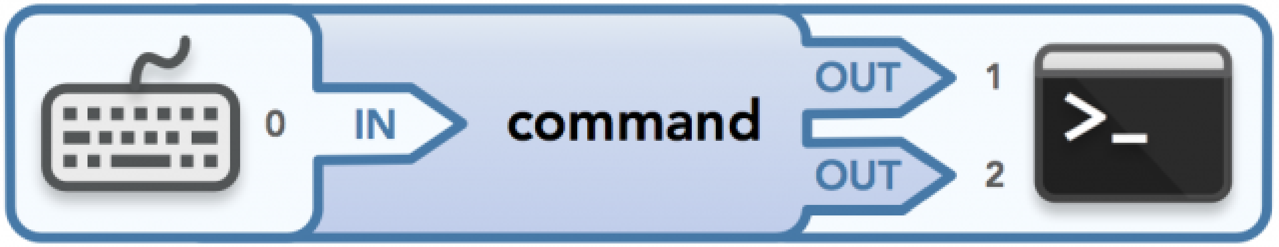
你所不知道的C語言: Stream I/O, EOF 和例外處理

直播錄影

資料處理流程



data flow for the standard input (0), output (1), and error (2) streams
[source]

在 UNIX 的設計哲學是 “Everything is a file”，扣除少數的例外 (如 BSD socket)，任何資源都可當作檔案來操作，不過當我們查閱 C 語言手冊和文獻時，“file” 的操作被分類在 stream I/O 中，聽起來有點奇怪吧？而 stream I/O 是 C 語言心裡最軟的一塊，這話怎說？

我們先來看 `<ctype.h>` 標頭檔宣告的 islower() 函式，其作用很單純，就是判定輸入的「字元」是否為英文小寫字元，不過在宣告中，islower() 函式的參數卻是 int 型態，類似的現象也出現於 getchar() 函式的回傳值，也是 int 型態。在現行 64 位元處理器架構上運作的作業系統常用 32-bit 表

示 int 型態，甚至某些作業系統用 64-bit 表示 int 型態，讓我們不禁納悶：「C 語言不是很講究效率嗎？能用 1 個 byte (char) 處理的資料，為何要用 4 個 bytes 甚至更多 (int) 去表達呢？」

簡單來說，要考慮到 EOF (end-of-file)！後者跟編譯器與執行環境有關，上述的 islower() 或 getchar() 之所以將原本 char 型態可保存的資料「擴充」為 int 型態，即是考慮到輸入和輸出的過程可能被某種情況或條件給「打斷」，倘若要深入理解更多， 就得回到 stream I/O 的原理。此外，從 C89, C99 規格就存在的 signal，依據手冊的說法，這是 “ANSI C signal handling”，實務上會在 Stream I/O 使用中作為例外處理機制，本議程預計分析現有開放原始碼專案中的情境，探討 Stream I/O, EOF 和 signal 之間的微妙關聯。

> 延伸閱讀: Redirection in bash

有沒有 C++ 標準幫 Apple 背書的八卦

C++ 標準函式庫竟然有 ios::good

「iOS 真棒！」

EOF 的發生往往不是單一程式的事

C99 規格書 7.19.6.2 談及 `fscanf` 一類函式的行為, 第 5 段:

> “A directive composed of white-space character(s) is executed by reading input up to the first non-white-space character (which remains unread), or until no more characters can be read.”

end-of-file 的設定是由輸入函式在讀取資料的過程中設定的。

搜尋 glibc 原始程式碼，可發現 `EOF` 定義在 `libio/stdio.h`，但這是 `Define ISO C stdio on top of C++ iostreams`

```
/* The value returned by fgetc and similar
   functions to indicate the end of the file.  */
#define EOF (-1)
```

`ctype.h` 實作於 glibc 原始程式碼的 `ctype/ctype.h`

這裡要補充一點, 在有些系統 (特別是 UNIX 系統) 上, 每一行資料 (這裡只指 text 資料) 必須以 end-of-line 來分割, 包括最後一行. 否則, 某些程式語言在, 或系統處理最後一行時有可能會出問題。

```
〔資料〕<EOL>
. . .
〔資料〕<EOF> <-- 有可能會造成問題
```

最好是這樣:

```
〔資料〕<EOL>
. . .
```

```
〔資料〕<EOL>
<EOF>
```

以 ctrl-Z 做文字檔的 end-of-file marker 源自 DEC 早期的系統, 後來被 CP/M 借用, 再後來用在 MS-DOS 上。

使用 ctrl-Z 的原因是因為在早期的檔案系統，檔案長度是以 128-byte 的 sector 為單位的, 當檔案的大小不是 sector 的整數倍數的話, ctrl-Z 用來標示檔案的真正結尾, 並以 ctrl-Z 把剩餘的 sector 填滿。

自 MS-DOS 2.0 起就可以正確的記錄檔案的正確長度, 已可不需要用 ctrl-Z 的機制了. 但這個機制到目前還在支援. 因為在某個情況下它非常有用。

當你在 console 上用鍵盤來輸入資料時, 可以用 ctrl-Z 來產生 end-of-file 訊息。

```
〔執行〕
1 2 3 4
5 6 7 8^Z
you entered 8 numbers.
```

UNIX 系統上用 ctrl-D 來 signal end-of-file.

