Operating Systems

Timo Hönig
Bochum Operating Systems and System Software (BOSS)
Ruhr University Bochum (RUB)

III. Processes and Threads
April 26, 2023 (Summer Term 2023)

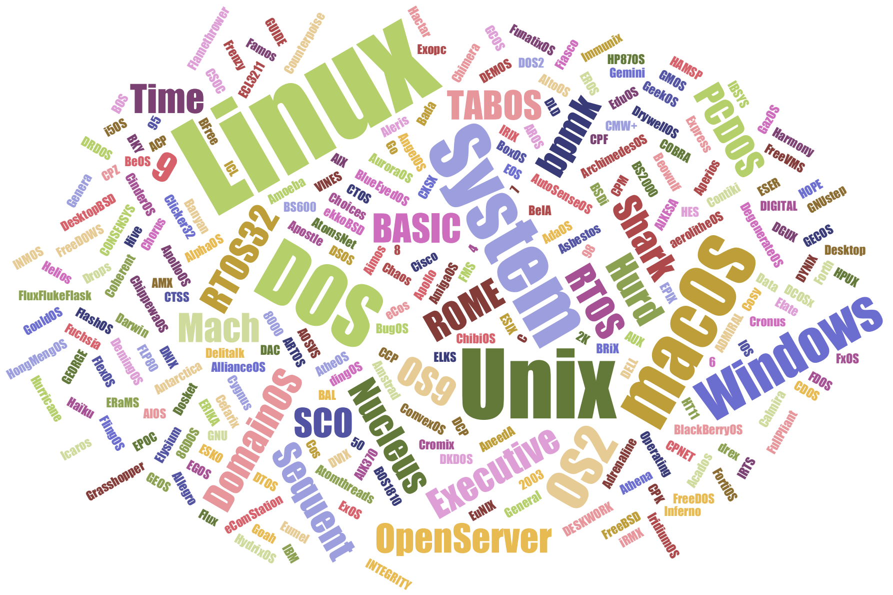
RUB Faculty of Computer Science

RUHR UNIVERSITÄT BOCHUM RUB

www.informatik.rub.de
Chair of Operating Systems and System Software

Agenda

- Recap
- Organizational Matters
- Process and Thread
  - Background and History
  - On the Weight of Processes
  - Threads (Light-weight Processes)
  - User-level Threads (Feather-weight Processes)
  - Process and Thread Models
- Processes in Practice:
  The Unix Process Model
  - Process States
  - Threads in Unix/Linux, Shells and I/O
  - Unix Philosophy
  - Process Creation
- Summary and Outlook



Literature References

Silberschatz, Chapters 3.1.-3.3, 20.4

Tanenbaum, Chapters 10.1, 10.3

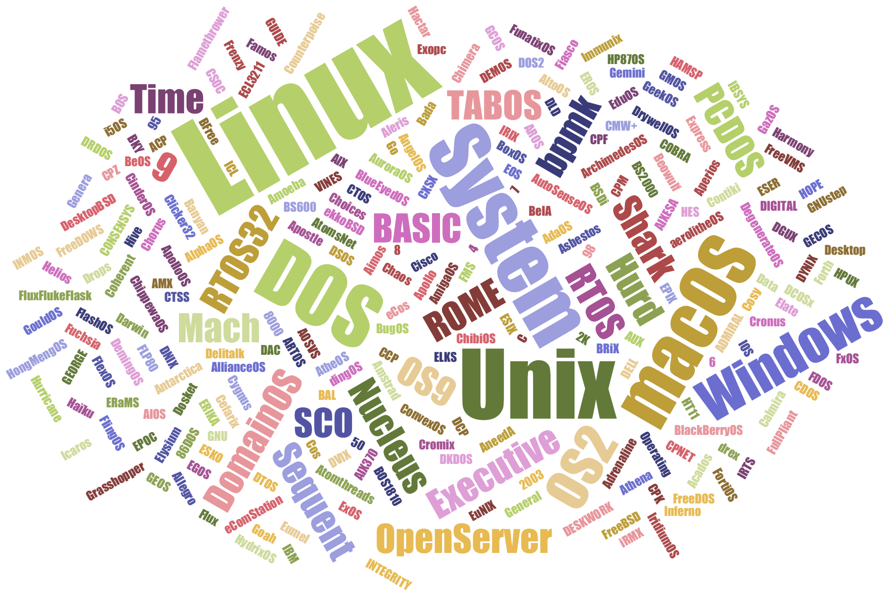
Literature References

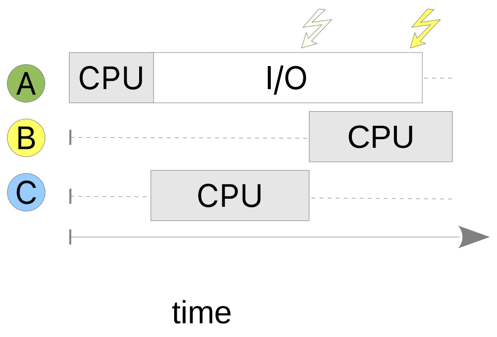
Silberschatz, Chapter 4

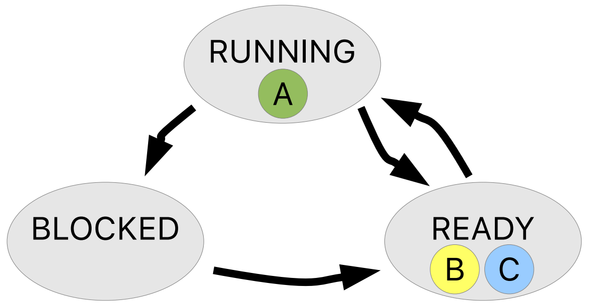
Tanenbaum, Chapter 2.2

⏮ Recap

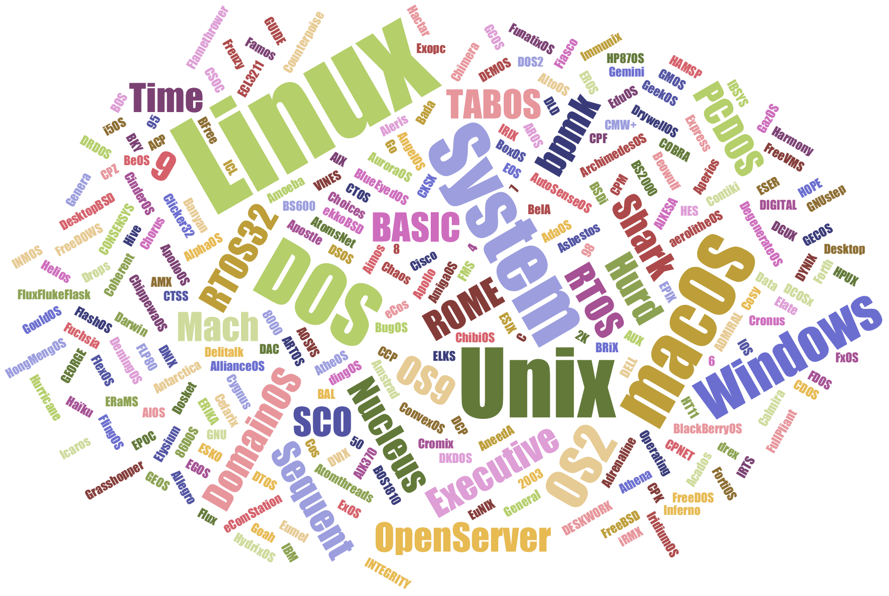
- what is a process?
  - discussion: definitions
    - → **process := program in execution**
    - → alternating CPU and I/O bursts
  - processes need **resources**
    - → CPU, memory, peripheral devices
  - processes have a **state**
    - → READY, RUNNING, BLOCKED
- **conceptually:** processes are viewed as independent, concurrent control flows
- processes are under control of the **operating system** which grants CPU time and **preempts** processes, if necessary





