

Systems 3

Processes

Marcel Waldvogel

Department of Computer and Information Science
University of Konstanz

Winter 2019/2020

Chapter Goals

- How is the CPU usage split among processes?
- How do the instruction sequences differ, when seen by CPU vs. seen by the process?
- What is the lifecycle of a process?
- What are the process states? When and how do transitions between them happen?
- How is a process created and terminated?

- How do signals work?
- Asynchronous notification with signals?

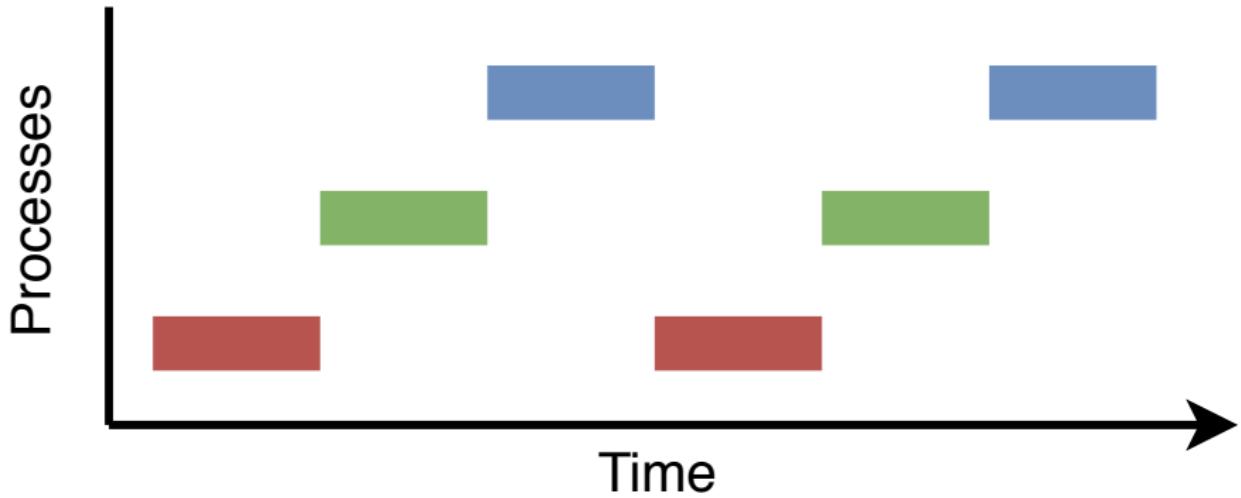




A close-up view of a green first aid kit. The cover is made of a ribbed green fabric with a large white cross and the words "FIRST AID" printed in white. Two tan-colored straps with metal buckles are attached to the sides. The kit is resting on a dark surface.

FIRST AID

Multiprogramming



Process Creation

Principal events that cause processes to be created:

- 1 System initialization
- 2 **Execution of a process creation system call by a running process**
- 3 A user request to create a new process
- 4 Initiation of a batch job

Process Hierarchies

```
1 klaus 16437 2306 0 Nov07 ?          01:11:58 \_ /usr/bin/tilix
2 klaus 28498 16437 0 Nov11 pts/2      00:00:00 |  \_ /bin/bash
3 klaus 2113 16437 0 Nov11 pts/4       00:00:00 |  \_ /bin/bash
4 klaus 2323 16437 0 Nov11 pts/5       00:00:00 |  \_ /bin/bash
5 klaus 5978 16437 0 Nov11 pts/6       00:00:00 |  \_ /bin/bash
6 klaus 25024 16437 0 Nov12 pts/8      00:00:00 |  \_ /bin/bash
7 klaus 12380 16437 0 Nov14 pts/3      00:00:00 |  \_ /bin/bash
8 klaus 5420 12380 0 13:23 pts/3      00:00:00 |  |  \_ ps -ef --forest
```

Process Termination

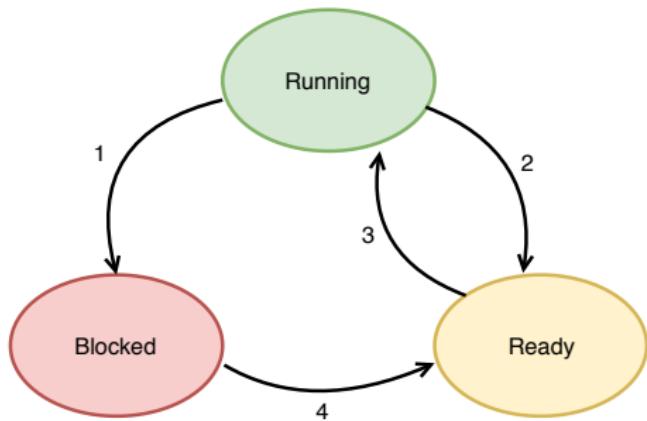
Conditions that cause a process to terminate:

- 1 Normal exit (voluntary)
- 2 Error exit (voluntary)
- 3 Fatal error (**involuntary**)
- 4 Killed by another process (**involuntary**)

Process States

Possible process states:

- 1 Process blocks for input
- 2 Scheduler picks another process
- 3 Scheduler picks this process
- 4 Input becomes available