以目前的系統來說, EOF 只是一個概念上的存在, 文字檔並不需要這個 marker，文字編輯器不會自動的加入這個 marker。

在 MS-Windows 上, ctrl-Z 不再做為檔案大小的依據, 也不會導致 file truncation. Ctrl-Z 可以存在文字檔內, 但它會影響輸入函式的行為: 如果檔案是以 text mode 來開啟的話, ctrl-Z 會終結輸入，即使 ctrl-Z 後面還有資料也是一樣。

Standard I/O
C++ iostream 的 `cin` , `cout` 和 C stdio 的 `scanf` , `printf` 效能比較

signal

```
#include <signal.h>

void (*signal(int sig, void (*func)(int)))(int);
```

或者透過 typedef 得到的等效且簡短的宣告:

```
typedef void (*sig_t) (int);
sig_t signal(int sig, sig_t func);
```

C Notes for Professionals
Chapter 31 (對應 Page 208)

signal numbers can be synchronous (like SIGSEGV– segmentation fault) when they are triggered by amalfunctioning of the program itself or asynchronous (like SIGINT - interactive attention) when they are initiatedfrom outside the program, e.g by a keypress as Cntrl-C

合法的 C11 程式:

```
#include <signal.h> /* signal() */
#include <stdio.h>  /* printf() */
#include <stdlib.h> /* abort()  */

void handler_nonportable(int sig)
{
    /* undefined behavior, maybe fine on specific platform */
    printf("Catched: %d\n", sig);

```

```
        /* abort is safe to call */
        abort();
    }

    sig_atomic_t volatile finished = 0;

    void handler(int sig)
    {
        switch (sig) {
            /* hardware interrupts should not return */
        case SIGSEGV:
        case SIGFPE:
        case SIGILL:
            /* quick_exit is safe to call */
            quick_exit(EXIT_FAILURE);

        default:
            /* Reset the signal to the default handler,
             * so we will not e called again if things go
             * wrong on return.
             */
            signal(sig, SIG_DFL);

            /* let everybody know that we are finished */
            finished = sig;
            return;
        }
    }
    int main(void)
    {
        /* Catch the SIGSEGV signal, raised on segmentation faults
         * (i.e NULL ptr access)
         */
        if (signal(SIGSEGV, &handler) == SIG_ERR) {
            perror("could not establish handler for SIGSEGV");
            return EXIT_FAILURE;
        }

        /* Catch the SIGTERM signal, termination request */
        if (signal(SIGTERM, &handler) == SIG_ERR) {
            perror("could not establish handler for SIGTERM");
            return EXIT_FAILURE;
        }

        /* Ignore the SIGINT signal, by setting the handler to `SIG_IGN`. */
        signal(SIGINT, SIG_IGN);

        /* Do something that takes some time here, and leaves
         * the time to terminate the program from the keyboard.
         */

        /* Then: */
        if (finished) {
            fprintf(stderr, "we have been terminated by signal %d\n",
                    (int) finished);
            return EXIT_FAILURE;
        }

        /* Try to force a segmentation fault, and raise a SIGSEGV */
        {
            char *ptr = 0;
            *ptr = 0;
        }

        /* This should never be executed */
        return EXIT_SUCCESS;
    }
```

注意 quick_exit 在 C11 才出現，在 C99 可改寫為:

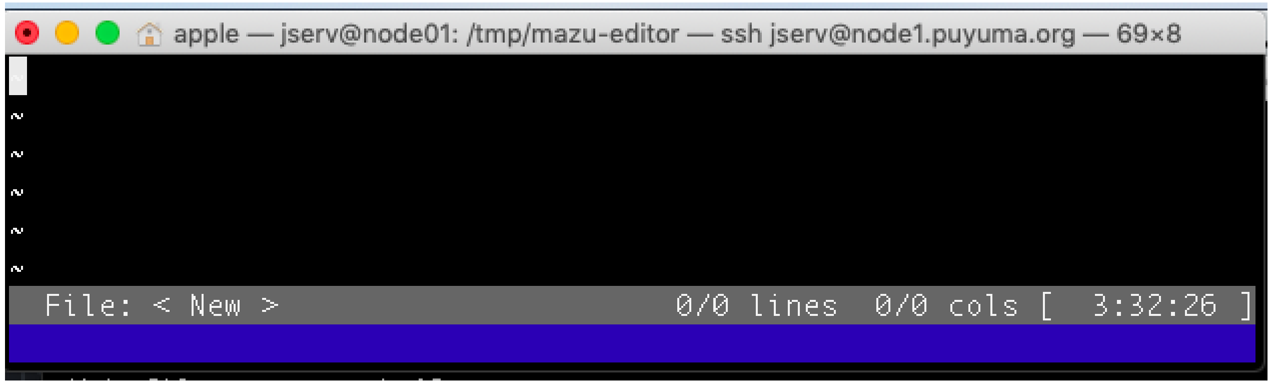
```
_exit(EXIT_FAILURE);
```

POSIX recommends the usage of sigaction() instead of signal(), due to its underspecified behavior and significant implementation variations. POSIX also defines many more signals than ISO C standard, including SIGUSR1 and SIGUSR2, which can be used freely by the programmer for any purpose.

signal 的存在並非總是「善後」的行為，而是藉由其同步和非同步的處理，讓程式設計者能先專注在 business logic，再來思考例外狀況該如何回應。以 mazu-editor 媽祖程式碼編輯器 為例:

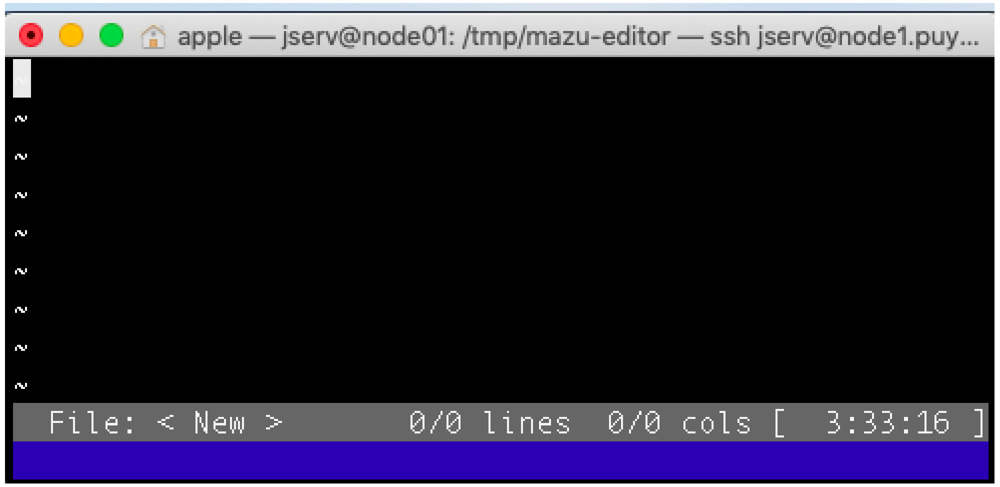
```
void init_editor() {
    signal(SIGWINCH, handle_sigwinch);
}

void handle_sigwinch() {
    update_window_size();
    if (ec.cursor_y > ec.screen_rows)
        ec.cursor_y = ec.screen_rows - 1;
    if (ec.cursor_x > ec.screen_cols)
        ec.cursor_x = ec.screen_cols - 1;
    refresh_screen();
}
```

