Systems 3

Processes

Marcel Waldvogel

(Handout)

Department of Computer and Information Science
University of Konstanz

Winter 2019/2020

Chapter Goals

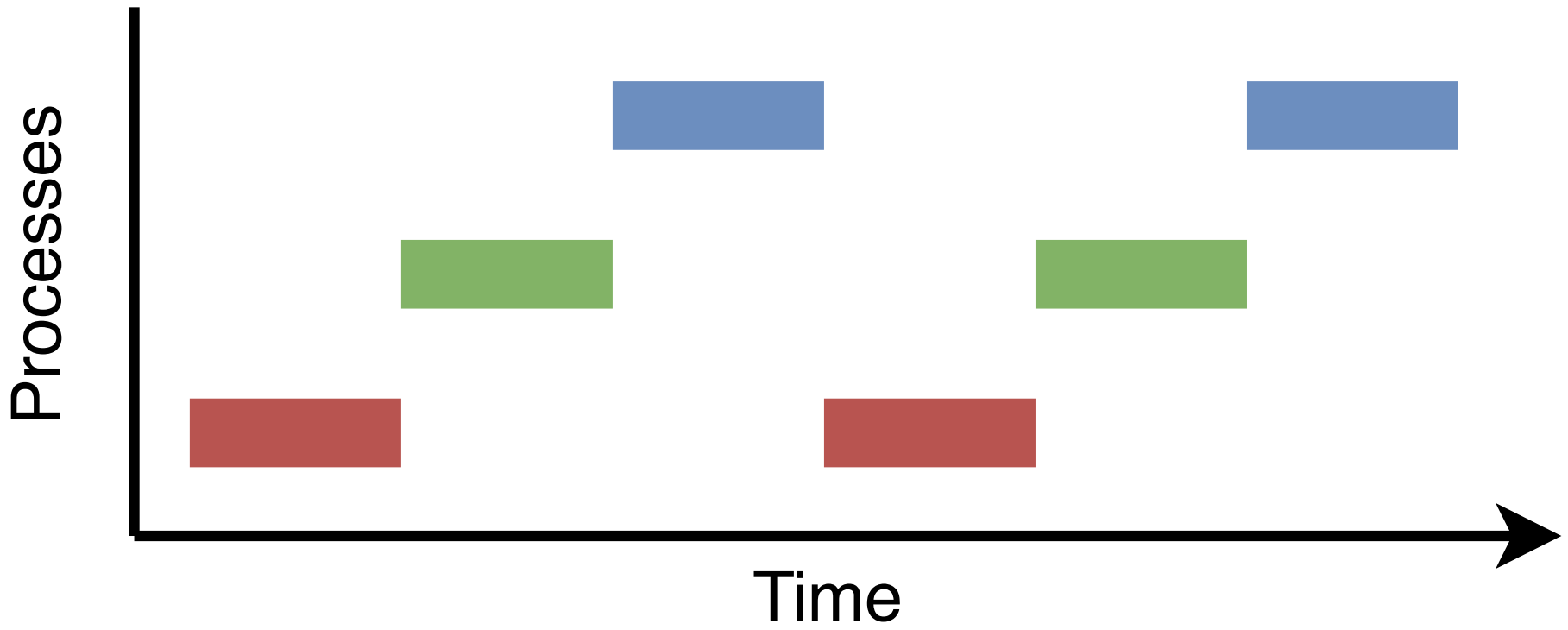
- How is the CPU usage split among processes?
- How do the instruction sequences differ, when seen by CPU vs. seen by the process?
- What is the lifecycle of a process?
- What are the process states? When and how do transitions between them happen?
- How is a process created and terminated?

