

ÜBUNG 3

Pthreads, Signals & Linux Scheduling

Max Schrötter

schroetter@cs.uni-potsdam.de

Institute for Computational Science
University of Potsdam

20. November 2025



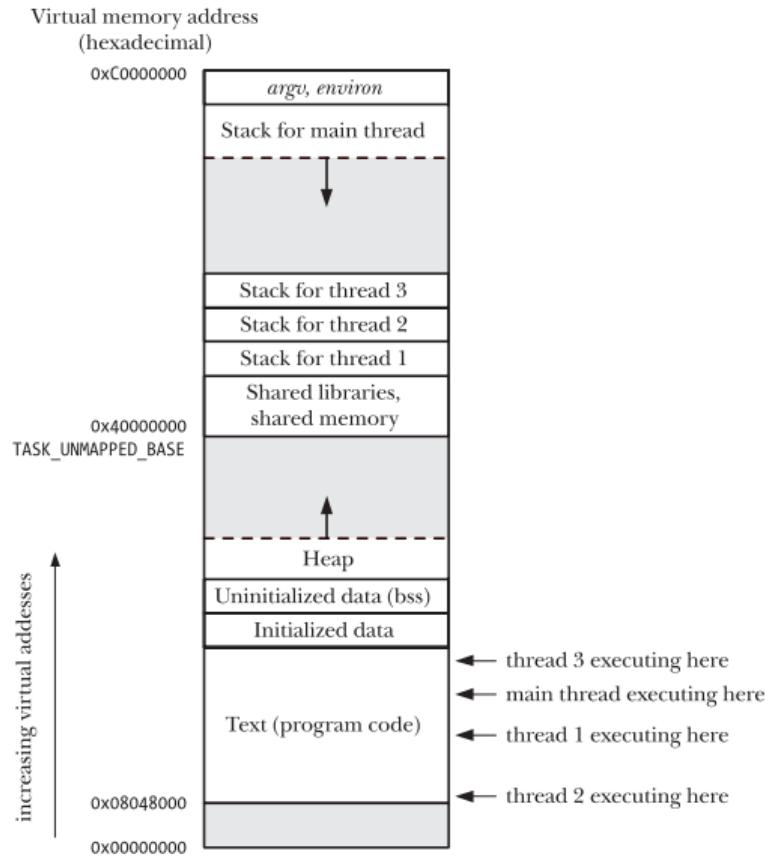
AGENDA

1. Pthreads
2. Debugging
3. Linux Scheduling
4. Signal for Broken Pipe



PTHREAD API

- `pthread_attr_init` initialisiert das `pthread_attr_t` struct
- `pthread_attr_ ... set und get` Attribute eines Threads
- `pthread_create` erstellt einen neuen Thread
- `pthread_join` wartet auf die Beendigung des angegebenen Threads
- `pthread_exit` terminiert den aktuellen Thread
- `pthread_detach` “detaches” diesen Thread
- ...



PTHREAD_CREATE

```
int pthread_create(pthread_t *restrict thread, // the thread handle
                  const pthread_attr_t *restrict attr, // attributes can be NULL
                  → to use defaults
                  void *(*start_routine)(void *), // the function to be executed
                  void *restrict arg); // void pointer to the arguments of the
                  → function
```

- Argumente: falls Sie **mehr als ein Argument** übergeben wollen, übergeben Sie einen **pointer** zu einer Struktur bestehend aus den Argumenten.
- Für **jeden** erfolgreich erzeugten thread, **muss pthread_join()** aufgerufen werden,
- außer wenn **pthread_detach()** aufgerufen wurde oder **PTHREAD_CREATE_DETACHED** in den Attributen gesetzt wurde.
- **Treffen Sie keine Annahmen über das Scheduling.**