當使用者嘗試調整終端機模擬程式的有效寬度和高度，從原本:


apple — jserv@node01: /tmp/mazu-editor — ssh jserv@node1.puyuma.org — 69×8

```
~
~
~
~
~
 File: < New >                        0/0 lines   0/0 cols [  3:32:26 ]
```

到後來:

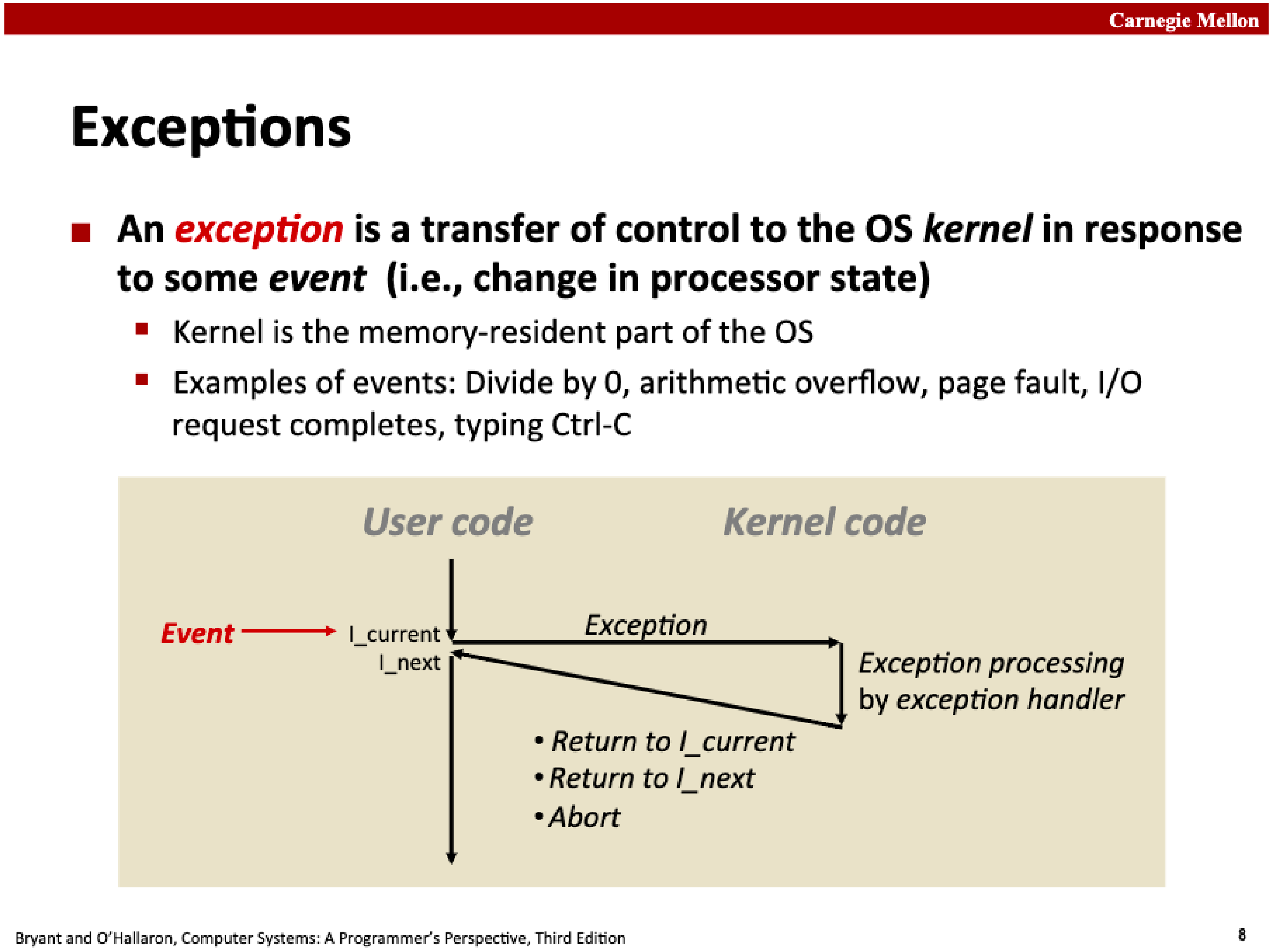

apple — jserv@node01: /tmp/mazu-editor — ssh jserv@node1.puy...

```
~
~
~
~
~
~
~
~
 File: < New >           0/0 lines   0/0 cols [  3:33:16 ]
```

就涉及到刷新畫面的操作，但若程式都要週期性更新，這樣不足以反映操作，於是可藉由 `SIGWINCH` (Window size change) signal 及其註冊的 handler 來實作更新。

