, NULL);
                _exit(127);     /* exec error */
        } else {        /* parent */
                wait(&status);
        }
        return(status);
}
```

This solution fails requirement (3): `wait` can retrieve the termination status of a process other than the shell executed by `system`.

waitpid

To solve the problem we need a new syscall, capable of waiting for the termination of a specific process. Enter `waitpid`:

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

pid_t waitpid(pid_t pid, int *statloc, int options);
```

Returns: *process ID if OK, 0 or -1 on error*

waitpid

To solve the problem we need a new syscall, capable of waiting for the termination of a specific process. Enter waitpid:

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

pid_t waitpid(pid_t pid, int *statloc, int options);
```

Returns: *process ID if OK, 0 or -1 on error*

The child to wait for depends on the *pid* argument:

pid == 1	waits for any child (wait-like semantics)
pid > 1	waits for a specific child, that has *pid* as its PID
pid == 0	waits for any child in the same process group of caller
pid < -1	waits for any child in the process group abs(*pid*)

options provide more control over waitpid semantics; it is a bitwise OR of flags that include:

- WNOHANG do not block if child hasn't exited yet (and return 0)

system — a rough implementation

```c
#include <sys/wait.h>
#include <errno.h>
#include <unistd.h>

int system(const char *cmd) {   /* no signal management */
        pid_t pid;
        int status;
        if (cmd == NULL)
                return(1);      /* system is available*/
        if ((pid = fork()) < 0) {
                status = -1;    /* fork error */
        } else if (pid == 0) {  /* child */
                execl("/bin/sh", "sh", "-c", cmd, NULL);
                _exit(127);     /* exec error */
        } else {        /* parent */
                while (waitpid(pid, &status, 0) < 0) {
                        if (errno != EINTR) {
                                status = -1; /* generic error */
                                break;
                        }
                }
        }
        return(status);
}  // based on APUE, Figure 8.22
```

waitid

The more recent waitid syscall provides even more flexibility and a saner interface to wait for specific processes:

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

int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

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

id is interpreted according to the value of *idtype*:

P_PID	wait for a process with PID *id*
P_PGID	wait for any child process in process group *id*
P_ALL	wait for any child

options is a bitwise OR of states the caller wants to monitor:

WSTOPPED	wait for a stopped process
WCONTINUED	wait for a (stopped and then) continued process
WEXITED	wait for terminated processes
WNOWAIT	leave the process in zombie state
WNOHANG	as per waitpid

Outline

1. Process startup and termination
2. Memory layout
3. Process control
4. Fork inheritance
5. Program execution
6. Simple UNIX architectures
7. Summary

(fork && exec) || spawn ?

Some (non-UNIX) operating systems combine fork and exec in a single operation called spawn.

UNIX's separation is convenient for various reasons:

1. there are use cases where fork is useful alone
2. when coupled with inheritance, the separation allows to change per-process attributes between fork and exec, e.g.:
   - set up redirections
   - change user IDs (e.g. to drop privileges)
   - change signal masks
   - set up execution "jails"
   - ...

fork+exec = cheap!

But we need to watch out that the fork+exec does not induce unacceptable performance penalty wrt the spawn approach.

Qualitative considerations:

- an important part of fork/exec cost is writing segments into memory
- for fork the cost is minimized by virtual memory and copy-on-write
- for exec the cost lower bound is program loading from disk
- extra process management bookkeeping should be measured

Quantitative analysis:

- (Linux) fork/exec shown to be twice as fast than (Win NT) spawn

Randy Appleton
Improving context switching performance for idle tasks in Linux
CATA 1999
http://math.nmu.edu/~randy/Research/Papers/Scheduler/

UNIX philosophy in a nutshell

> *This is the Unix philosophy: Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface.*
>
> — *Doug McIlroy (inventor of UNIX pipes)*
> *in "A Quarter Century of Unix"*

Practically, the UNIX style of designing do-one-thing-well architectures is multiprocessing, i.e. breaking down applications into small programs that communicate through well-defined interfaces.