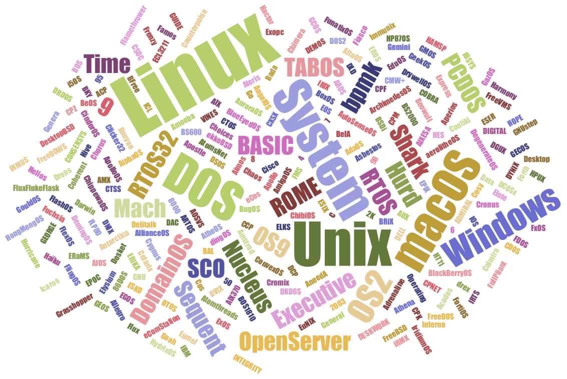


Agenda



- Recap
- Organizational Matters
- Process and Thread
  - Background and History
  - On the Weight of Processes
  - Threads (Light-weight Processes)
  - User-level Threads (Feather-weight Processes)
  - Process and Thread Models
- Processes in Practice: The Unix Process Model
  - Shells and I/O
  - Unix Philosophy
  - Process Creation
  - Process States
- Summary and Outlook

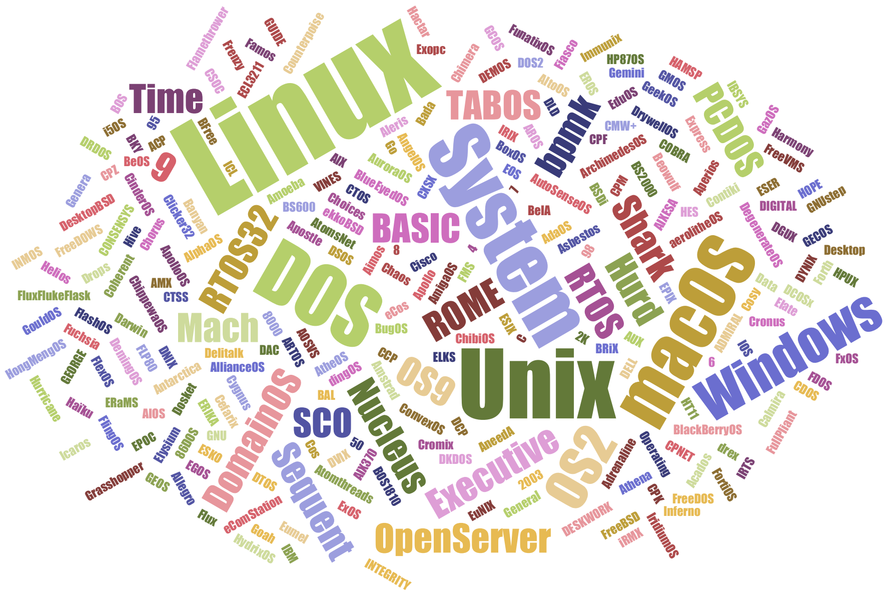
Organizational Matters



- lecture
  - Wednesday, 10:15 – 11:45
  - format: synchronous, **hybrid**
    - → in presence (Room HID, Building ID)
    - → online lecture (Zoom)
  - language: English/German
  - exercises: **group allocation ~~almost~~ complete**
    - if you are not yet signed up → Moodle
    - make use of group work - for your own benefit!
- manage course material, asynchronous communication: Moodle
- `https://moodle.ruhr-uni-bochum.de/course/view.php?id=50698`

Agenda

- Recap
- Organizational Matters
- Process and Thread
  - Background and History
  - On the Weight of Processes
  - Threads (Light-weight Processes)
  - User-level Threads (Feather-weight Processes)
  - Process and Thread Models
- Processes in Practice: The Unix Process Model
  - Process States
  - Threads in Unix/Linux, Shells and I/O
  - Unix Philosophy
  - Process Creation
- Summary and Outlook

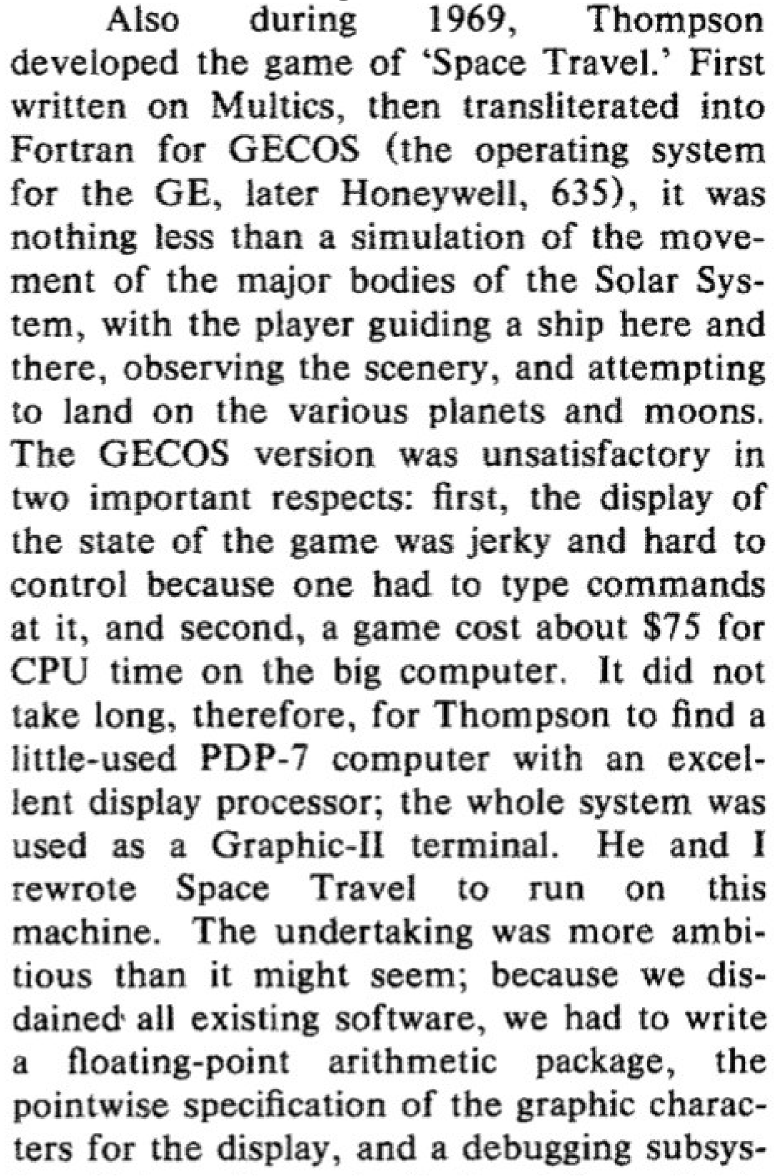


Unix by Ken Thompson, Dennis Ritchie (1968)