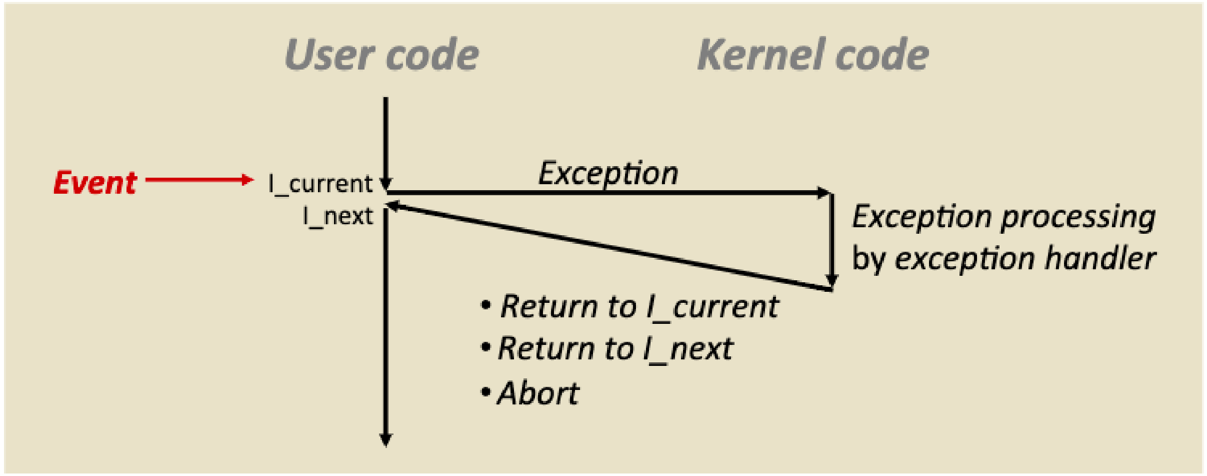
回顧 CS:APP 第 8 章

- [ ] Exceptional Control Flow: Exceptions and Processes



Exceptions

- An ***exception*** **is a transfer of control to the OS** ***kernel*** **in response to some** ***event*** **(i.e., change in processor state)**
  - Kernel is the memory-resident part of the OS
  - Examples of events: Divide by 0, arithmetic overflow, page fault, I/O request completes, typing Ctrl-C



- Error Handling in C programs
  - `errno`
  - `perror`, `strerror`
  - `EXIT_SUCCESS`, `EXIT_FAILURE`

```
int dividend = 50;
int divisor = 0;
int quotient;

quotient = (dividend/divisor); /* This will produce a runtime error! */
```

觸發 SIGFPE signal

根據 IEEE 754 7.2 節的說明，會產生 quiet NAN 的運算如下：

1. any general-computational or signaling-computational operation on a signaling NaN (see 6.2), except for some conversions (see 5.12)
2. multiplication: multiplication(0, ∞) or multiplication(∞, 0) fusedMultiplyAdd: fusedMultiplyAdd(0, ∞, c) or fusedMultiplyAdd(∞, 0, c) unless c is a quiet NaN; if c is a quiet NaN then it is implementation defined whether the invalid operation exception is signaled
3. addition or subtraction or fusedMultiplyAdd: magnitude subtraction of infinities, such as: addition(+∞, −∞)
4. division: division(0, 0) or division(∞, ∞)
5. remainder: remainder(x, y), when y is zero or x is infinite and neither is NaN
6. squareRoot if the operand is less than zero
7. quantize when the result does not fit in the destination format or when one operand is finite and the other is infinite

會產生 signal NAN 的運算如下：

1. conversion of a floating-point number to an integer format, when the source is NaN, infinity, or a value that would convert to an integer outside the range of the result format under the applicable rounding attribute

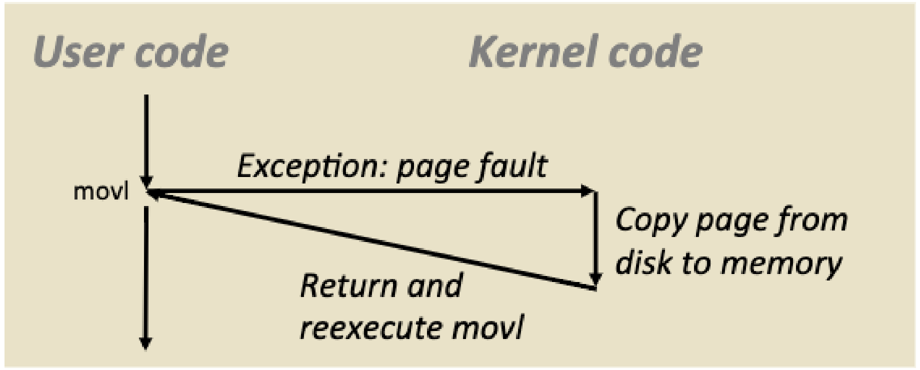
2. comparison by way of unordered-signaling predicates listed in Table 5.2, when the operands are unordered
3. logB(NaN), logB(∞), or logB(0) when logBFormat is an integer format (see 5.3.3).

Fault Example: Page Fault

- User writes to memory location
- That portion (page) of user's memory is currently on disk

```
int a[1000];
main ()
{
    a[500] = 13;
}
```

```
 80483b7:        c7 05 10 9d 04 08 0d   movl   $0xd,0x8049d10
```

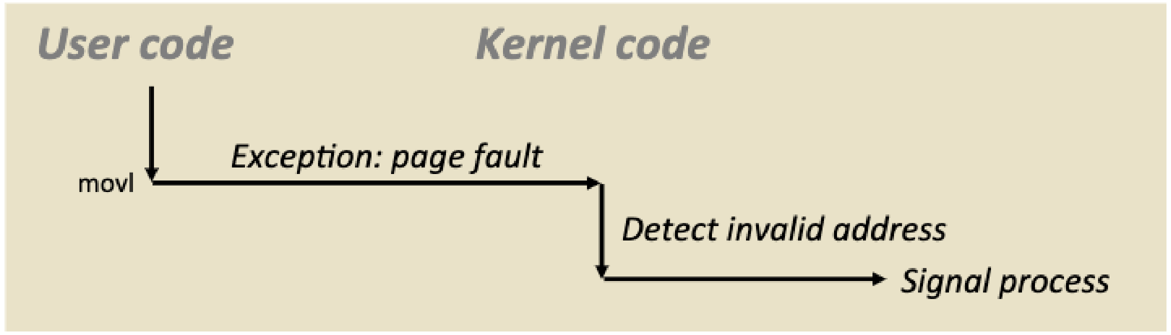


Fault Example: Invalid Memory Reference

```
int a[1000];
```

```
main ()
{
    a[5000] = 13;
}
```