Enabling traits for this are:

1. cheap and easy process spawning (i.e. `fork/exec`)
2. methods that ease inter-process communication
3. usage of simple, transparent, textual data formats

UNIX philosophy — some consequences

- we aim for a reduction in global application complexity
- if not, at least individual programs tend to be more manageable

- we focus on stable interfaces and we are encouraged to think upfront at data formats and protocols
- this make it easier to adapt the software to different contexts (interfaces, other data sources, and sinks); it makes the software more *hackable*
- it also encourages cooperation with other developers, especially—but not only—when source code is available

UNIX philosophy — references

In-depth discussion of UNIX philosophy is outside the scope of this course. We will only highlight typical architectures that are enabled by specific UNIX programming interfaces, as we encounter them.

Eric S. Raymond
The Art of UNIX Programming
Addison-Wesley Professional, 2003.
http://www.faqs.org/docs/artu/

Shelling out — the simplest fork architecture

Most UNIX architectures are based on IPC mechanisms we haven't yet discussed. But the simplest architecture, based on cheap process spawning, only needs the process management primitives we've already introduced.

Definition (Shelling out)

Shelling out is the practice of delegating tasks to external programs, handing over terminal to them for the duration of the delegation, and waiting for them to complete.

It is called *shell*-ing out as it has been traditionally implemented using `system`, which relies on the system *shell*.

Shelling out — discussion

- communication is minimal, when compared to other IPC architectures
  - control information: termination status, argv, environment
  - protocol design is not an issue
- data are passed via the filesystem
  - data format design is a concern

Typical shell out work-flow

1. parent create a temporary file and write data to it
2. parent shell out a child passing a path to the temporary file
3. child work on the temporary file
4. child exit
5. parent (re-)read temporary file and delete it

Case study: the `mutt` mail user agent

> *"All mail clients suck. This one just sucks less."*
>
> — *Mutt homepage, `http://www.mutt.org`*

`mutt` is (one of) the most popular console-based Mail User Agent on UNIX systems. It implements a typical shelling out use case: shelling out an editor.

When asked to compose a mail, Mutt:

1. examines the `EDITOR` and `VISUAL` environment variable to figure out user preferred editor
2. creates a temporary file
   - fills it in with a mail template (e.g. headers, signature, quoted text, etc.)
3. spawn the editor on the temporary file
4. [the user uses the editor to write the mail and then quits]
5. parses the composed email from the temporary file, delete it
6. resume normal operation (e.g. to propose sending the email)

Shelling out — example

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include "helpers.h"

char tpl[] = "From: \nTo: \nCc: \nBcc: \nSubject: \n\n";
int main(void) {
        char tmp[] = "/tmp/shellout.XXXXXX";
        char cmd[1024];
        int fd, status;
        if ((fd = mkstemp(tmp)) == -1)  err_sys("mktemp error");
        if (write(fd, tpl, strlen(tpl)) != strlen(tpl))
                err_sys("write error");
        /* Exercise: support insertion of ~/.signature, if it exists */
        if (close(fd) == -1)  err_sys("close error");
        if (snprintf(cmd, sizeof(cmd), "/usr/bin/vim %s", tmp) < 0)
                err_sys("snprintf error");  /* Exercise: use $EDITOR */
        if ((status = system(cmd)) == -1) /* shoud inspect better... */
                err_sys("system error");
        snprintf(cmd, sizeof(cmd), "echo ---; cat %s; echo ---", tmp);
        system(cmd);     /* dummy mail processing */
        if (unlink(tmp) == -1)  err_sys("unlink error");
        exit(EXIT_SUCCESS);
} // end of shellout.c
```

Shelling out — example

Demo

Race conditions

Shelling out, the risk of unwanted interference among parent and child processes is almost non-existent.[6] Other fork-based architectures won't be so lucky.

Definition (Race condition)

A *race condition* occurs when multiple processes cooperate on shared storage and the correctness of the overall result depends on the order in which the processes are run (a factor which is, in general, outside our control).
Intuition: the processes "race" to access the shared storage.