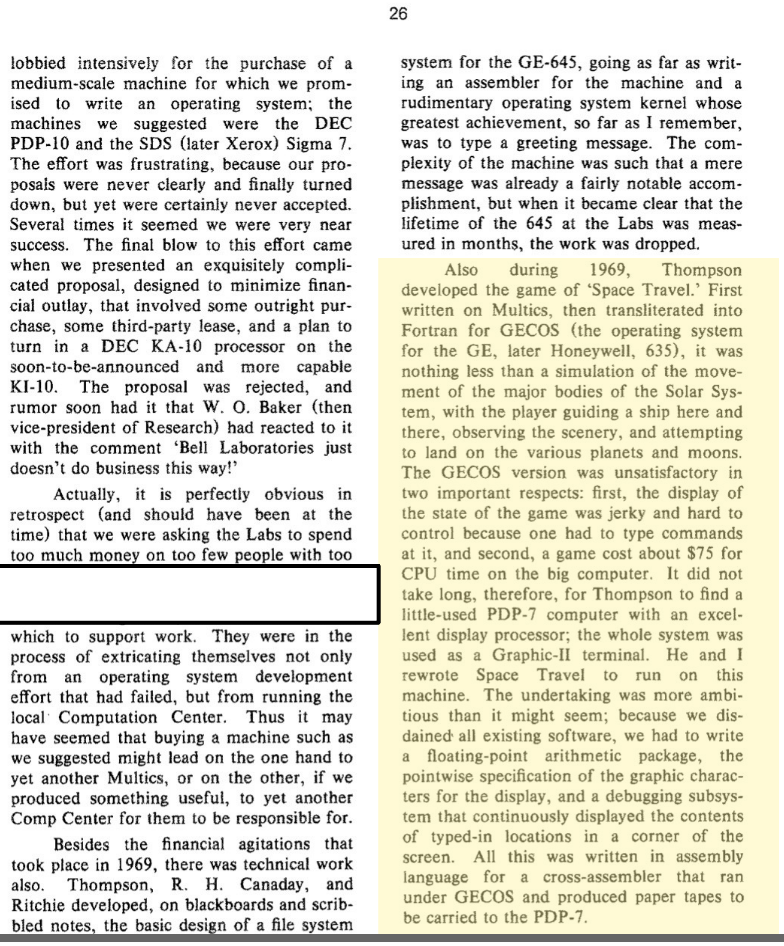
- a long (success) history...
- origin: Bell Labs
  - alternative to „Multics“
- Unix Version 1 (1972) was developed on a DEC PDP 7
  - assembler, 9 kByte memory, 18 bit words
- Unix Version 3 (1981) programmed in „C“

System: PDP-11, 16-bit *minicomputers* by Digital Equipment Corp. (DEC)
Programmer: Ken Thompson (sitting) and Dennis Ritchie
Date: ca. 1972, Source: https://www.bell-labs.com/usr/dmr/www/picture.html

Unix by Ken Thompson, Dennis Ritchie (1968)



Also during 1969, Thompson developed the game of 'Space Travel.' First written on Multics, then transliterated into Fortran for GECOS (the operating system for the GE, later Honeywell, 635), it was nothing less than a simulation of the movement of the major bodies of the Solar System, with the player guiding a ship here and there, observing the scenery, and attempting to land on the various planets and moons. The GECOS version was unsatisfactory in two important respects: first, the display of the state of the game was jerky and hard to control because one had to type commands at it, and second, a game cost about $75 for CPU time on the big computer. It did not take long, therefore, for Thompson to find a little-used PDP-7 computer with an excellent display processor; the whole system was used as a Graphic-II terminal. He and I rewrote Space Travel to run on this machine. The undertaking was more ambitious than it might seem; because we disdained all existing software, we had to write a floating-point arithmetic package, the pointwise specification of the graphic characters for the display, and a debugging subsys-



26

lobbied intensively for the purchase of a medium-scale machine for which we promised to write an operating system; the machines we suggested were the DEC PDP-10 and the SDS (later Xerox) Sigma 7. The effort was frustrating, because our proposals were never clearly and finally turned down, but yet were certainly never accepted. Several times it seemed we were very near success. The final blow to this effort came when we presented an exquisitely complicated proposal, designed to minimize financial outlay, that involved some outright purchase, some third-party lease, and a plan to turn in a DEC KA-10 processor on the soon-to-be-announced and more capable KI-10. The proposal was rejected, and rumor soon had it that W. O. Baker (then vice-president of Research) had reacted to it with the comment 'Bell Laboratories just doesn't do business this way!'

Actually, it is perfectly obvious in retrospect (and should have been at the time) that we were asking the Labs to spend too much money on too few people with too

which to support work. They were in the process of extricating themselves not only from an operating system development effort that had failed, but from running the local Computation Center. Thus it may have seemed that buying a machine such as we suggested might lead on the one hand to yet another Multics, or on the other, if we produced something useful, to yet another Comp Center for them to be responsible for.

Besides the financial agitations that took place in 1969, there was technical work also. Thompson, R. H. Canaday, and Ritchie developed, on blackboards and scribbled notes, the basic design of a file system

system for the GE-645, going as far as writing an assembler for the machine and a rudimentary operating system kernel whose greatest achievement, so far as I remember, was to type a greeting message. The complexity of the machine was such that a mere message was already a fairly notable accomplishment, but when it became clear that the lifetime of the 645 at the Labs was measured in months, the work was dropped.

Also during 1969, Thompson developed the game of 'Space Travel.' First written on Multics, then transliterated into Fortran for GECOS (the operating system for the GE, later Honeywell, 635), it was nothing less than a simulation of the movement of the major bodies of the Solar System, with the player guiding a ship here and there, observing the scenery, and attempting to land on the various planets and moons. The GECOS version was unsatisfactory in two important respects: first, the display of the state of the game was jerky and hard to control because one had to type commands at it, and second, a game cost about $75 for CPU time on the big computer. It did not take long, therefore, for Thompson to find a little-used PDP-7 computer with an excellent display processor; the whole system was used as a Graphic-II terminal. He and I rewrote Space Travel to run on this machine. The undertaking was more ambitious than it might seem; because we disdained all existing software, we had to write a floating-point arithmetic package, the pointwise specification of the graphic characters for the display, and a debugging subsys-tem that continuously displayed the contents of typed-in locations in a corner of the screen. All this was written in assembly language for a cross-assembler that ran under GECOS and produced paper tapes to be carried to the PDP-7.