PROZESSE & THREADS IN LINUX

- Der Linux Kernel unterscheidet nicht zwischen Prozessen und Threads, alles sind **Tasks**.
- Sie unterscheiden sich nur durch ihre Eigenschaften wie **Adressraum** und **Ressourcen**.
- Wenn ein Programm in Linux gestartet wird, wird ein **Task** im Kernel erzeugt.
- Wenn dieser Task weitere Threads erzeugt sind dies ebenfalls Tasks.
- Diese unterscheiden sich nur durch ihren Stack und Attributen.
- Sobald ein Task exit (glibc) aufruft, werden alle Tasks der selben PID beendet.
- Zum beenden eines einzelnen Threads innerhalb eines Prozesses muss `pthread_exit` aufgerufen werden.
- Das Joinen nicht detachter Threads ist notwendig, um **Resource-Leaks** zu vermeiden.

AGENDA

1. Pthreads
2. Debugging
3. Linux Scheduling
4. Signal for Broken Pipe



ANSÄTZE

- **Debugging** $\hat{=}$ **Suche und Beseitigung** von Fehlern in Programmen
- Möglichkeiten zur Fehlersuche:
 1. (wo)men who stare at code → Code anstarren
 - lobenswert (Code lesen bildet), aber: fehlerhaftes Codeverständnis **verhindert** mitunter Fehlererkennung
 2. Logging (`printf(3)`, Datei, Syslog)
 - mühsam (zusätzlicher Programmieraufwand), kann existierende Fehler **verdecken** oder neue Fehler **hervorrufen**

HEISENBUGS

Ein **Heisenbug**: falls man versucht den Fehler zu verstehen oder zu reproduzieren, tritt er plötzlich nicht mehr auf

Beispiel: Bei (Neu-)Start eines Spiels → Spielerposition und Punktestand soll zurückgesetzt werden, funktioniert jedoch nicht

```
/* initialization for game (re)starts */
if (round == 0){
    reset_xy();
    reset_score();
}
```

HEISENBUGS

Beispiel: Wende **Logging** in diesem Fall an

```
/* initialization for game (re)starts */
if (round == 0){
    printf("score: %d, x: %d und y: %d" (score , x , y));
    reset_xy();
    reset_score();
    printf("score: %d, x: %d und y: %d" (score , x , y));
}
```



Werte wurden bei **zweiter Ausführung erfolgreich** zurückgesetzt → Fehler muss in **Funktionen** auftreten

DEBUGGING

3. Einfache Alternative durch Debugger-Tool:

Idee: Programmzustand **beobachten, verfolgen oder ändern**

- **nach** Programmabsturz sowie
- **während** der Ausführung

Voraussetzungen:

3.1 Programm mit Debug-Symbolen kompilieren:

- ▶ Debug-Symbole **bewahren** Namen von Quelltext-Objekten (z.B. Funktionen, Variablen) im kompilierten Programm
- ▶ Compiler-Option (in der Regel): **-g**

3.2 nach Möglichkeit Compiler-Optimierungen abschalten:

- ▶ optimierter Code schwerer zu debuggen, da **verändert**
- ▶ Compiler-Option (in der Regel): **-O0**

GDB: THE GNU PROJECT DEBUGGER (C, C++, PYTHON)

```
vera@vera-l5410:~$ gdb ./a.out
GNU gdb (Ubuntu 9.2-0ubuntu1-20.04) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./a.out...
(gdb) break test.c:4
Breakpoint 1 at 0x1155: file test.c, line 4.
(gdb) run
Starting program: /home/vera/a.out

Breakpoint 1, main () at test.c:4
4          int i = 0, j = 3;
(gdb) next
5          printf("Hallo Welt!\n");
(gdb) info locals
i = 0
j = 3
(gdb) █
```

gdb-KOMMANDOS

- Programm unterbrechen

`break <location>`
`... if <expression>`
`watch <location>`
`catch <event>`
`delete <id>`

unterbricht wenn `<location>` erreicht wird
unterbricht an `<location>` falls `<expression>` wahr ist
unterbricht wenn speicher an `<location>` verändert wird
unterbricht wenn `<event>` eintritt
break/watch-point mit `<id>` löschen