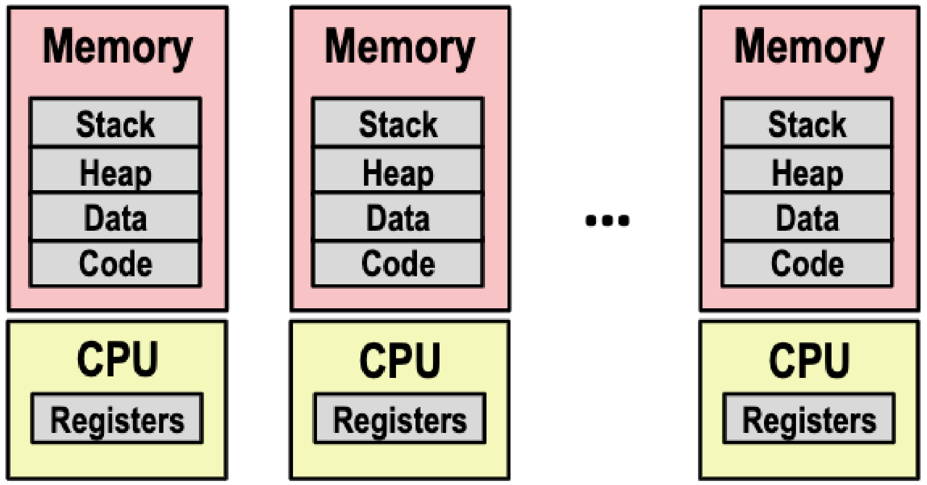
```
 80483b7:        c7 05 60 e3 04 08 0d   movl    $0xd,0x804e360
```



- Sends **SIGSEGV** signal to user process
- User process exits with “segmentation fault”

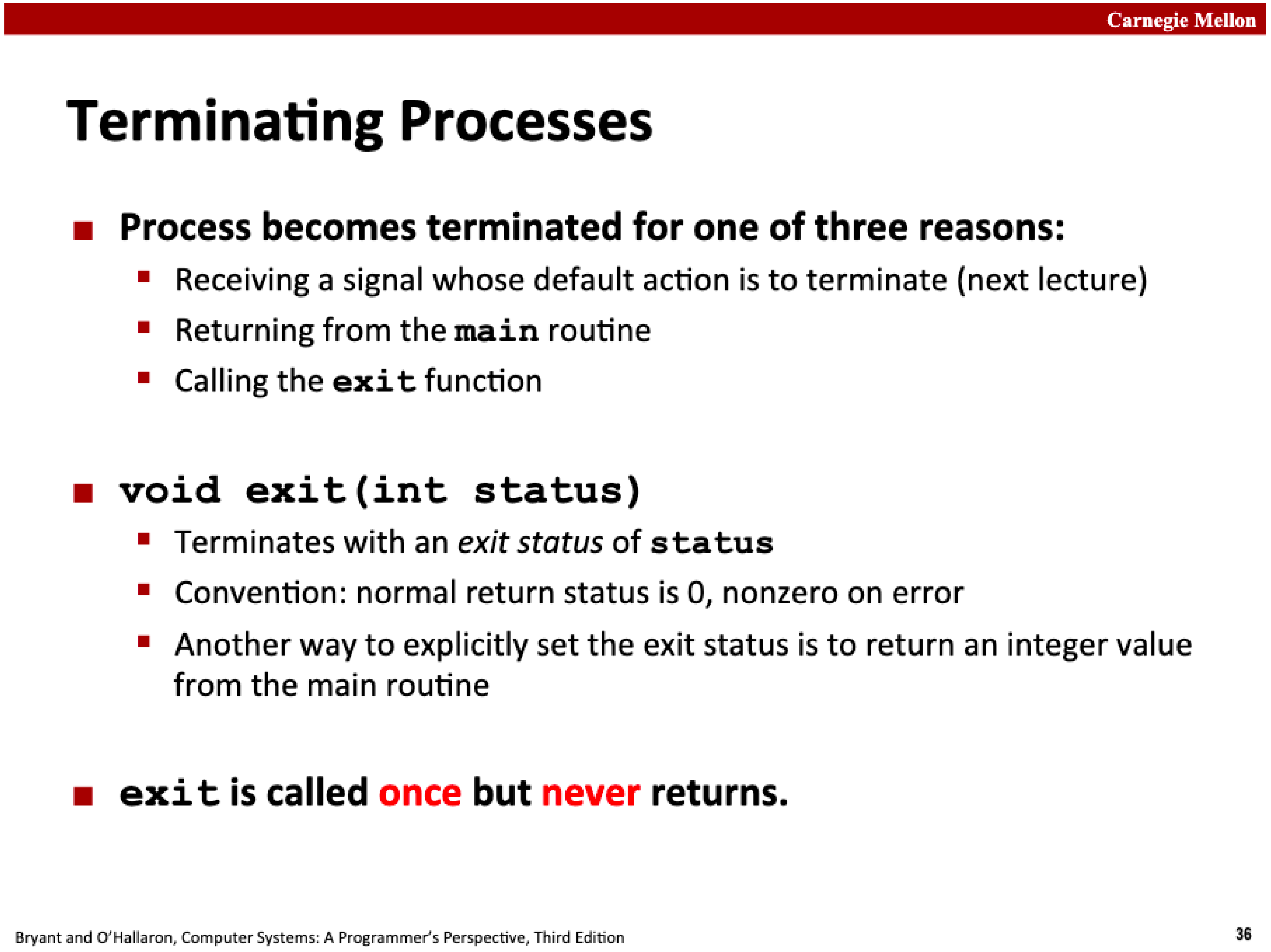
Page fault

Multiprocessing: The Illusion



- **Computer runs many processes simultaneously**
  - Applications for one or more users
    - Web browsers, email clients, editors, …
  - Background tasks
    - Monitoring network & I/O devices