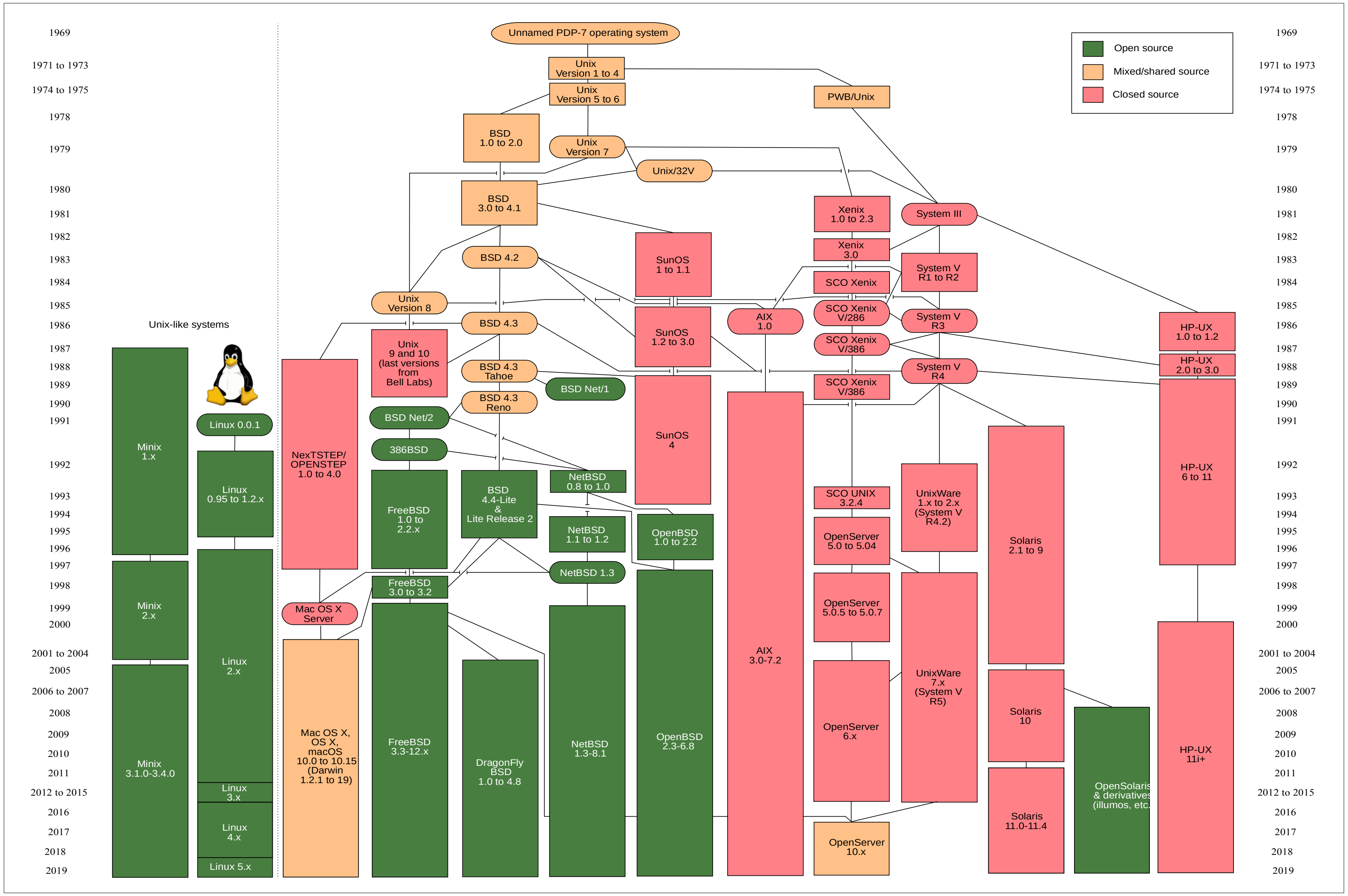
Dennis Ritchie: The evolution of the Unix time-sharing system. Proceedings of a Symposium on Language Design and Programming Methodology, 1979.
https://link.springer.com/content/pdf/10.1007/3-540-09745-7_2.pdf

Evolution of Unix, its Derivates, and Relatives

1968...

...1991...

...today



Source: Wikipedia

On the Weight of Processes

- the **weight** of a process is a figurative expression
- the weight is reflected with regard to its **operating efforts** (i.e., creation, scheduling, and termination of the process)
  - for the **size of its context** and, thus, the efforts (i.e., time, memory) needed for a context **switch**
  - CPU scheduling decisions
  - context saving and loading
- classic Unix processes are "heavyweight"

Thread Models

- kernel-level threading (1:1, one-to-one)
  - *light-weight processes*: multiple control flows in a single address space
  - each user-level thread maps to one kernel-level thread → `pthreads`
- user-level threading (m:1, many-to-one)
  - *feather-weight processes*: multiple <u>user-level</u> control flows in a single address space
  - many user-level threads map to one kernel-level thread
- hybrid threading (m:n, many-to-many)
  - *feather-weight process*: multiple <u>user-level</u> control flows <u>using multiple kernel-level threads</u> in a single address space
  - many user-level threads map to many kernel-level threads

Threads (Light-weight Processes)

- 1:1 relationship **control flow ↔ address space** is dissolved
  - cooperating threads (*dt.* Fäden) share a single address space
  - threads share all segments (`code`, `data`, `bss`, `heap`) **except the** stack
- **pro**:
  - complex operations can be offloaded to one or more threads while the parent process concurrently responds to input
    - typical example: web server concurrently handling multiple requests
  - programs that consist of multiple independent control flows benefit directly from multiprocessor hardware → parallelisation
  - fast context switching → staying in the same address space
  - depending on the scheduler, possibly more computing time
- **contra**:
  - multi-threaded programming is **hard**:
    access to shared data must be coordinated and synchronised

User-level Threads (Feather-weight Processes)

- user-level threads are completely implemented at the **application level** - the operating system has no knowledge about user-level threads
- **pro:**
  - extremely fast context switch: only a few processor registers have to be exchanged
  - no trap into kernel mode necessary
  - shared system resources increases efficiency
- **contra:**
  - blocking of a user-level thread leads to blocking of the whole process
  - no speed advantage due to multiple cores
  - no additional computing time

Threads in Linux

- Linux implements POSIX threads with the `pthread` library
- access by a Linux-specific system call

Linux System Call:

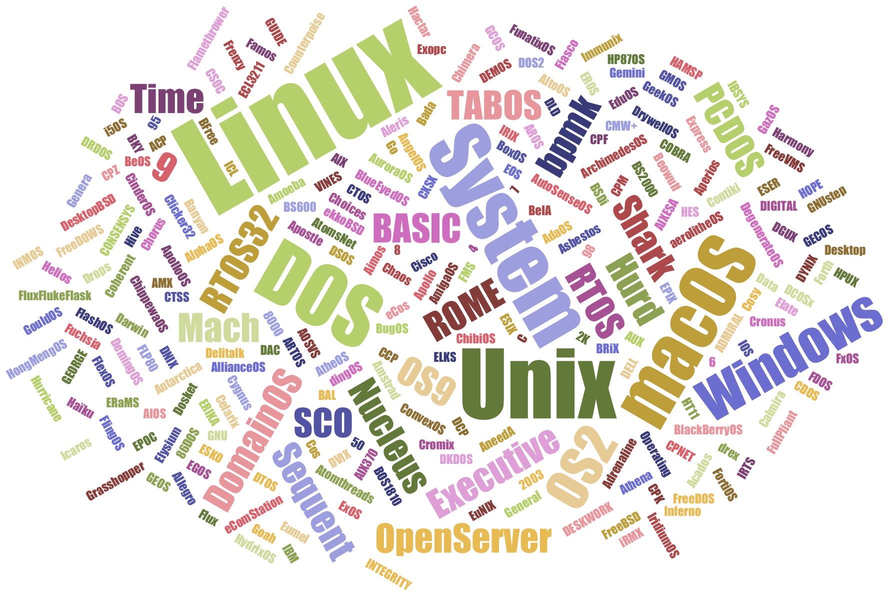
```
int __clone  (int (*fn)(void *), void *stack,
                int flags, void *arg)
```

- universal function parametrised by **flags**

`CLONE_VM`	share address space
`CLONE_FS`	share information about file system
`CLONE_FILES`	share file descriptors (e.g., open files)
`CLONE_SIGHAND`	share signal handling table