- Programm steuern

`run <args>`
`start <args>`
`next`
`step`
`continue`
`finish`

Programm starten mit `<args>`
Programm starten, bei main sofort wieder stoppen
schrittweise ausführen, Funktionen überspringen
schrittweise ausführen, in Funktionen springen
Programm weiter laufen lassen
Programm weiter ausführen bis zur Ende der Funktion

gdb-KOMMANDOS

- Programmzustand (Variablen, Stack,...) untersuchen

<code>print [/f] <what></code>	Wert eines Ausdrucks/Variablen ausgeben (mächtig)
<code>display</code>	Print nach jedem Step
<code>x [/nfu] <address></code>	Zeigt <code><n></code> Werte des Typs <code><f></code> der Breite <code><u></code> an <code><address></code> an
<code>list [location]</code>	Quellcode anzeigen
<code>info</code>	Angaben über diverse Zustände/Daten, z.B.: <ul style="list-style-type: none">• Lokale Variablen: <code>info locals</code>• Funktionsparameter: <code>info args</code>• Threads: <code>info threads</code>• Breakpoints: <code>info breakpoints</code>
<code>backtrace</code>	Aufrufe nachverfolgen (call stack)
<code>frame <n></code>	zum call frame <code><n></code> wechseln (call stack)
<code>thread <n></code>	zum Thread <code><n></code> wechseln

gdb-KOMMANDOS

- Programmzustand verändern

`set <var>=<value>`
`return <expr>`

speichert `<value>` in `<var>`
returns die aktuelle Funktion mit `<expr>`

- GDB Verhalten

`set scheduler-locking <mode>`

Legt das Verhalten von Multithreaded Programmen fest

- `<mode>` = off: alle Threads laufen weiter
- `<mode>` = on: GDB blockiert alle Threads außer dem aktuellen
- `<mode>` = step: GDB blockiert alle Threads außer dem aktuellen beim step

- TUI

`<C+x> a`
`<C+x> 2`
`<C+x> o`
`<C+p/n>`
`<C+l>`

Öffnen & schließen der TUI
nächstes Layout
aktives Fester wechseln
history next/previous command, falls das Quellcodefenster aktiv ist
refresh screen

Weitere Kommandos siehe `help` oder Debugging with gdb

VERWENDUNG WÄHREND DER AUSFÜHRUNG (LIVE)

Ablauf (Voraussetzungen siehe Folie 14):

1. Programm in Debugger laden:

`gdb programm` (auf der Shell)

- 2a. Programm aus gdb heraus (ggf. mit Argumenten) starten:

`start` (Kommando in gdb)

bzw.

`start arg1 arg2 ...` (Kommando in gdb)

- 2b. Alternativ: an laufendes Programm anheften und debuggen

`gdb programm pid` (auf der Shell)

NACH PROGRAMMABSTURZ

Idee: Programmzustand bei Absturz **sichern und später betrachten**

Weitere Voraussetzungen:

- **Max. Größe für Speicherabzug (core dump) ändern, falls zu gering:**

`ulimit -c SIZE`

(für Bash)

(z.B. `ulimit -c unlimited`; aktuellen Wert prüfen mit `ulimit -c`)

Ablauf für Debugging mit **core dump**:

1. Programm normal **laufen lassen**

2. bei Absturz wird core dump erstellt, **auflisten** mit:

`coredumpctl list`

(für Bash)