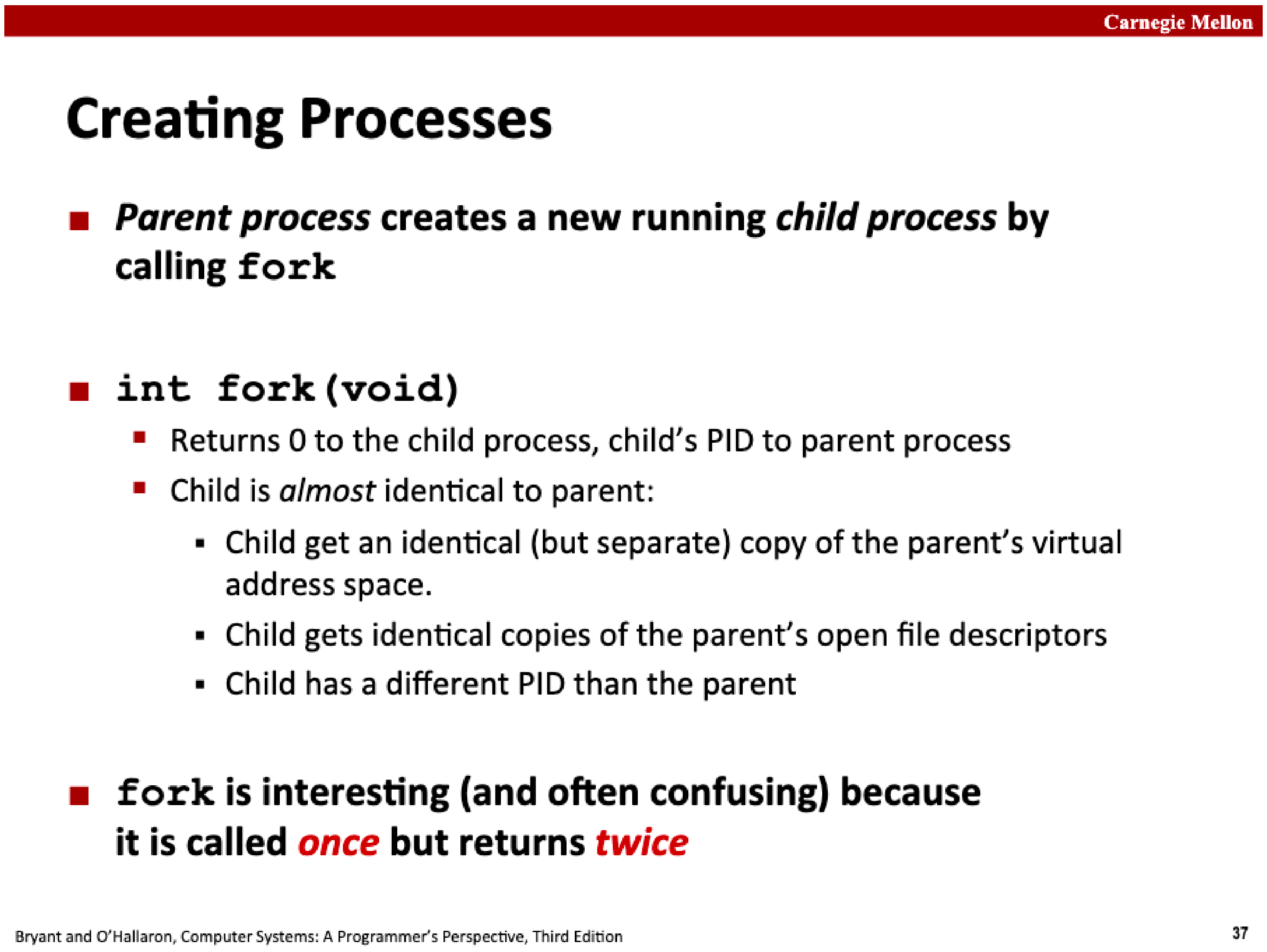
> "Time is an illusion. Lunchtime doubly so."
> — Douglas Adams, The Hitchhiker's Guide to the Galaxy



Terminating Processes

- **Process becomes terminated for one of three reasons:**
  - Receiving a signal whose default action is to terminate (next lecture)
  - Returning from the **`main`** routine
  - Calling the **`exit`** function
- **`void exit(int status)`**
  - Terminates with an *exit status* of **`status`**
  - Convention: normal return status is 0, nonzero on error
  - Another way to explicitly set the exit status is to return an integer value from the main routine
- **`exit` is called once but never returns.**