- to Linux, all threads and processes are mapped to "tasks" → `struct task_struct`
- the Linux process schedulers makes **no difference** between heavy-weight and light-weight processes

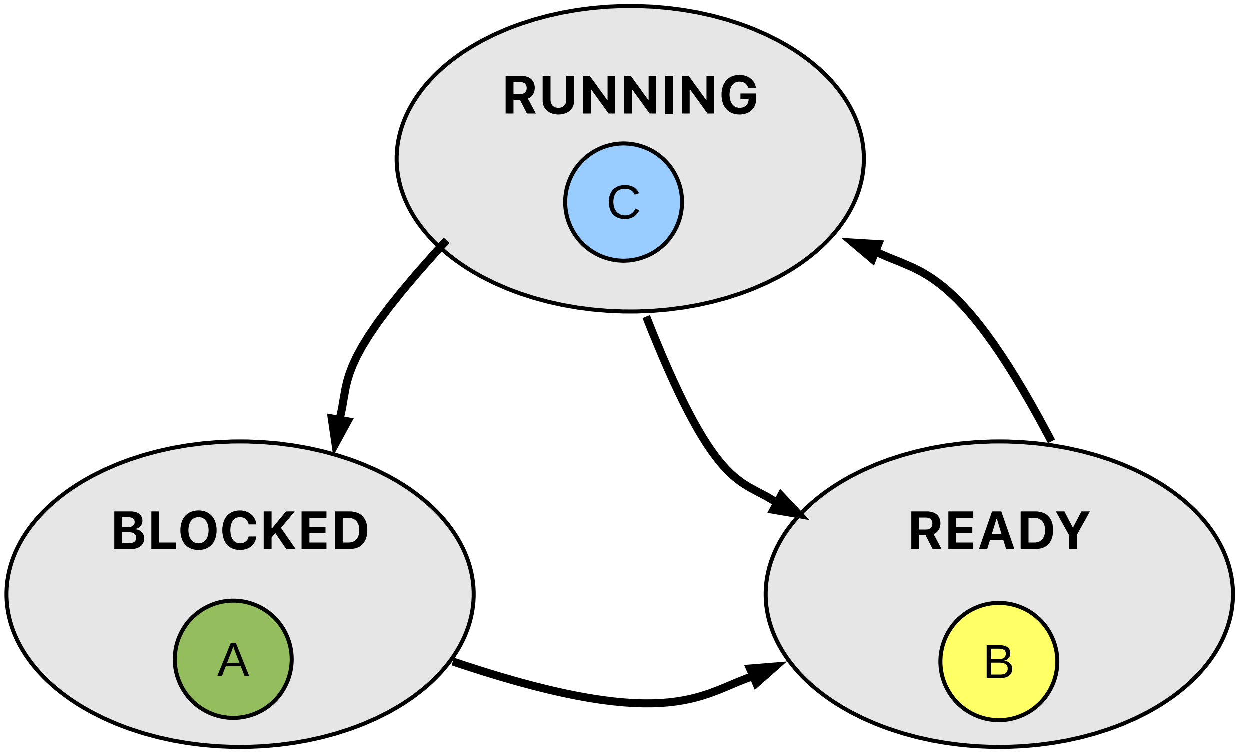
Agenda



- Recap
- Organizational Matters
- Process and Thread
  - Background and History
  - On the Weight of Processes
  - Threads (Light-weight Processes)
  - User-level Threads (Feather-weight Processes)
  - Process and Thread Models
- Processes in Practice:
  The Unix Process Model
  - Process States
  - Threads in Linux, Shells and I/O
  - Unix Philosophy
  - Process Creation
- Summary and Outlook

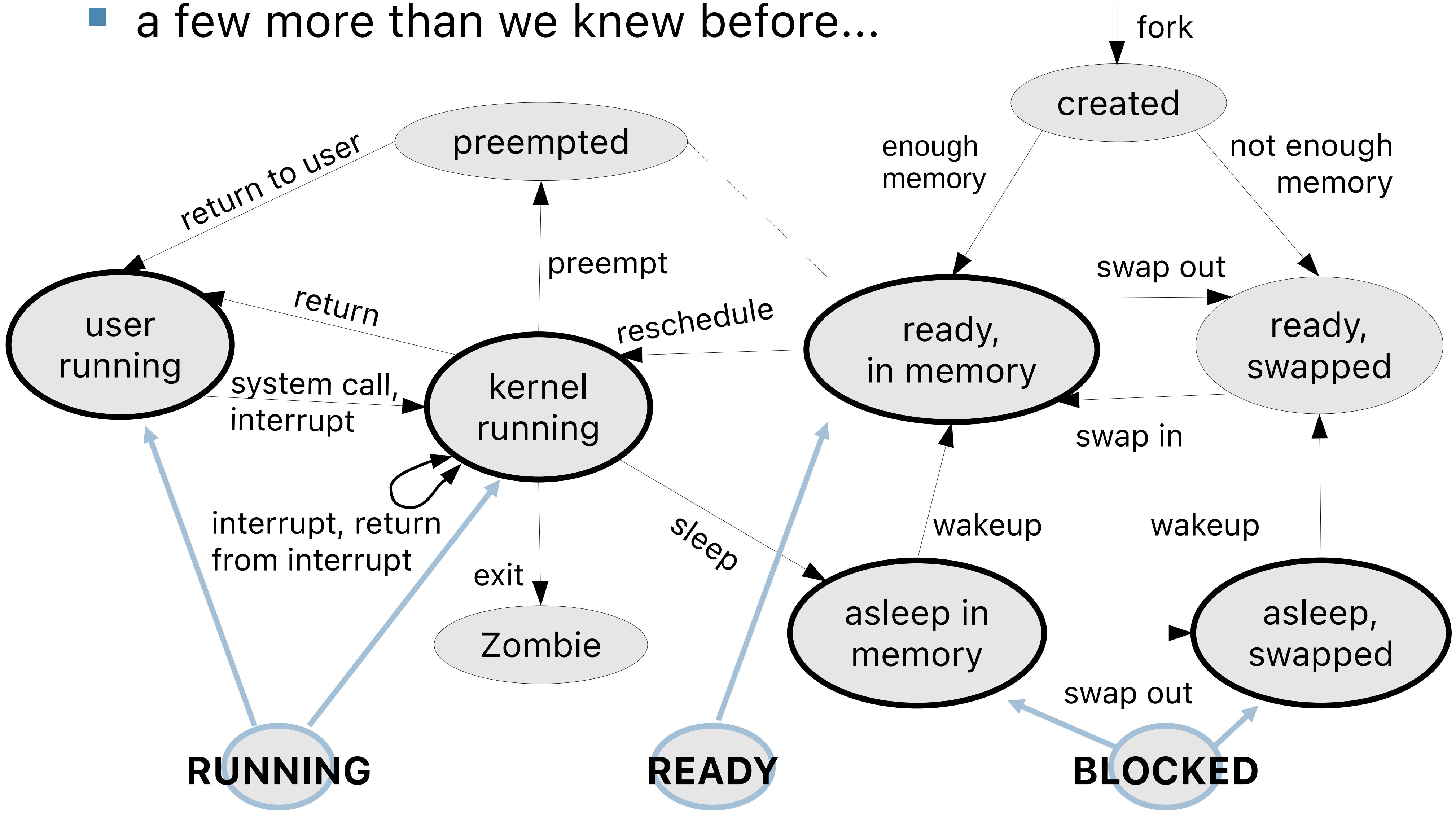
Process States in Unix

- recap



Process States in Unix

- a few more than we knew before...