3. Programm mit core dump (= Zustand bei Absturz) **in gdb laden**:

`coredumpctl gdb [Programmname]`

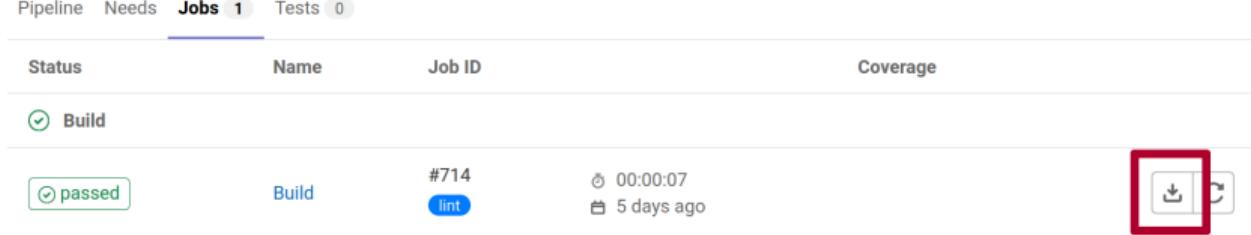
(für Bash)

4. Programmzustand untersuchen...

NACH PROGRAMMABSTURZ

Core dump kann auch in CoFee heruntergeladen werden:

Pipeline	Needs	Jobs 1	Tests 0
Status	Name	Job ID	Coverage
✓ Build			
passed	Build	#714 lint	⌚ 00:00:07 🕒 5 days ago



AGENDA

1. Pthreads
2. Debugging
3. Linux Scheduling
4. Signal for Broken Pipe



VRUNTIME

- CFS führt die Scheduling Entscheidungen basierend auf der **virtual runtime** einer Task aus

$$\text{vruntime} = \text{delta_exec} \times \frac{\text{NICE_0_LOAD}}{\text{sched_prio_to_weight(prio)}}$$

- CFS sortiert die “Tasks” als Scheduling-Entitäten in einem **Red-Black Tree (RBT)**
- **jeder Thread** gilt als eine andere Entität
→ Besonderheit von **Linux**: je mehr Threads ein Prozess hat, umso **mehr CPU-Zeit** bekommt er

PRIORITIES

Nice levels:

Nice levels are **multiplicative**, with a gentle **10% change for every nice level changed**. I.e. when a CPU-bound task goes from nice 0 to nice 1, it will get 10% less CPU time than another CPU-bound task that remained on nice 0. [...] to achieve that we use a **multiplier of 1.25**.

```
const int sched_prio_to_weight[40] = {  
    /* -20 */     88761,      71755,      56483,      46273,      36291,  
    /* -15 */     29154,      23254,      18705,      14949,      11916,  
    /* -10 */     9548,       7620,       6100,       4904,       3906,  
    /* -5 */      3121,       2501,       1991,       1586,       1277,  
    /* 0 */        1024,       820,        655,        526,        423,  
    /* 5 */        335,        272,        215,        172,        137,  
    /* 10 */       110,        87,         70,         56,         45,  
    /* 15 */       36,         29,         23,         18,         15,  
};
```

SCHEDULING GROUPS

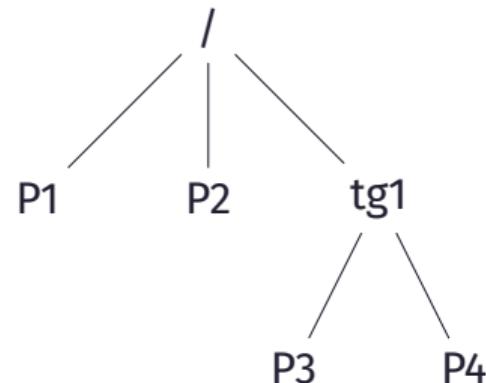
Group Scheduler Extensions to CFS:

Normally, the scheduler operates on individual tasks and strives to provide fair CPU time to each task. Sometimes, it may be desirable to group tasks and provide fair CPU time to each such task group. For example, it may be desirable to first provide fair CPU time to each user on the system and then to each task belonging to a user.