- POSIX 中，exit 沒有返回值
- execve 在成功執行時沒有返回值



Creating Processes

- *Parent process* creates a new running *child process* by calling `fork`

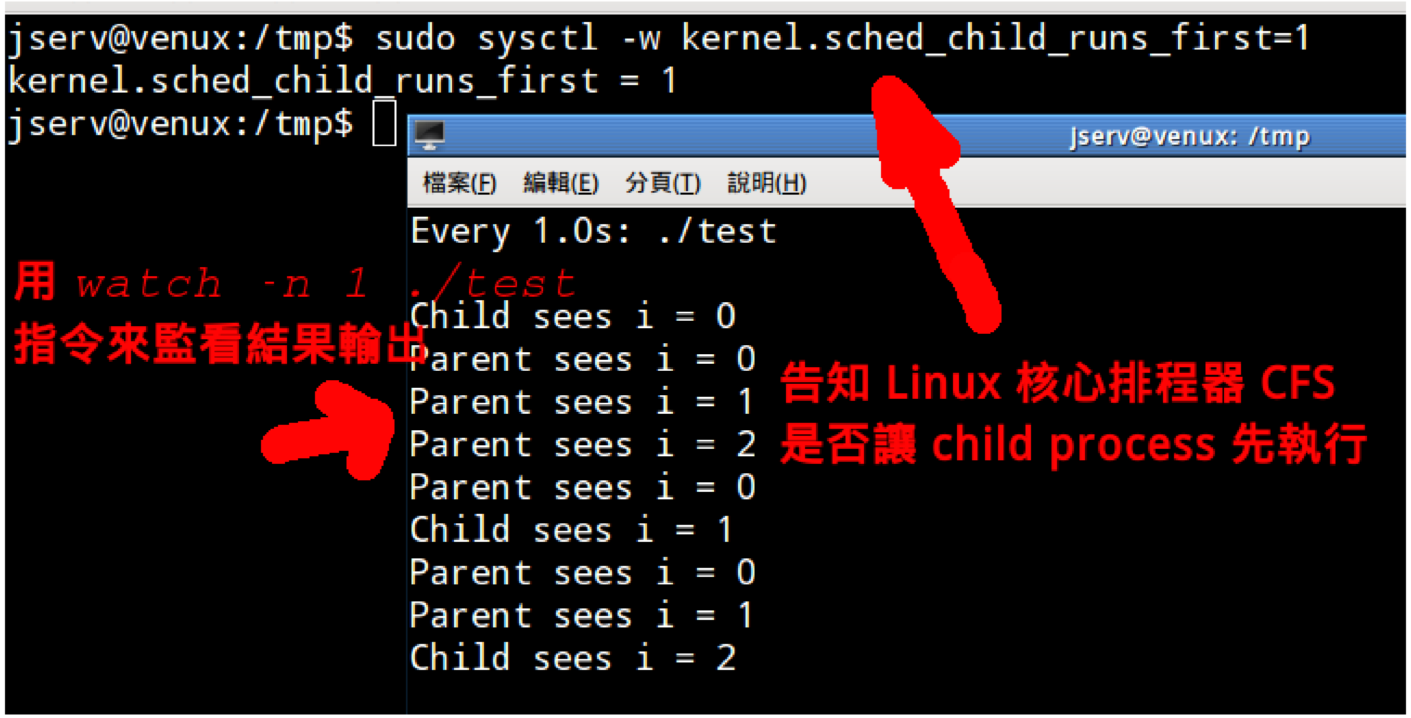
- `int fork(void)`
  - Returns 0 to the child process, child's PID to parent process
  - Child is *almost* identical to parent:
    - Child get an identical (but separate) copy of the parent's virtual address space.
    - Child gets identical copies of the parent's open file descriptors
    - Child has a different PID than the parent

- **`fork` is interesting (and often confusing) because it is called *once* but returns *twice***

- fork，呼叫 1 次，返回 2 次
- 在 1969 年開發的 UNIX 第一版就提供 fork 系統呼叫，至今恰好滿 50 年，當初提出有其時空意義 (存在 Multics 的影子)，但 fork 在現代計算機架構上運作的類似 UNIX 作業系統 (如 Linux 和 FreeBSD)，實作有其彆扭之處，例如經典的 fork+exec 的「最佳化」(透過 lazy fork)，於是來自

Microsoft Research, Boston University, ETH Zurich 等單位的研究人員，提交論文 A fork() in the road，回顧這 50 年間 fork 系統呼叫的發展和探討其侷限，建議人們改用 posix_spawn (這是 POSIX.1-2001 規範的一部分) 並徹底捨棄 fork 的使用

- 一個fork的面試題
  - 假設 fork.c 編譯出來的結果是 `fork`，則可用這個指令計算 '-' 數量: `./fork | wc -c`
- In fork() which will run first, parent or child?
  - `sudo sysctl -w kernel.sched_child_runs_first=1`
- 在 GNU/Linux 上測試：事先開啟兩個虛擬終端機視窗，如下圖



```
jserv@venux:/tmp$ sudo sysctl -w kernel.sched_child_runs_first=1
kernel.sched_child_runs_first = 1
jserv@venux:/tmp$
```

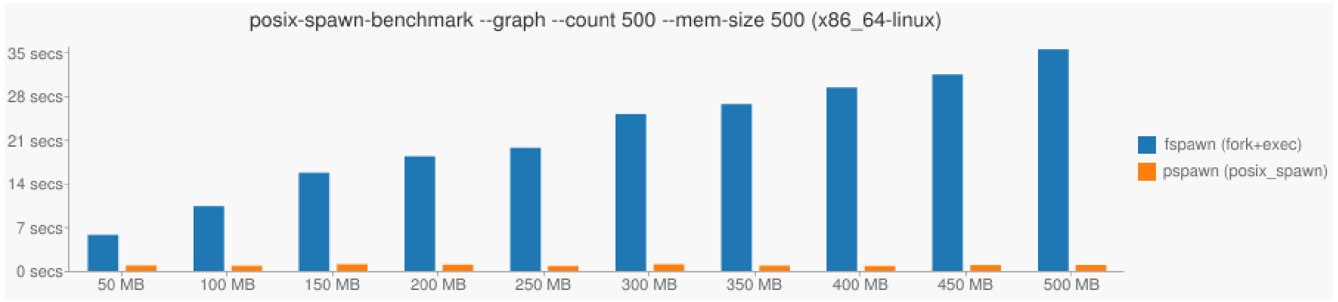
```
Every 1.0s: ./test

Child sees i = 0
Parent sees i = 0
Parent sees i = 1
Parent sees i = 2
Parent sees i = 0
Child sees i = 1
Parent sees i = 0
Parent sees i = 1
Child sees i = 2
```

jserv@venux: /tmp
檔案(F) 編輯(E) 分頁(T) 說明(H)
用 watch -n 1 ./test
指令來監看結果輸出
告知 Linux 核心排程器 CFS
是否讓 child process 先執行

  - 右邊是測試程式 (test.c)，使用 `watch -n 1 ./test` 每秒更新執行結果一次。耐心凝視結果一陣子，是否發現輸出結果會跳動？
  - 使用左邊執行 `sudo sysctl -w kernel.sched_child_runs_first=1`，以便要求 Linux 排程器 (CFS) 讓 child process 優先於 parent process 執行

- How a fix in Go 1.9 sped up our Gitaly service by 30x
    - Gitlab 在運作過程中發現其 RPC 處理程序 Gitaly 的 latency 日漸增加，CPU 的使用率也顯著提高, 原本工程團隊以為是 resource leaking 問題，然而透過 pprof 與 cAdivisor(cgroup 分析工具) 分析發現並沒有 leak 的現象, 最後追蹤到了 SIGABRT thread dump 中發現問題在於系統呼叫 ForkLock 而該問題指向了 clone() 的方式, 而這問題在於 Go 1.8 中使用的 fork 方式會複製 parent process memory space, 因此當系統逐漸增大後 fork cost 就會變高, 而在 Go 1.9 後便改採用 posix_spawn 來避免此問題
- posix-spawn

> fork(2) calls slow down as the parent process uses more memory due to the need to copy page tables. In many common uses of fork(), where it is followed by one of the exec family of functions to spawn child processes (Kernel#system, IO::popen, Process::spawn, etc.), it's possible to remove this overhead by using special process spawning interfaces (posix_spawn(), vfork(), etc.)