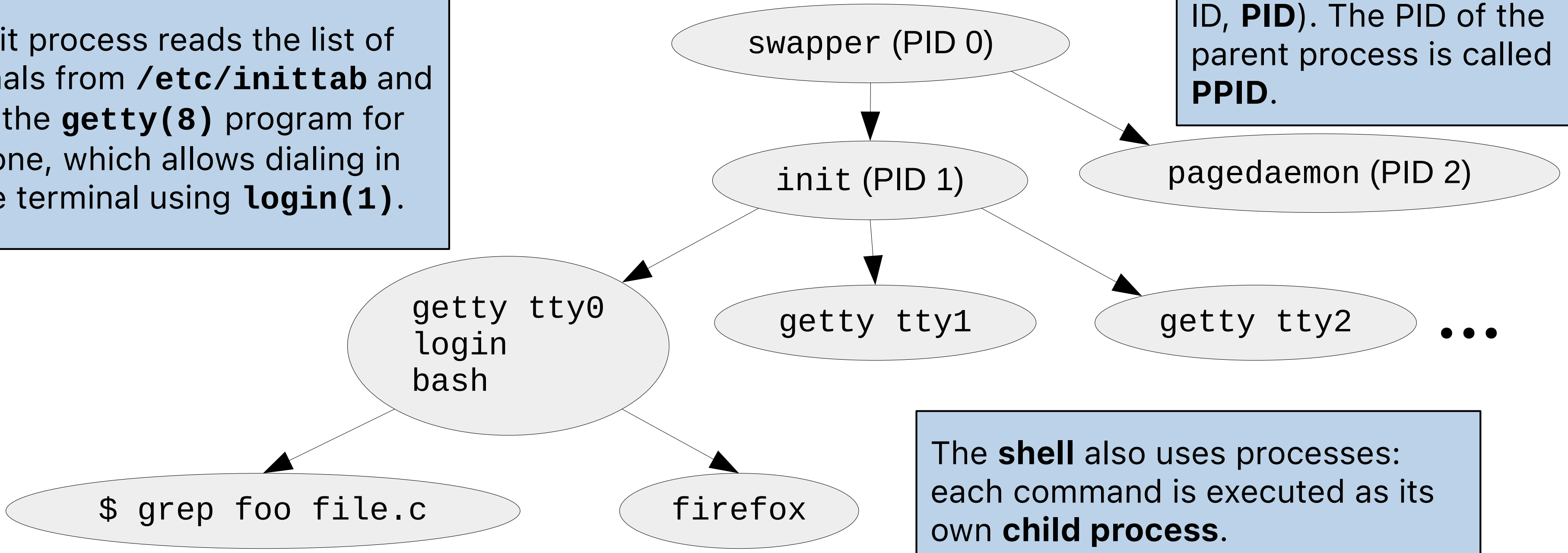


Unix Processes...

- ...are the **primary structuring concept** for activities
  - user processes vs. (operating) system processes
- ...can easily (and quickly) create **additional processes**
  - parent process → child process
- ...build a **process hierarchy**:

Each Unix process has a unique number (process ID, **PID**). The PID of the parent process is called **PPID**.

The init process reads the list of terminals from **`/etc/inittab`** and starts the **`getty(8)`** program for each one, which allows dialing in via the terminal using **`login(1)`**.



The **shell** also uses processes: each command is executed as its own **child process**.

pstree

```
systemd-+-agetty
        |-avahi-daemon---avahi-daemon
        |-cron---cron---sh---tv-service.sh---ping
        |-dbus-daemon
        |-dhclient
        |-exim4
        |-mount.davfs
        |-nmbd
        |-ntpd---{ntpd}
        |-polkitd---2*[{polkitd}]
        |-rngd---3*[{rngd}]
        |-rsyslogd---3*[{rsyslogd}]
        |-screen-+-zsh
        |        |-zsh---sudo---slurm
        |        |-zsh---bash
        |        |-zsh---pstree
        |        `-zsh---bash
        |-smbd-+-cleanupd
        |      |-lpqd
        |      `-smbd-notifyd
        |-sshd-+-sshd---sshd---screen
        |      `-sshd---sshd
        |-systemd---(sd-pam)
        |-systemd-journal
        |-systemd-logind
        `-systemd-udevd
```

```
init-+-collectdmon---collectd---12*[{collectd}]
     |-cron---cron---sh---perl
     |-dbus-daemon
     |-dovecot-+-anvil
     |         |-config
     |         |-12*[imap]
     |         |-12*[imap-login]
     |         |-log
     |         `-stats
     |-fail2ban-server---10*[{fail2ban-server}]
     |-getty
     |-irqbalance---{irqbalance}
     |-master-+-pickup
     |        |-qmgr
     |        `-tlsmgr
     |-nginx---4*[nginx]
     |-ntpd---{ntpd}
     |-php-fpm7.4---3*[php-fpm7.4]
     |-rsyslogd---3*[{rsyslogd}]
     |-screen-+-zsh---sudo---log.sh---sudo---multitail---
6*[tail]
     |        |-zsh---sudo---slurm
     |        |-zsh---pstree
     |        `-2*[zsh]
     |-smartd
     |-spamd---2*[spamd child]
     |-sshd---sshd---sshd---screen
     |-startpar
     |-sudo---sh---ssh
     |-systemd-udevd
     |-unbound
     `-watchdog
```

Interlude: Unix Shells

- shells (*dt. Außenhaut, Randzone, Ummantelung*) **wrap** the kernel (*dt. Kern*) and provide a **command line interface**
- text-based user interface for executing commands:
  - Example: locate a command according to the `$PATH` (e.g., `/usr/bin:/bin etc.`) environment variable

```
user@host:~> which vi
/usr/bin/vi
```

The command **`which`** shows where a particular command is found.

  - each executed command is a separate **child process**
  - the shell typically **blocks** until child commands terminate
  - one can **stop** and **continue** commands (job control) or request **background execution**

Interlude: Unix Shells, Job Control

```
user@host:~> vi foo.c
```

- start command (vi)
- shell blocks

Ctrl-Z

- command is stopped
- shell resumes

```
[1]+  Stopped     vi foo.c
user@host:~> emacs bar.c &
[2] 19504
user@host:~> jobs
[1]+  Stopped     vi foo.c
[2]-  Running     emacs bar.c &
user@host:~> bg %1
[1]+ vi foo.c &
user@host:~> jobs
[1]-  Running     vi foo.c &
[2]+  Running     emacs bar.c &
```

- the supplied parameter **&** at the end will start command **emacs** in the background

- **jobs** shows all started commands

- **bg** sends a already stopped command into the background, it resumes

Standard I/O Channels of Unix Processes

- usually connected to the **terminal** running the shell that started the process:
  - **standard input (`stdin`)** — read user input (keyboard)
  - **standard output (`stdout`)** — process text output (terminal)
  - **standard error (`stderr`)** — separate channel for error messages (also the terminal, by default)
- most of the commands also accept **files** as **input or output** channels (instead of the terminal)
- shells provide a simple **interface** to redirect the standard I/O channels...