Process Table

Process ID	Process Control Block or pointer to it
1	...
2	...
3	...
⋮	⋮

Process Control Block

Process Control Block

- Process ID
- Process State (Running, Ready, Blocked, ...)
- Memory assignment
- File table (open files)
- Accounting data/statistics
- Program Counter (when not Running)
- CPU registers (when not Running)
- Next Pointer (for PCB linked list)

Interrupts

Interrupt handling on the lowest level:

- 1 Hardware stacks program counter, etc.
- 2 Hardware loads new program counter from interrupt vector
- 3 Assembly language procedure saves registers
- 4 Assembly language procedure sets up new stack
- 5 C interrupt service runs
- 6 Scheduler marks waiting task as ready
- 7 Scheduler decides which process is to run next
- 8 C procedure returns to the assembly code
- 9 Assembly language procedure starts up new current process

Threads

Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

In PCB or CPU
In process memory

Creating a new Process

It takes two steps to run a program as a new process
(in Unix-like operating systems):

1 Fork:

creates a new process by duplicating the calling process. `fork(2)`

2 Exec:

replaces the current process image with a new process image. `exec(2)`

We will look at those steps in more detail on the next slides.

Fork

- 1 Check to see if process table is full
- 2 Try to allocate memory for the child's data and stack
- 3 Copy the parent's data and stack to the child's memory
- 4 Find a free process slot and copy parent's slot to it
- 5 Enter child's memory map in process table
- 6 Choose a PID for the child
- 7 Tell kernel and file system about child
- 8 Report child's memory map to kernel
- 9 Send reply messages to parent and child

Exec

- 1 Check permissions – is the file executable?
- 2 Read the header to get the segment and total sizes
- 3 Fetch the arguments and environment from the caller
- 4 Allocate new memory and release unneeded old memory
- 5 Copy stack to new memory image
- 6 Copy data (and possibly text) segment to new memory image
- 7 Check for and handle setuid, setgid bits
- 8 Fix up process table entry
- 9 Tell kernel that process is now runnable

Exit

exit() operations

- Close files
- Free memory and other resources
- Return exit code (and statistics) to parent process

High-level overview

- Preparation program code prepares for possible signal
- Response signal is received and action is taken
- Cleanup restore normal operation of process

Signals

Every signal has a name starting with SIG. You find a list in [signal\(7\)](#).

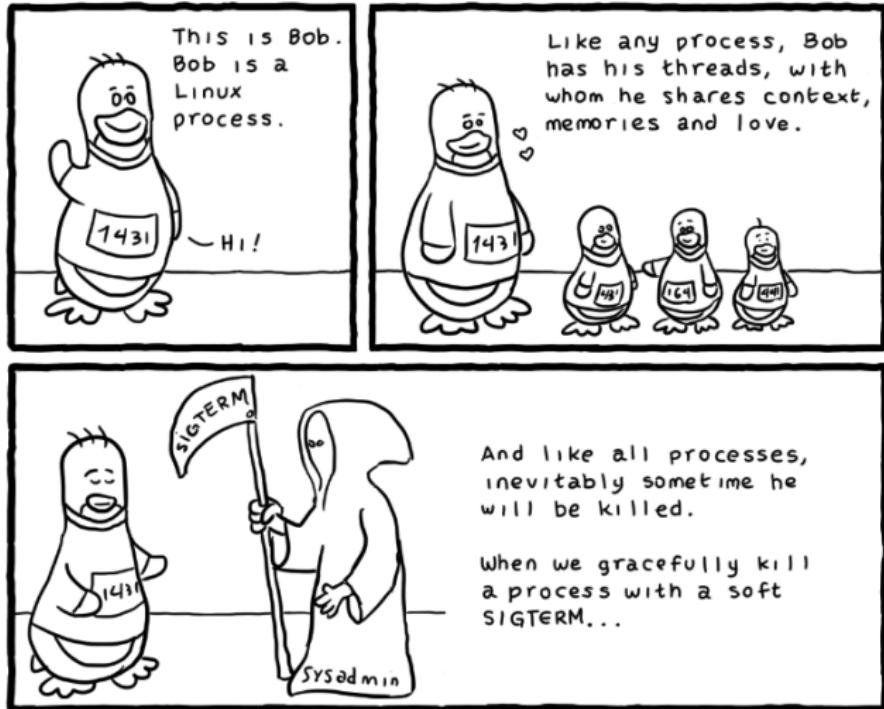
1	Signal	Standard	Action	Comment
2	<hr/>			
3	SIGABRT	P1990	Core	Abort signal from abort(3)
4	SIGALRM	P1990	Term	Timer signal from alarm(2)
5	SIGBUS	P2001	Core	Bus error (bad memory access)
6	SIGCHLD	P1990	Ign	Child stopped or terminated
7	SIGCLD	-	Ign	A synonym for SIGCHLD
8	SIGCONT	P1990	Cont	Continue if stopped
9	SIGEMT	-	Term	Emulator trap
10	SIGFPE	P1990	Core	Floating-point exception
11	SIGHUP	P1990	Term	Hangup detected on controlling terminal or death of controlling process
12	SIGILL	P1990	Core	Illegal Instruction
13	SIGINFO	-		A synonym for SIGPWR
14	SIGINT	P1990	Term	Interrupt from keyboard
15	SIGIO	-	Term	I/O now possible (4.2BSD)
16	SIGIOT	-	Core	IOT trap. A synonym for SIGABRT
17	SIGKILL	P1990	Term	Kill signal
18	...			