- Gruppen werden ebenfalls als Scheduling-Entitäten betrachtet
- default prio für Tasks und Gruppen beträgt 1024 (**entspricht NICE_0_LOAD**)

SCHEDULING GROUPS

- jede Gruppe hat zur Verwaltung ihren eigenen RBT, Beispiel:
 - Run queue (/): P1, P2, tg1
 - Run queue (tg1): P3, P4



UPDATING VRUNTIME

- `update_curr` aktualisiert die vruntime des **derzeitigen Tasks und aller ihrer "Vorfahren"**
- der RBT wird **balanciert**, wenn die vruntime **aktualisiert oder** eine Task **hinzugefügt** wird
- **Unterschied** bei einem idealen System: CPU-Zeit ist gleich dem **Quotienten** bestehend aus der load des einzelnen Tasks durch die load der gesamten Run-Queue

RED-BLACK-TREE

Grundlegende Eigenschaften:

1. Wurzel ist **schwarz**
2. Kinder von **roten** Knoten sind **schwarz**
3. gleiche **Schwarzhöhe**: jeder Pfad ausgehend von **einem gegebenen Knoten** bis zu einem Blatt enthält **dieselbe Anzahl** an schwarzen Knoten

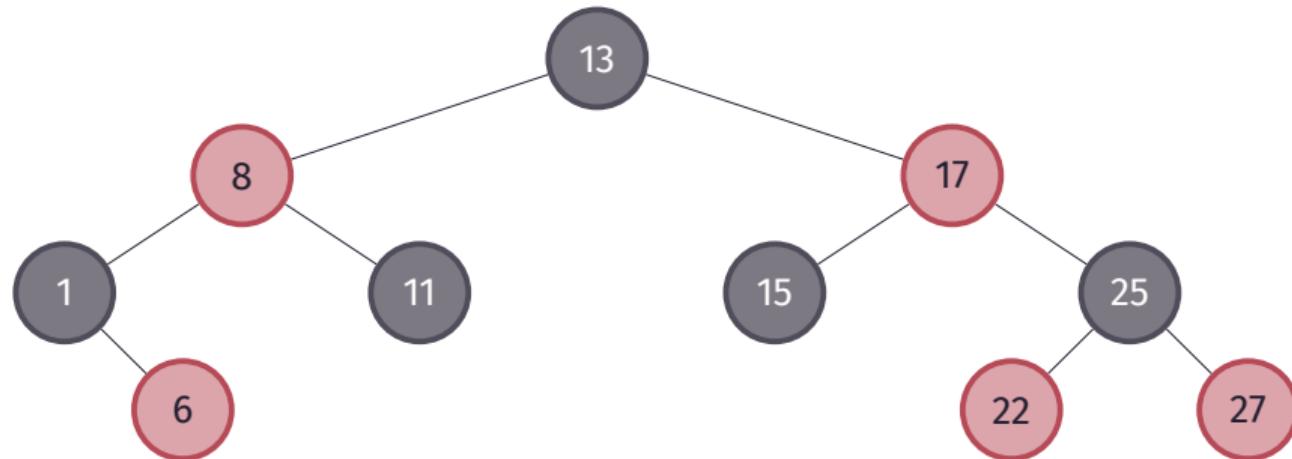
RED-BLACK-TREE

Operation - Einfügen:

1. Knoten wie in **binären Suchbaum** mit roter Farbe einfügen
2. Fall: **Elternteil schwarz**: Fertig
3. Fall: **Elternteil rot**, dann rebalancieren:
 - Fall: Onkel (von Elternteil) ist **schwarz**:
 - ▶ der neue Knoten ist das äußere Kind → rotiere **Großelternteil** (bspw. LL-Rotation)
 - ▶ der neue Knoten ist das innere Kind → rotiere **Großelternteil und Elternteil** (bspw. LR-Rotation)
 - ▶ färbe Großelternteil und Elternteil passend
 - Fall: Onkel (von Elternteil) ist **rot**: → färbe Elternteil und Onkel **schwarz** sowie Großelternteil **rot**, falls es nicht die Wurzel ist