We want to avoid race conditions to preserve deterministic program behavior and to avoid corrupting shared data structures. Race conditions are hard to debug, because—by definition—they are hard to reproduce.

[6] except for signals, that the parent should block while executing child

fork race conditions

fork is a common source of race conditions: we cannot tell (in a portable way. . .) which process—parent or child—goes first.

If output correctness depends on that ordering, you have a problem.

The syscall sleep does not solve the problem, at best it mitigates it

- e.g. under heavy load it is possible that the non-sleeping process is delayed so much, that the sleeping process goes first anyhow

Race conditions — example

```
#include <stdio.h>
#include <unistd.h>
#include "helpers.h"

void charatatime(char *str) {
        char *ptr;
        int c;
        setbuf(stdout, NULL);   /* set unbuffered */
        for (ptr = str; (c = *ptr++) != 0; )
                putc(c, stdout);
}
int main(void) {
        pid_t pid;
        if ((pid = fork()) < 0)  err_sys("fork error");
        else if (pid == 0) {
                charatatime("output from child\n");
        } else {
                charatatime("output from parent\n");
        }
        exit(EXIT_SUCCESS);
} // end of race.c
```

Race conditions — example (cont.)

Desired behaviour

one line of output for the parent, one line of output for the child (in an arbitrary order)

Note: arbitrariness is not *always the* problem.

Demo

ain't always easy to reproduce a race condition...

```
while true ; do ./race ; done
```

Polling

Example

A child process wants to wait until its parent has terminated.
The situation can be detected as `getppid() == 1`.

One way to solve the race condition problem is by polling, i.e. periodically checking if the situation has happened:

```
while (getppid() != 1)
    sleep(1)
```

The problem with polling is that it keeps the CPU busy.

Goal: get rid of race conditions avoiding polling.

Parent/child synchronization

We need synchronization primitives that processes can use to synchronize and avoid race conditions.

As a proof of concept we will consider the following primitives:[7]

WAIT_PARENT child blocks waiting for (a "signal" from) parent

WAIT_CHILD parent blocks waiting for (a "signal" from) children

TELL_PARENT(pid) child "signals" parent

TELL_CHILD(pid) parent "signals" child

Note: they allow synchronization only at the parent/child border. But that gives all the expressivity we need, given that the only way to create new processes is fork.

[7] we'll also have TELL_WAIT in both processes, for initialization

Tell/Wait — intended usage

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

    TELL_WAIT();

    if ((pid = fork()) < 0)  err_sys("fork error");
    else if (pid == 0) {
        WAIT_PARENT();    /* parent first */
        charatatime("output from child\n");
    } else {
        charatatime("output from parent\n");
        TELL_CHILD(pid);
    }
    exit(EXIT_SUCCESS);
}
```

Tell/Wait — intended usage (cont.)

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

    TELL_WAIT();

    if ((pid = fork()) < 0)  err_sys("fork error");
    else if (pid == 0) {
        charatatime("output from child\n");
        TELL_PARENT(getppid());
    } else {
        WAIT_CHILD();    /* child first */
        charatatime("output from parent\n");
    }
    exit(EXIT_SUCCESS);
}
```

Tell/Wait — exercise

Exercise

Provide an implementation of the tell/wait primitives.

we'll see several alternative implementations in the future. . .

vfork

Many forks are followed by an exec. In those cases, duplicating the parent address space is not needed, as it'll be replaced.

When duplicating address space was expensive—i.e. before the advent of COW—vfork provided a cheaper alternative.

```
#include <unistd.h>

pid_t vfork(void);
```

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

It is identical to fork except that:

1. the child executes in the same address space of the parent
2. the parent blocks until child exec or _exit

Thou shalt not use this.[a]

[a] although no copying is better than some copying...