Signal handler in C

```
1 #include <stdio.h>
2 #include <signal.h>
3
4 void sigIntHandler(int sig_num)
5 {
6     signal(SIGINT, sigintHandler);
7     printf(" Cannot be terminated using Ctrl+C :-P ");
8     fflush(stdout);
9 }
10
11 int main ()
12 {
13     signal(SIGINT, sigIntHandler);
14     while (1) {
15         // infinite loop
16     }
17     return 0;
18 }
```

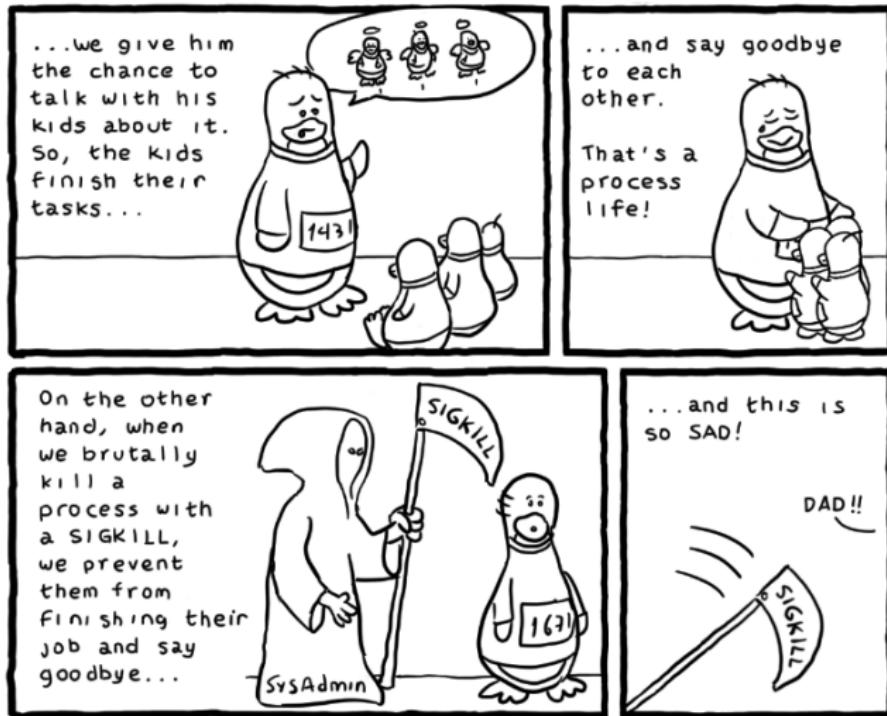
This is wasteful. Why? Busy waiting; better use e.g. `sleep(2)` or `pause(2)`

Killing me softly with this SIG...



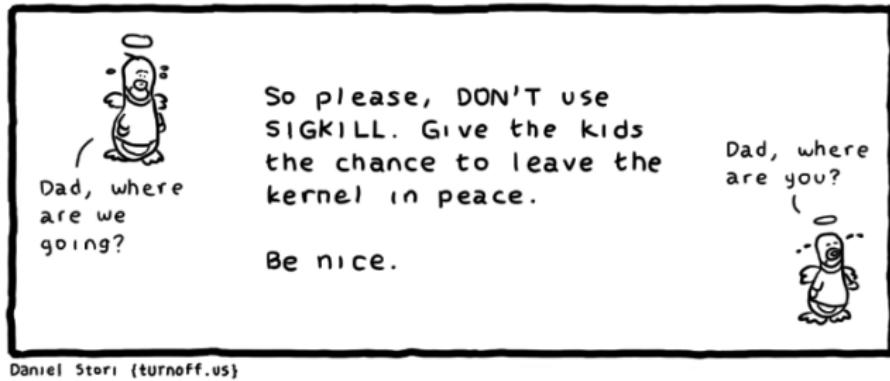
Source: <https://turnoff.us/geek/dont-sigkill/>

Killing me softly with this SIG...



Source: <https://turnoff.us/geek/dont-sigkill/>

Killing me softly with this SIG...



Source: <https://turnoff.us/geek/dont-sigkill/>

System calls related to signals

System call	Purpose
<code>sigaction(2)</code>	Modify response to future signal
<code>kill(2)</code>	Send signal to another process
<code>alarm(2)</code>	Send ALRM signal to self after delay
<code>pause(2)</code>	Suspend self until future signal
<code>sigprocmask(2)</code>	Change set of blocked signals
<code>sigsuspend(2)</code>	Atomic <code>sigprocmask(2)</code> , then <code>pause(2)</code>
<code>sigpending(2)</code>	Examine set of pending (blocked) signals
<code>sigreturn(2)</code>	Clean up after signal handler