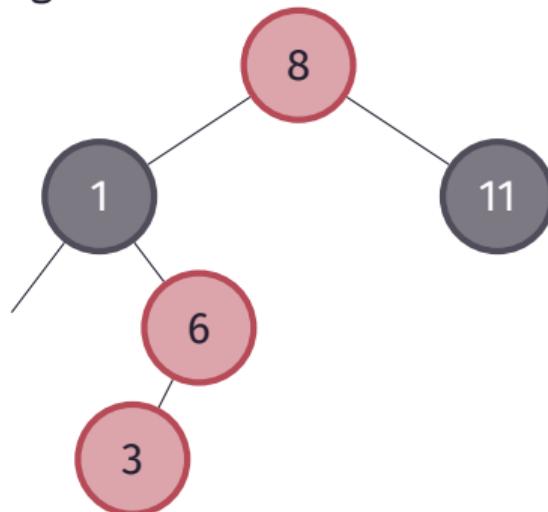
RED-BLACK-TREE

Betrachte das folgende Beispiel:



REBALANCING RED-BLACK TREE (I)

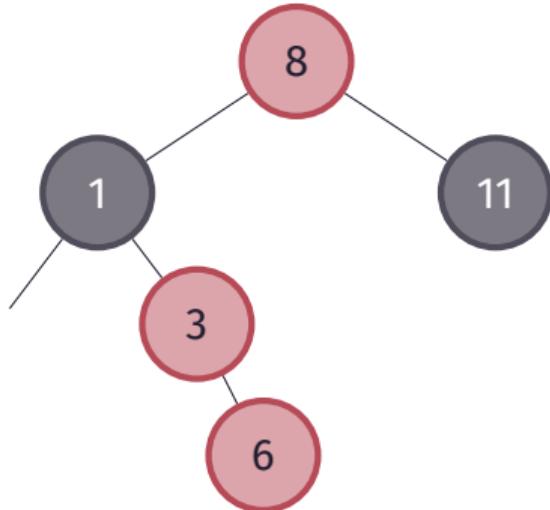
1. Füge Task mit vruntime=3 ein:



→Rebalanciere →Onkel (von Elternteil) ist schwarz →es ist ein inneres Kind
→rotiere Elternteil und Großelternteil (**RL-Rotation**)

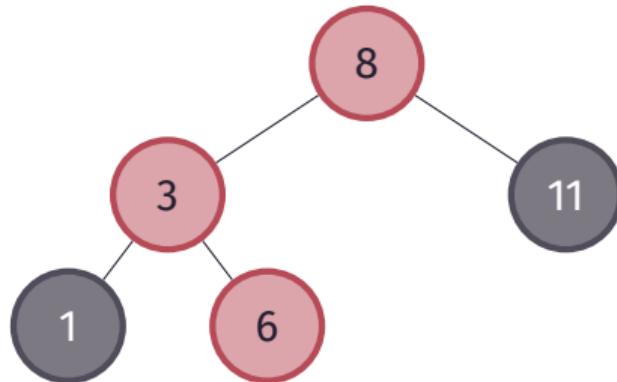
REBALANCING RED-BLACK TREE (II)

2.



REBALANCING RED-BLACK TREE (III)

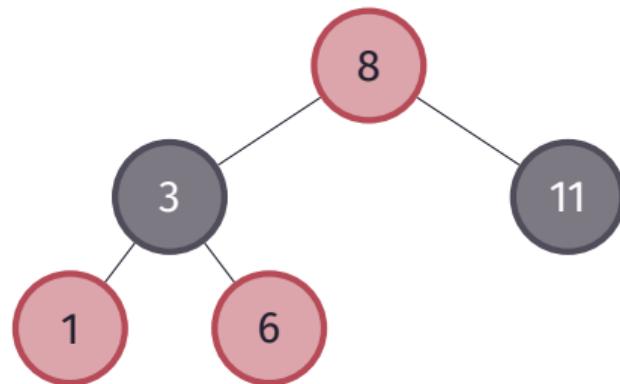
3.



→farbe Knoten, Elternteil und Großelternteil passend

REBALANCING RED-BLACK TREE (IV)

4.



