Programmation Systèmes
Cours 3 — Process Management

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

Laboratoire PPS, Université Paris Diderot

2012–2013

URL http://upsilon.cc/zack/teaching/1213/progsyst/
Copyright © 2011–2012 Stefano Zacchiroli
License Creative Commons Attribution-ShareAlike 3.0 Unported License
http://creativecommons.org/licenses/by-sa/3.0/

Outline

1. Fork inheritance
2. External program execution
3. Simple UNIX architectures
4. 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 buffered, executing it before forking ensure 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.

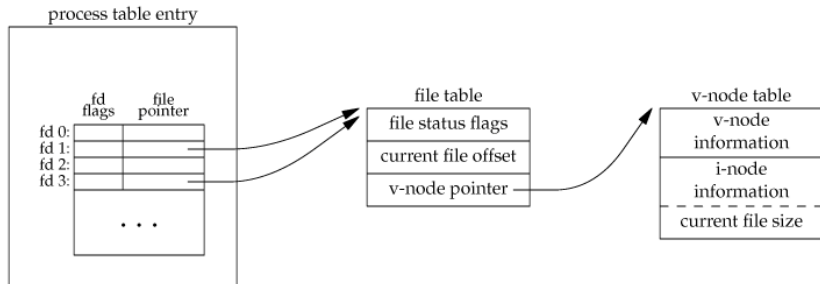
> Upon fork all file descriptors of the parent are duplicated into the the child.

- 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);
```

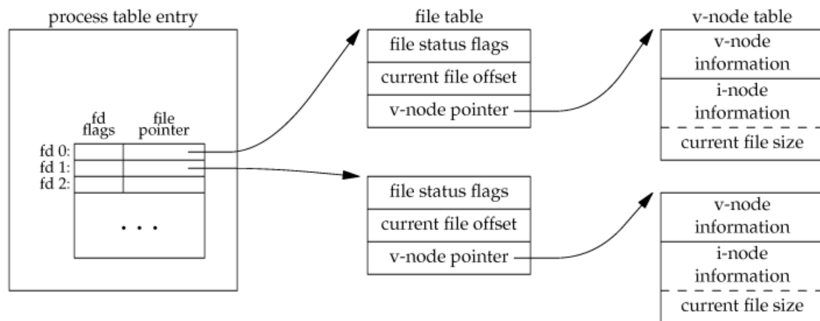
Returns: *new file descriptor if OK, -1 on error*



APUE, Figure 3.8

sample situation after dup(STDIN_FILENO);

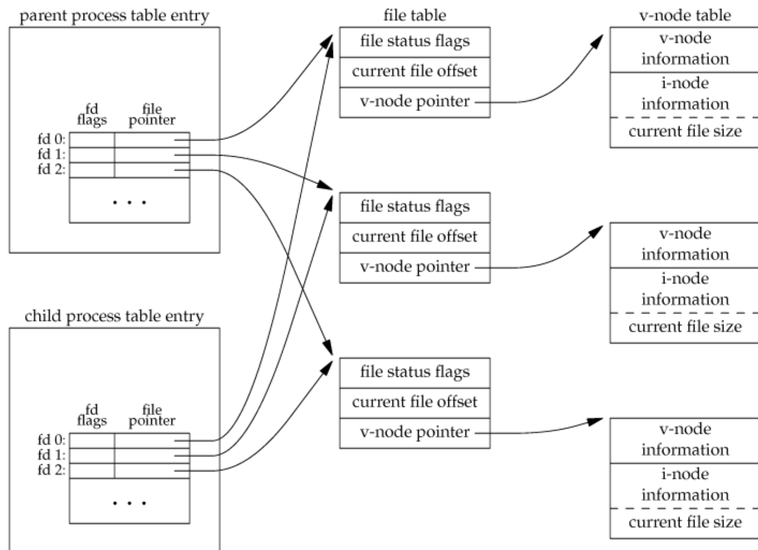
File sharing — before fork



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 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
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:[1]

- 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

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

Distinguishing traits after fork

On the other hand, differences remain among parent and child:[2]

- 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)

[2] 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. Fork inheritance
2. External program execution
3. Simple UNIX architectures
4. 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 the exec syscall to run 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 invoking process with a new one, obtained reading a program 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 write “exec” for short, but there are 6 different exec:

```
#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 process.

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 as *vectors* of pointers.
   - list of pointers shall be given including a trailing NULL
   - as, no wonder, it happens for `argv` in `main()` calling convention
   - note: the calling convention allow to have basename(*path*) $\neq$ `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.

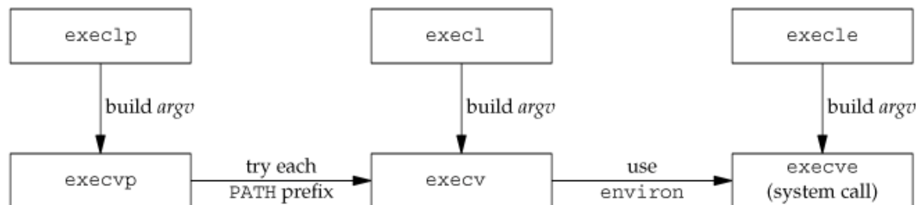
(. . . but why section 3?)

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

```
#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);
}
```

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);
}
```

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 `system` function is provided:

```
#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 system function is provided:

```
#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

```c
#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

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

int system(const char *cmd) {
	pid_t pid;
	int status;
	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 ((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 — implementation (*buggy*)

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

int system(const char *cmd) {
        pid_t pid;
        int status;
        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 `waitid` 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

```
#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 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. Fork inheritance
2. External program execution
3. Simple UNIX architectures
4. 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 (e.g. to drop privileges) between fork and exec, e.g.:
   - set up redirections
   - change user IDs
   - change signal masks
   - set up "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 programmers, especially—but not only—when source code is available

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, arguments, environment
  - protocol design is not an issue
- data are passed through 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 relies on 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, 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);
}
```

Shelling out — example

Demo

Race conditions

Shelling out, the risk of unwanted interference among parent and child processes is almost non-existent.[3] 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.

[3] 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.

`sleep` mitigates but does not solve the problem

- 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

```c
#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);
}
```

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 and avoid 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:[4]

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.

[4] 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.)

```c
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

```
#include <unistd.h>

pid_t vfork(void);
```

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

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. 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, as the parent is 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

```c
#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. Fork inheritance
2. External program execution
3. Simple UNIX architectures
4. 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?

```c
#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.

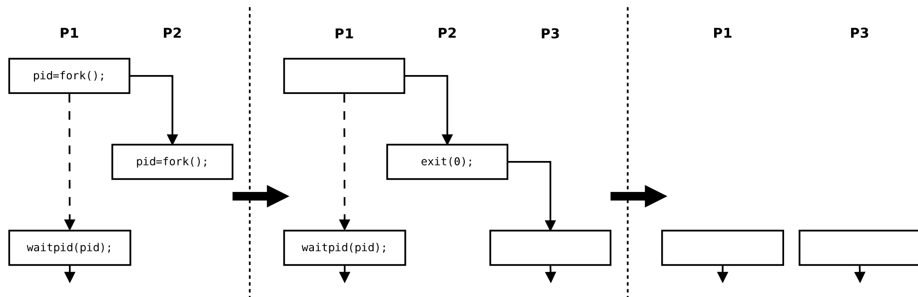
Double fork

1. parent `fork`s a 1st child and `wait` for the children
2. 1st child `fork`s 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`