vfork subtleties

- vfork might lead to deadlocks if the child wait for the parent, because the parent is also blocked. . .
- most actions performed in the child will affect the parent when it resumes
  - touching data, heap, stack
  - changing process properties, etc.
- some actions do not affect the parent
  - actions on file descriptors, as the file table is in kernel space and it's duplicated by vfork
  - yep, it's tricky. . .
- vfork guarantees that child goes first and can be used as a (very dangerous!) synchronization primitive

vfork — example

```
#include <stdio.h>
#include <unistd.h>
#include "helpers.h"

int glob = 6;   /* initialized data */
int main(void) {
        int var;        /* automatic variable */
        pid_t pid;
        var = 88;
        printf("before vfork\n");
        if ((pid = vfork()) < 0) {
                err_sys("vfork error");
        } else if (pid == 0) {  /* child */
                glob++; /* modify parent's variables */
                var++;
                _exit(0);       /* child terminates */
        } /* parent continues here... */
        printf("pid = %d, glob = %d, var = %d\n", getpid(),
               glob, var);
        exit(EXIT_SUCCESS);
} // based on APUE, Figure 8.3
```

vfork — example (cont.)

```
$ ./vfork
before vfork
pid = 6121, glob = 7, var = 89
$
```

Notes:

- no need for the parent to sleep (child goes first)
- variable changes are visible in the parent (same address space)
- we use `_exit` instead of `exit`
  - to avoid clean shutdown of Standard I/O, as the parent might still want to use it

Outline

1. Process startup and termination
2. Memory layout
3. Process control
4. Fork inheritance
5. Program execution
6. Simple UNIX architectures
7. Summary

The story so far

- basics of user- and kernel-space
- process management primitives: the `fork` / `wait` / exec triad
- very primitive IPC: arguments, exit codes, file system
- challenges: race conditions

Ahead of us:

- communication mechanisms
- synchronization mechanisms

Trivia — what will second child print?

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

int main(void) {
        pid_t pid;

        if ((pid = fork()) < 0) {
                err_sys("fork error");
        } else if (pid == 0) {          /* 1st child */
                if ((pid = fork()) < 0)
                        err_sys("fork error");
                else if (pid > 0)  /* parent from 2nd fork == 1st child */
                        exit(EXIT_SUCCESS);
                /* 2nd child */
                sleep(2);
                printf("2nd child, parent pid = %d\n", getppid());
                exit(EXIT_SUCCESS);
        }
        if (waitpid(pid, NULL, 0) != pid)       /* wait for 1st child */
                err_sys("waitpid error");
        /* parent (original process) */
        exit(EXIT_SUCCESS);
} // based on APUE, Figure 8.8
```

Double fork

We want to write a program that `fork` a child and wants to both avoid `waiting` for it and avoid zombies.

How to do that?

Double fork (cont.)

We want to write a program that fork a child and wants to both avoid waiting for it and avoid zombies.

How to do that? Using the double fork technique.

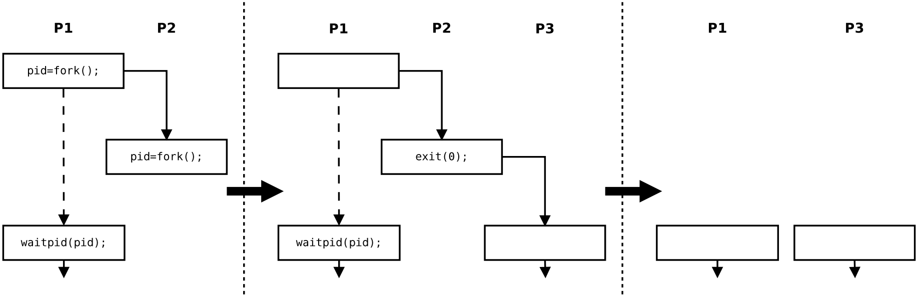
Double fork

1. parent forks a 1st child and wait for the children
2. 1st child forks again and exit
   - providing an exit status for the parent
3. as soon as 1st child dies, 2nd child will be re-parented to init

Double fork (cont.)

We want to write a program that fork a child and wants to both avoid waiting for it and avoid zombies.

How to do that? Using the double fork technique.



Double fork — example

```
$ ./double−fork
$ 2nd child, parent pid = 1
```

- main parent returns immediately
- shell prompt arrives before second child's output
- 2 seconds pass before output from second child
- second child has been reparented to `init`