AGENDA

1. Pthreads
2. Debugging
3. Linux Scheduling
4. Signal for Broken Pipe



SIGNALHANDLER FÜR BROKEN PIPE

broken_pipe.c:

- Eltern- und Kind-Prozess kommunizieren über eine **unnamed Pipe**
 - Kind-Prozess: führt **mehrere writes** aus (mit Text aus Datei foo)
 - Eltern-Prozess: führt **ein read** aus
- !** **Race-Condition:** Kind schreibt ggf. zu langsam und Eltern-Prozess beendet sein Lesen bereits zu früh

Definition (Race-Condition)

Wiederholung

Das Ergebnis einer Ausführung nebenläufiger Prozesse ist von der **zeitlichen Reihenfolge** der **Read- und Write-Operationen** abhängig.

Kind erkennt dies am Signal **?SIGPIPE**

Nachlesen unter: man 7 signal

SIGNALHANDLER FÜR BROKEN PIPE

Welche Operation **muss** der Elternprozess nach einem **fork** durchführen?

NAME

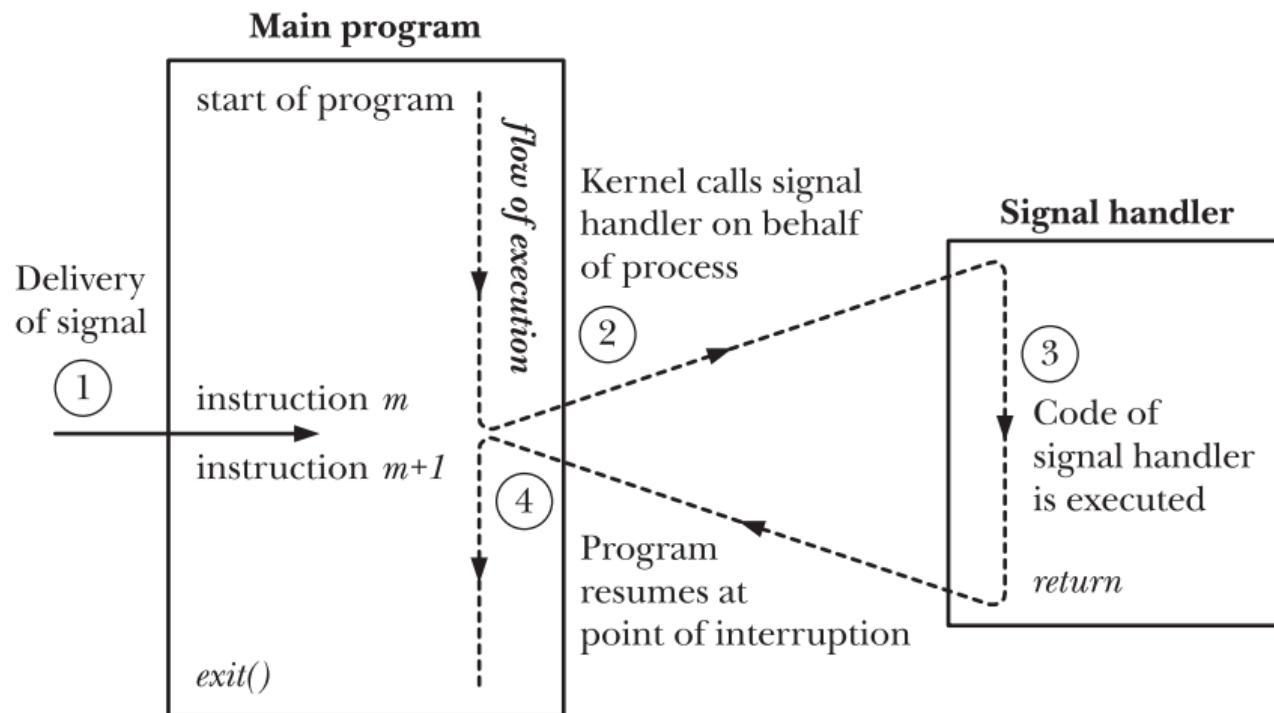
wait, waitpid, waitid - wait **for** process to change state