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

Photo by DDP on Unsplash

Photo by milan degraeve on Unsplash

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

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

Process Termination

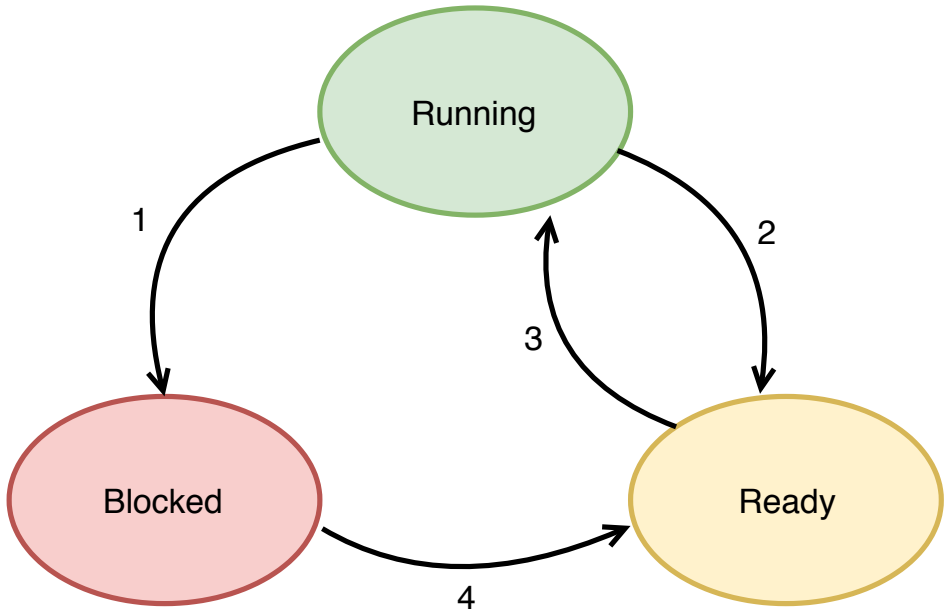
Conditions that cause a process to terminate:

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[1]
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 and set it as the return value for the parent (in the PCB's register)
7. Tell filesystem about the additional open file handles
8. Report child's memory map to kernel
9. Set both PCBs' state to Ready and start the scheduler[2]

[1] Some of these operations are done lazily today, to boost performance. Explaining today's workflow in detail would, however, complicate the description.

[2] As a result, it will seem that `fork(2)` will be called once but return twice. (Both PCBs have a Program Counter pointing to the return of the syscall, and the registers will hold the right values.)

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. Load code and data from file into memory[3]
6. Check for and handle setuid, setgid bits
7. Fix up process table entry
8. Set the process state to Ready

[3] Laziness here again.

Exit

exit() operations

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

High-level overview

Phase	Action
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)`.

Signal	Action	Comment
SIGHUP	Terminate	Hangup detected on controlling terminal
SIGINT	Terminate	Interrupt from keyboard (Ctrl-C)
SIGTERM	Terminate	Termination signal
SIGKILL	Terminate	Kill signal
SIGALRM	Terminate	Timer signal from `alarm(2)`
SIGSEGV	Dump Core	Invalid memory reference
SIGILL	Dump Core	Illegal Instruction
SIGFPE	Dump Core	Floating-point exception
SIGXCPU	Dump Core	CPU time limit exceeded
SIGCHLD	Ignore	Child stopped or terminated
SIGCONT	Continue	Continue if stopped
SIGIO	Terminate	I/O now possible
SIGPWR	Terminate	Power failure
...	...	...