Redirecting Standard I/O Channels

redirect the standard output to the file "`file1.txt`" with **>**

```
user@host:~> ls -l > file1.txt
user@host:~> grep "Operating Systems" < file1.txt > file2.txt
user@host:~> wc < file2.txt
  2  18 118
```

redirect the standard input to the file "`file2.txt`" with <

Even more compact:

```
user@host:~> ls -l | grep "Operating Systems" | wc
  2  18 118
```

with **|** (pipe), the shell connects the standard output of the left process to the standard input of the right process

The Unix Philosophy

Douglas McIlroy, the inventor of Unix pipes, once summarized the philosophy behind Unix as follows:

"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."

Common abbreviation:
„Do one thing, do it well.“

Unix System Calls: Process Control

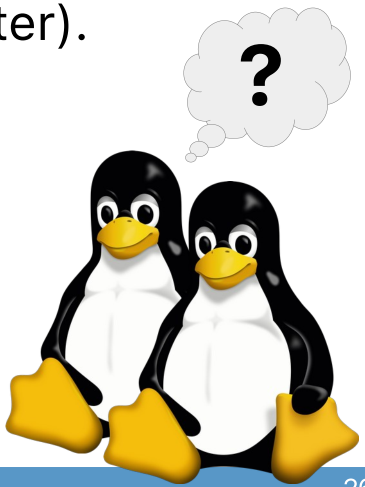
A first overview...

`getpid(2)`	returns PID of the running process
`getppid(2)`	returns PID of the parent process (PPID)
`getuid(2)`	returns the user ID of the running process (UID)
`fork(2)`	creates new child process
`exit(3)`, `_exit(2)`	terminates the running process
`wait(2)`	waits for the completion of a child process
`execve(2)`	loads and starts a program in the context of the running process

Unix Processes in detail: fork

System Call: `pid_t fork (void)`

- duplicates the running process (**process creation**)
- the child process inherits:
  - address space (`code, data, bss, heap, stack`)
  - user ID
  - standard I/O channels
  - process group, signal table (more about this later)
  - open files, current working directory (more on this *much* later).
- what is <u>not</u> copied:
  - process ID (PID), parent process ID (PPID)
  - pending signals, accounting data, ...
- <u>one</u> process calls fork, but <u>two</u> processes return...



Using `fork()`

```
... /* includes */
int main () {
  int pid;
  printf("Parent process: PID %d PPID %d\n", getpid(), getppid());
  pid = fork(); /* process is duplicated!
                   Both continue to run at this point. */
  if (pid > 0) /* parent process */
    printf("In the parent process, child PID %d\n", pid);
  else if (pid == 0) /* child process */
    printf("In the child process, PID %d PPID %d\n",
           getpid(), getppid());
  else
    printf("Error occurred.\n");
}
```

```
user@host:~> ./fork
Parent process: PID 7553 PPID 4014
In the child process, PID 7554 PPID 7553
In the parent process, child PID 7554
```

Discussion: Fast Process Creation

- copying the address space takes a lot of time
- historical solution: `vfork()`
  - parent process is suspended until the child process calls `exec..()` or terminates with `_exit()`
  - the child process simply uses code and data from the parent process (zero copying!)
  - the child process must not modify any data
    - sometimes not so easy: for example, do not call `exit()`, but `_exit()`!
- today: copy-on-write (COW)
  - with the help of the MMU, parent and child process share the same code and data segment
  - only when the child process changes data, the segment is copied
  - if `fork()` is followed directly by an `exec..()`, this does not occur
  - ➔ `fork()` with COW is hardly slower than `vfork()`

Unix Processes in Detail: _exit( )

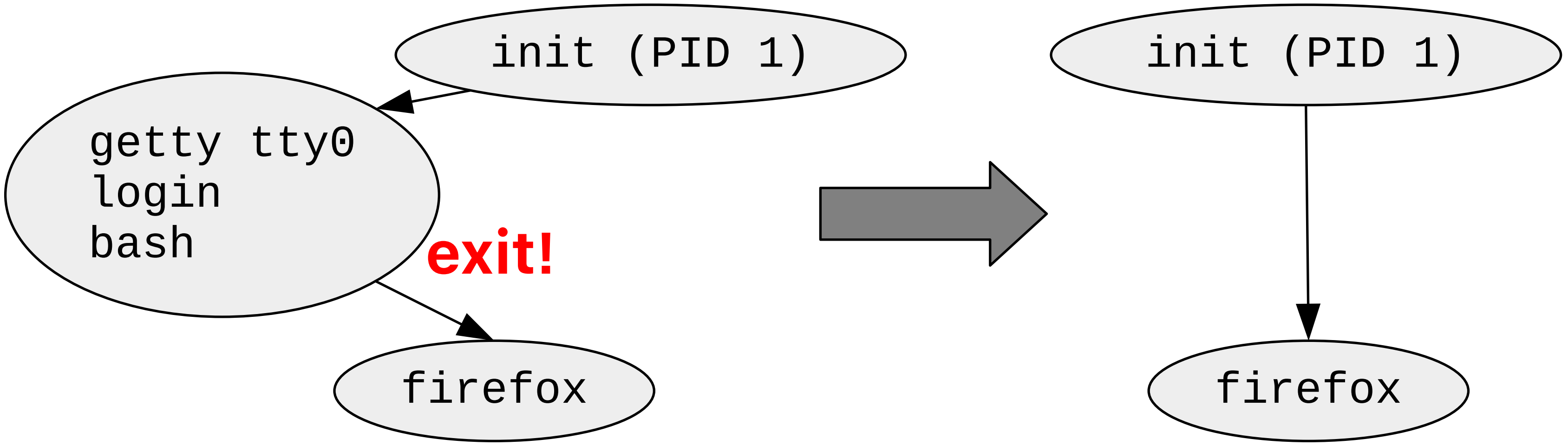
System Call: `void _exit (int)`

- terminates the running process and passes the argument as "exit status" to the parent process
  - system call does **not** return!
- OS releases the occupied resources of the process
  - open files, used memory, ...
- sends the `SIGCHLD` signal to its own parent process
- the library function `exit(3)` additionally clears the resources occupied by libc
  - buffered outputs, for example, are written out!
  - normal processes should use `exit(3)`, not `_exit`

Discussion: Orphaned Processes

(dt. „verwaiste Prozesse“)

- a Unix process becomes an orphan when its parent process terminates
- What happens to the process hierarchy?



`init (PID 1)` adopts all orphaned processes which keeps the process hierarchy intact.