DESCRIPTION

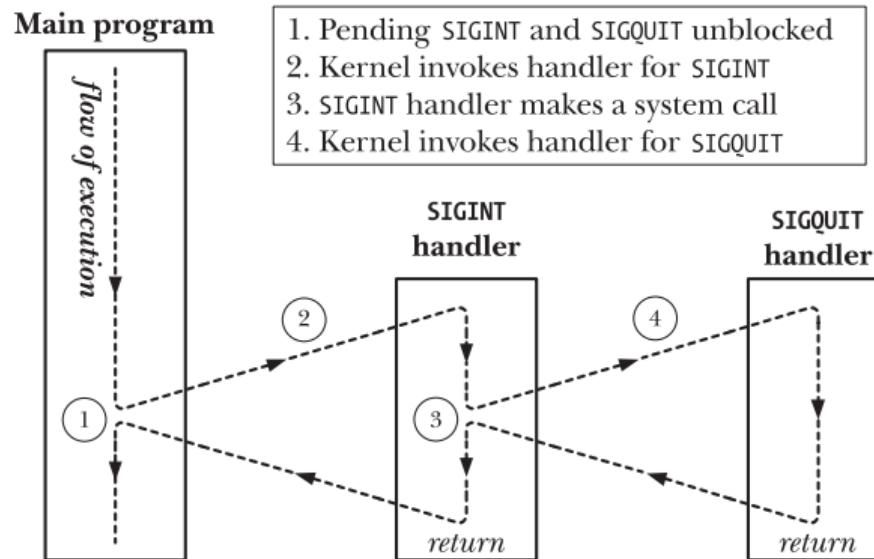
All of these system calls are used to wait **for** state changes in a child of the calling process, and obtain information about the child whose state has changed. A state change is considered to be: the child terminated; the child was stopped by a signal; or the child was resumed by a signal. [...]

Nachzulesen unter:  man 2 wait

SIGNALS



MULTIPLE SIGNALS



Mögliche Race Conditions:

- Nur Async-Signal-Safe Funktionen dürfen in Signalhandler aufgerufen werden! siehe: **man 7 signal-safety!**

SIGNALBEHANDLUNG

- jedes Signal hat eine **default Aktion**, häufig Terminierung des Prozesses (siehe `man 7 signal`)
- mittels `sigaction()` kann ein Signalhandler **registriert** werden oder ein Signal ignoriert werden
- mittels `sigprocmask()` kann ein Signal **temporär blockiert** werden
wird im Prozessleitblock (`task_struct`) gespeichert
- ein Signalhandler sollte **möglichst schnell** abgearbeitet werden
→ setzen einer `sig_atomic_t` Variable, die von der Hauptlogik des Programms ausgewertet wird

SIGNALHANDLER FÜR BROKEN PIPE

- Signal-Action mit `sigaction` registrieren

```
#include <signal.h>

int sigaction(int signum, const struct sigaction *act,
              struct sigaction *oldact);
```

- `struct sigaction` beschreibt die Action:

```
struct sigaction {
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask;
    int sa_flags;
    void (*sa_restorer)(void);
};
```

- `void (*sa_handler)(int)`: Funktion, die Signal behandelt
- `sigset_t sa_mask`: Menge anderer zu blockierender Signale setzen mit `sigemptyset`, `sigaddset`, ... (→ siehe: man `SIGSETOPS`)
- `int sa_flags`: Flags, die das Verhalten des Signals verändern

(Nachlesen & Beispiel unter: man 2 `sigaction`)

LITERATUREMPFEHLUNG

- *The Linux Programming Interface*, Michael Kerrisk

Kapitel:

- 21: Signals
- 29: Pthreads
- 43: Pipes and FIFOs

- man pages: Beispiel-Programme in:

- man 7 signal
- man sigaction
- man pipe
- man pthread_create

Noch Fragen?

Max Schrötter

Potsdam, 20. November 2025

schroetter@cs.uni-potsdam.de