Signal handler in C

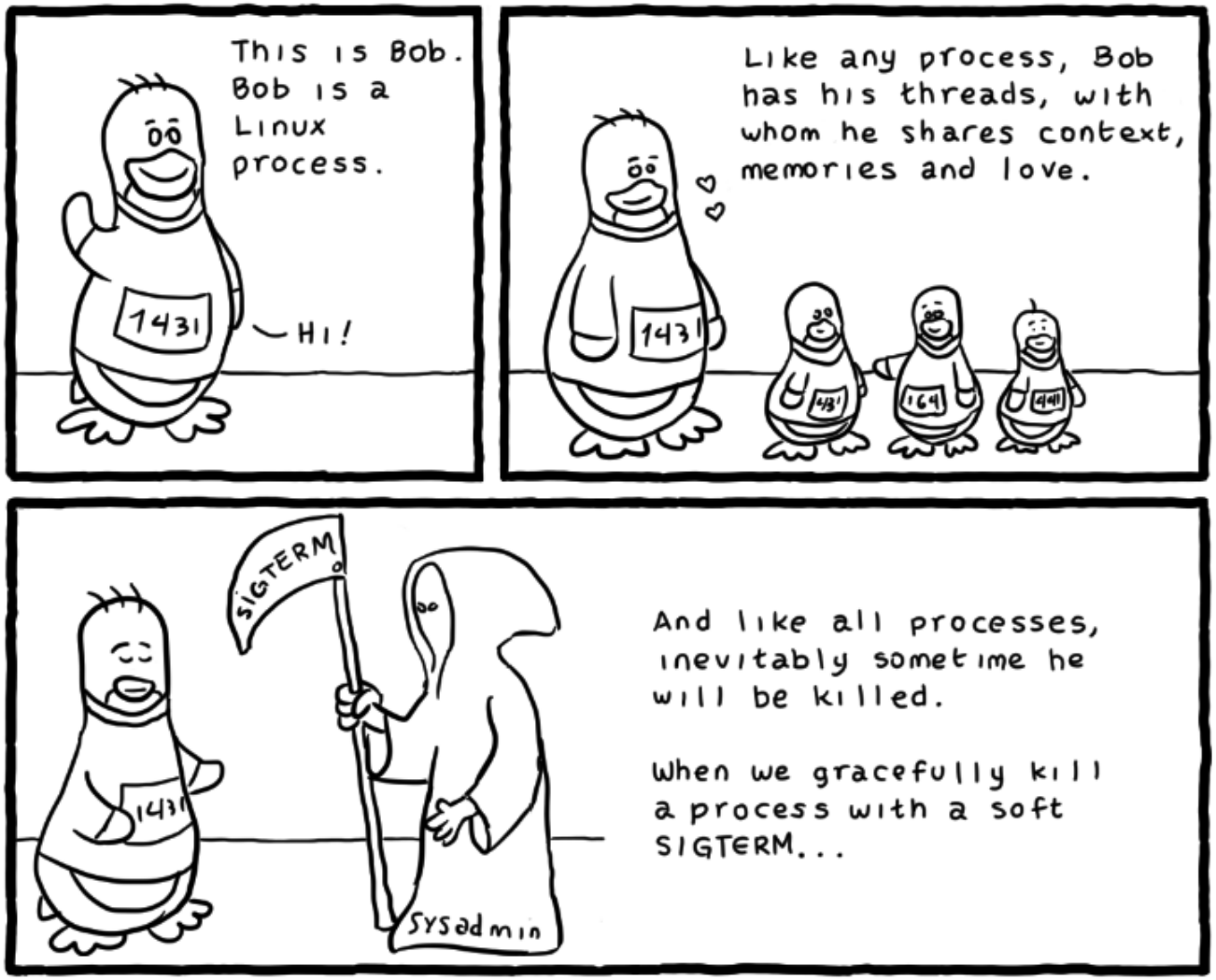
```c
#include <stdio.h>
#include <signal.h>

void handler(int sig_num)
{
   fprintf(stderr, "\nCannot be terminated using Ctrl-C :-P\n");
}

int main(void)
{
   struct sigaction act = {.sa_handler = &handler};
   sigaction(SIGINT, &act, NULL);
   while (1) {
      // infinite loop
   }
   return 0;
}
```

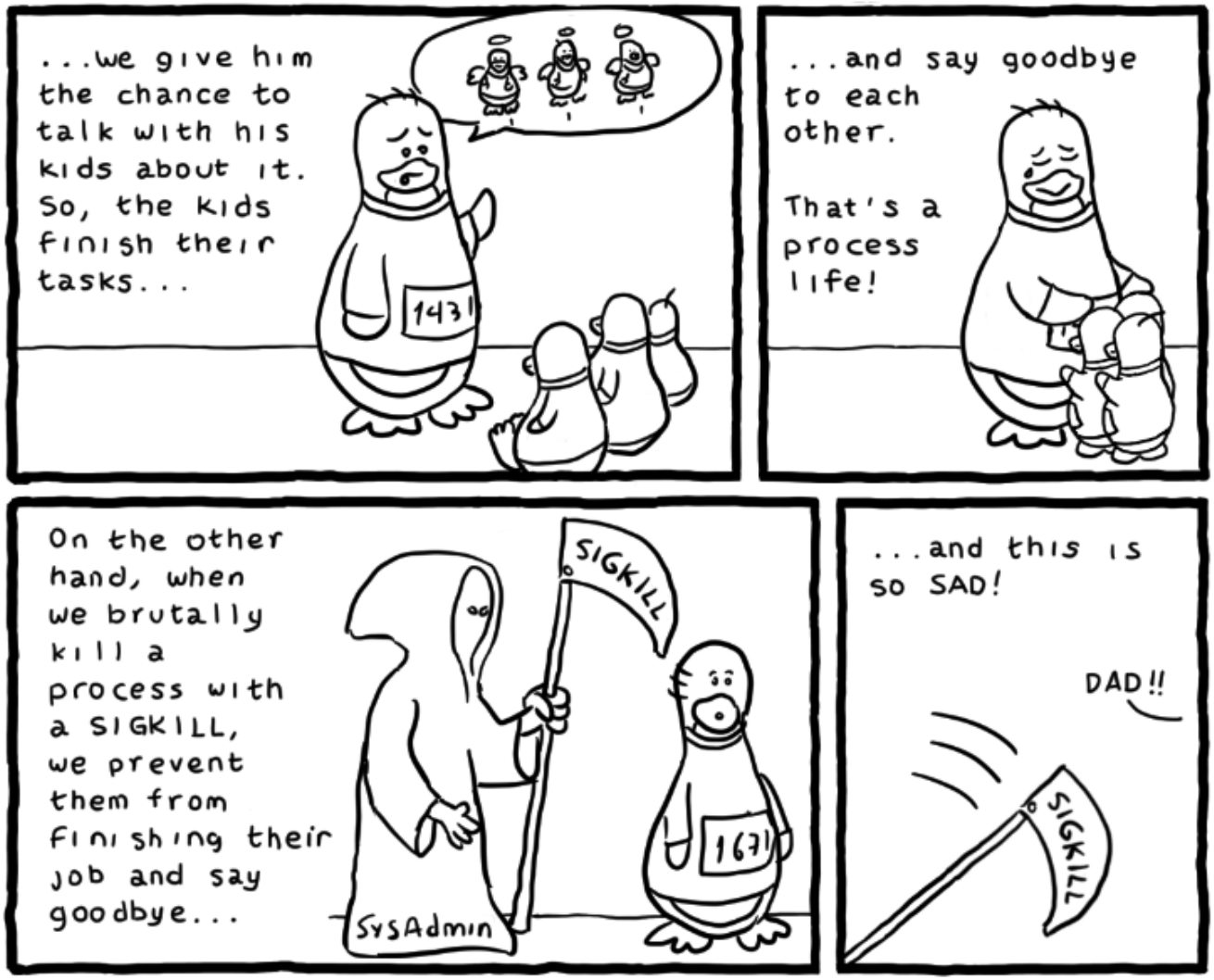
This is wasteful. Why? Busy waiting; better use e.g. `sleep`(2) or `pause`(2)