- [ ] Exceptional Control Flow: Signals and Nonlocal Jumps

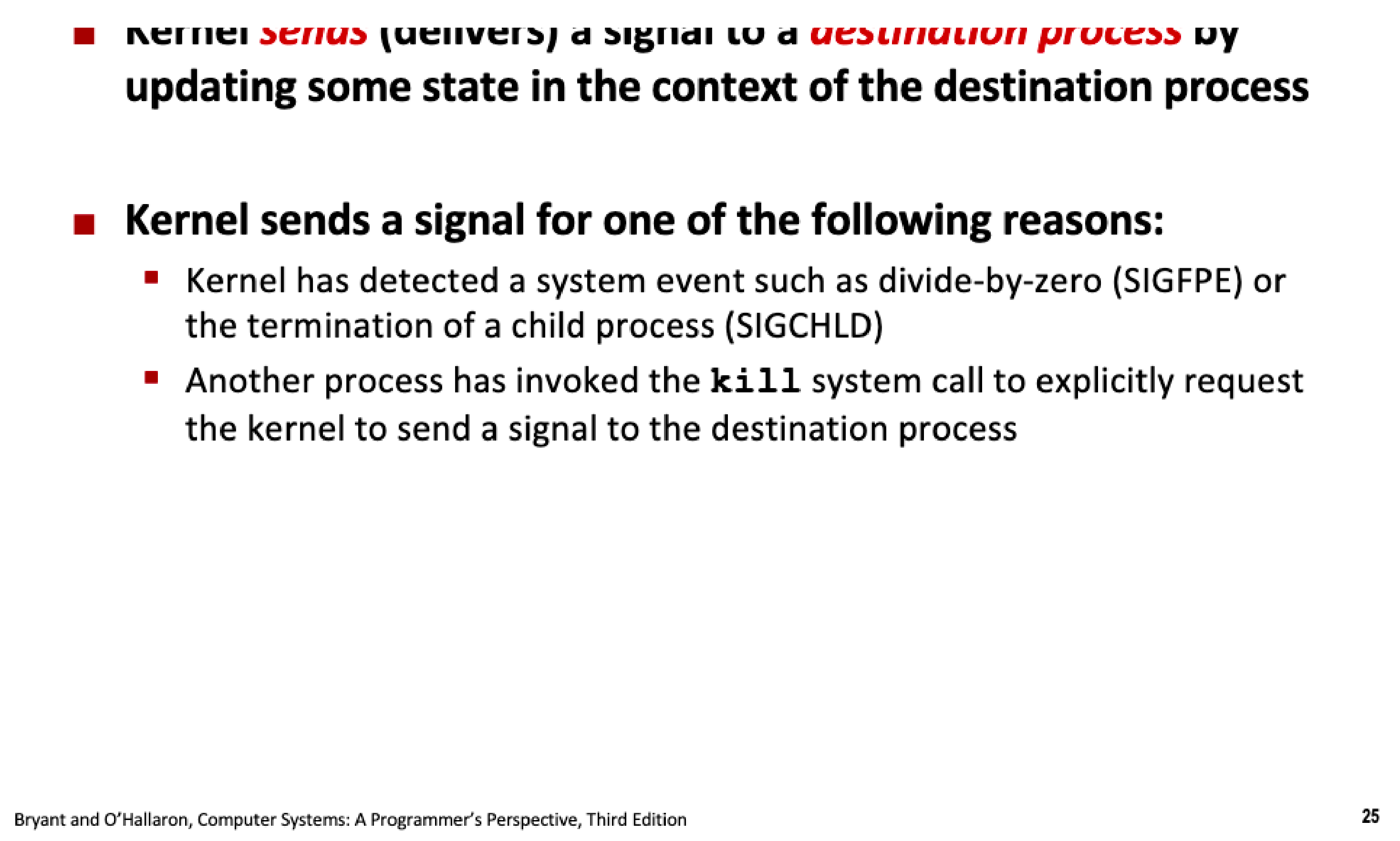
Signals

- **A *signal* is a small message that notifies a process that an event of some type has occurred in the system**
  - Akin to exceptions and interrupts
  - Sent from the kernel (sometimes at the request of another process) to a process
  - Signal type is identified by small integer ID's (1-30)
  - Only information in a signal is its ID and the fact that it arrived

ID	*Name*	*Default Action*	*Corresponding Event*
2	SIGINT	Terminate	User typed ctrl-c
9	SIGKILL	Terminate	Kill program (cannot override or ignore)
11	SIGSEGV	Terminate	Segmentation violation
14	SIGALRM	Terminate	Timer signal
17	SIGCHLD	Ignore	Child stopped or terminated

Signal Concepts: Sending a Signal

- **Kernel *sends* (delivers) a signal to a *destination process* by**



- **Kernel *sends* (delivers) a signal to a *destination process* by updating some state in the context of the destination process**
- **Kernel sends a signal for one of the following reasons:**
  - Kernel has detected a system event such as divide-by-zero (SIGFPE) or the termination of a child process (SIGCHLD)
  - Another process has invoked the **`kill`** system call to explicitly request the kernel to send a signal to the destination process

待整理

- Tearing apart printf()
- Your terminal is not a terminal: An Introduction to Streams
- The TTY demystified