Unix Processes in Detail: _wait( )

System Call: `pid_t wait (int *)`

- blocks the calling process (caller) until a child process terminates
    - the return value is its PID
    - via the pointer argument the caller gets the "exit status"
- if a child process is already terminated, the call returns immediately

```
$ man 2 wait
RETURN VALUE
       wait(): on success, returns the process ID of the terminated child; on error, -1 is returned.
```

Using `wait()`

```
/* ... includes, main() { ... */
pid = fork(); /* create child process */
if (pid > 0) { /* parent process */
  int status;
  sleep(5); /* library function: 5 s sleep */
  if (wait(&status) == pid && WIFEXITED(status))
    printf ("Exit Status: %d\n", WEXITSTATUS(status));
}
else if (pid == 0) { /* child process */
  exit(42);
}
...
```

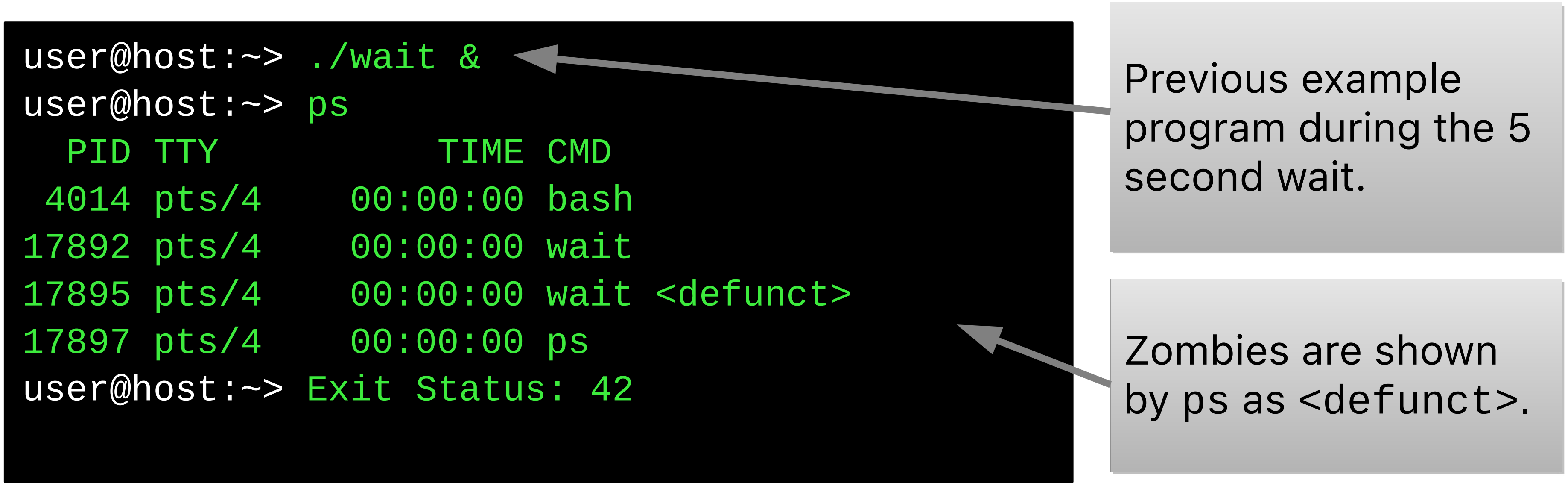
- a process can also be killed from the outside: it does not call exit
- in this case **`WIFEXITED`** would return 0

```
user@host:~> ./wait
Exit Status: 42
```

```
$ man 2 wait
WIFEXITED:    returns true if the child terminated normally
WEXITSTATUS:  returns the exit status of the child, (…)
              This macro should be employed only if WIFEXITED returned true.
```

Discussion: Zombies

- before the exit status of a terminated process is queried using **`wait`**, it is a so-called "zombie"
- allocated resources of such processes can be released, but the process management (OS) still needs to know them
  - in particular, the *exit status* must be saved



```
user@host:~> ./wait &
user@host:~> ps
  PID TTY          TIME CMD
 4014 pts/4    00:00:00 bash
17892 pts/4    00:00:00 wait
17895 pts/4    00:00:00 wait <defunct>
17897 pts/4    00:00:00 ps
user@host:~> Exit Status: 42
```

Previous example program during the 5 second wait.
Zombies are shown by ps as <defunct>.

Unix Processes in Detail: `execve()`

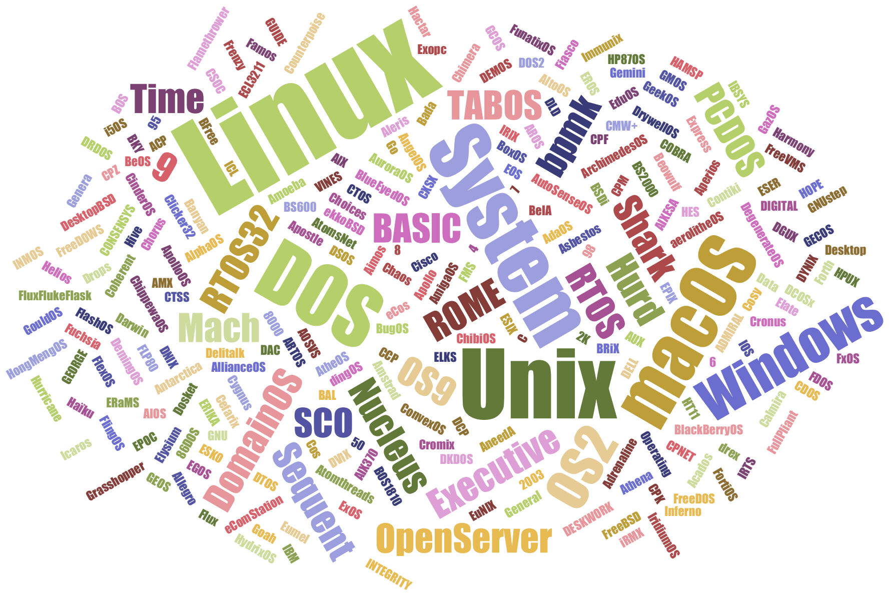
System Call:
```
int execve ( const char *command,
             const char *args[ ],
             const char *envp[ ] )
```

- loads and starts the **specified command**
- the call returns only in case of an error
- the complete address space is replaced
- however, it is still the **same** process!
  - same PID, PPID, open files, ...
- the **`libc`** provides some convenient helper functions that internally call `execve`: `execl`, `execv`, `execlp`, `execvp`, ...

Using exec..()

```
/* ... includes, main() { ... */
char cmd[100], arg[100];
while (1) {
  printf ("Command?\n");
  scanf ("%99s %99s", cmd, arg);
  pid = fork(); /* Process is duplicated, two processes continue
                   to run at this point. */
  if (pid > 0) {
    int status;
    if (wait(&status) == pid && WIFEXITED(status))
      printf ("Exit Status: %d\n", WEXITSTATUS(status));
  }
  else if (pid == 0) {
    execlp(cmd, cmd, arg, NULL);
    printf ("exec has failed\n");
  }
...
}
```

Agenda



⏭ Summary and Outlook

- **summary**
  - **processes** are the central abstraction for activities in today's operating systems
  - **process control**
    - Unix systems provide distinct system calls to **create**, **manage** and **link** processes
    - all in the spirit of the philosophy: "**do one thing, do it well**"
  - **weight** of processes
    - heavyweight processes (classic Unix process model)
    - faster due to lower operational efforts:
      threads (light-weight), user-level threads (feather-weight)
- **outlook: synchronisation**
  - **process synchronisation** to resolve conflicts in concurrent use of system resources, coordination of cooperation
  - locking algorithms and methods at operating system level

References and Acknowledgments

Lecture

- Systemnahe Programmierung in C (SPiC), Betriebssysteme (Jürgen Kleinöder, Wolfgang Schröder-Preikschat)
- Betriebssysteme und Rechnernetze (Olaf Spinczyk, Embedded Software Systems Group, Universität Osnabrück)

Teaching Books and Reference Book

[1] Avi Silberschatz, Peter Baer Galvin, Greg Gagne: *Operating System Concepts*, John Wiley & Sons*, 2018*.

[2] Andrew Tanenbaum, Herbert Bos: *Modern Operating Systems*, Pearson, 2015.

[3] Wolfgang Schröder-Preikschat: *Grundlage von Betriebssystemen – Sachwortverzeichnis*, 2023.
https://www4.cs.fau.de/~wosch/glossar.pdf