Killing me softly with this SIG...



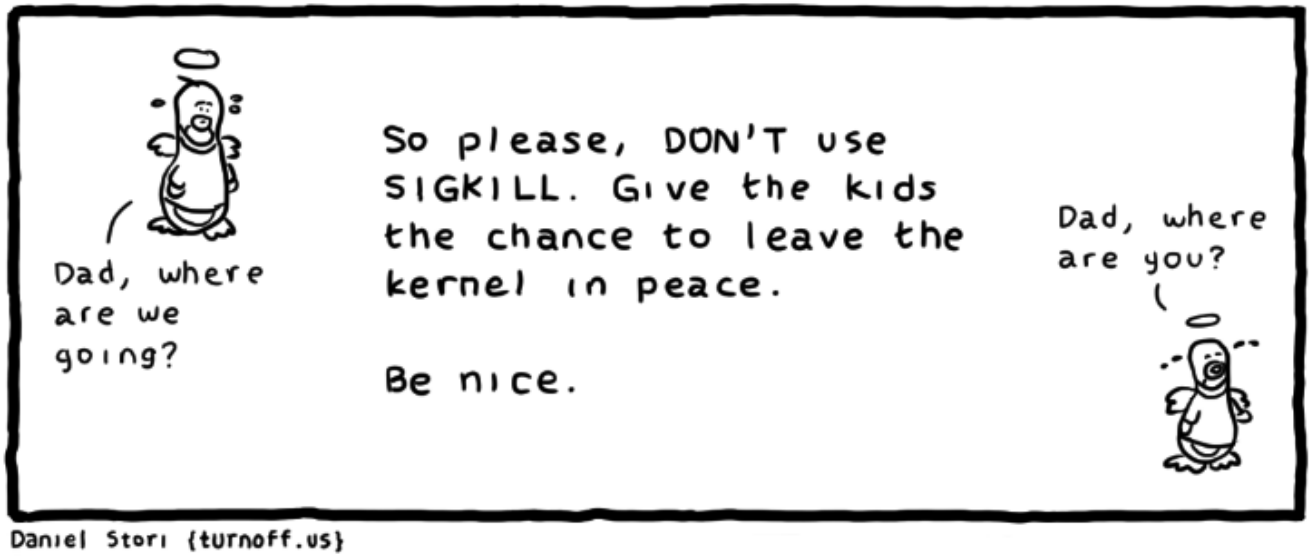
Source: https://turnoff.us/geek/dont-sigkill/

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
`kill`(2)	Send signal to another process
`sigaction`(2)	Modify response to future signal
~~`signal`(2)~~	~~`sigaction`(2)'s inconsistent predecessor, avoid~~
`alarm`(2)	Send ALRM signal to self after delay
`pause`(2)	Suspend self until future signal
`sigprocmask`(2)	Change set of blocked signals
`sigsuspend`(2)	Atomic `sigprocmask`(2), then `pause`(2)
`sigpending`(2)	Examine set of pending (blocked) signals
`sigreturn`(2)	Clean up after signal handler