

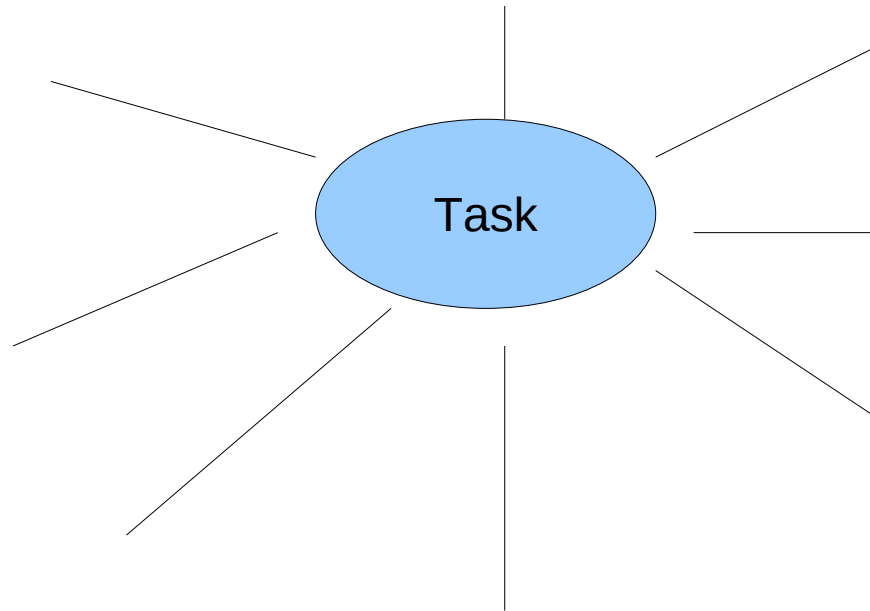
Tasks & Prozesse

PD Dr. Reinhard Bündgen
buendgen@de.ibm.com

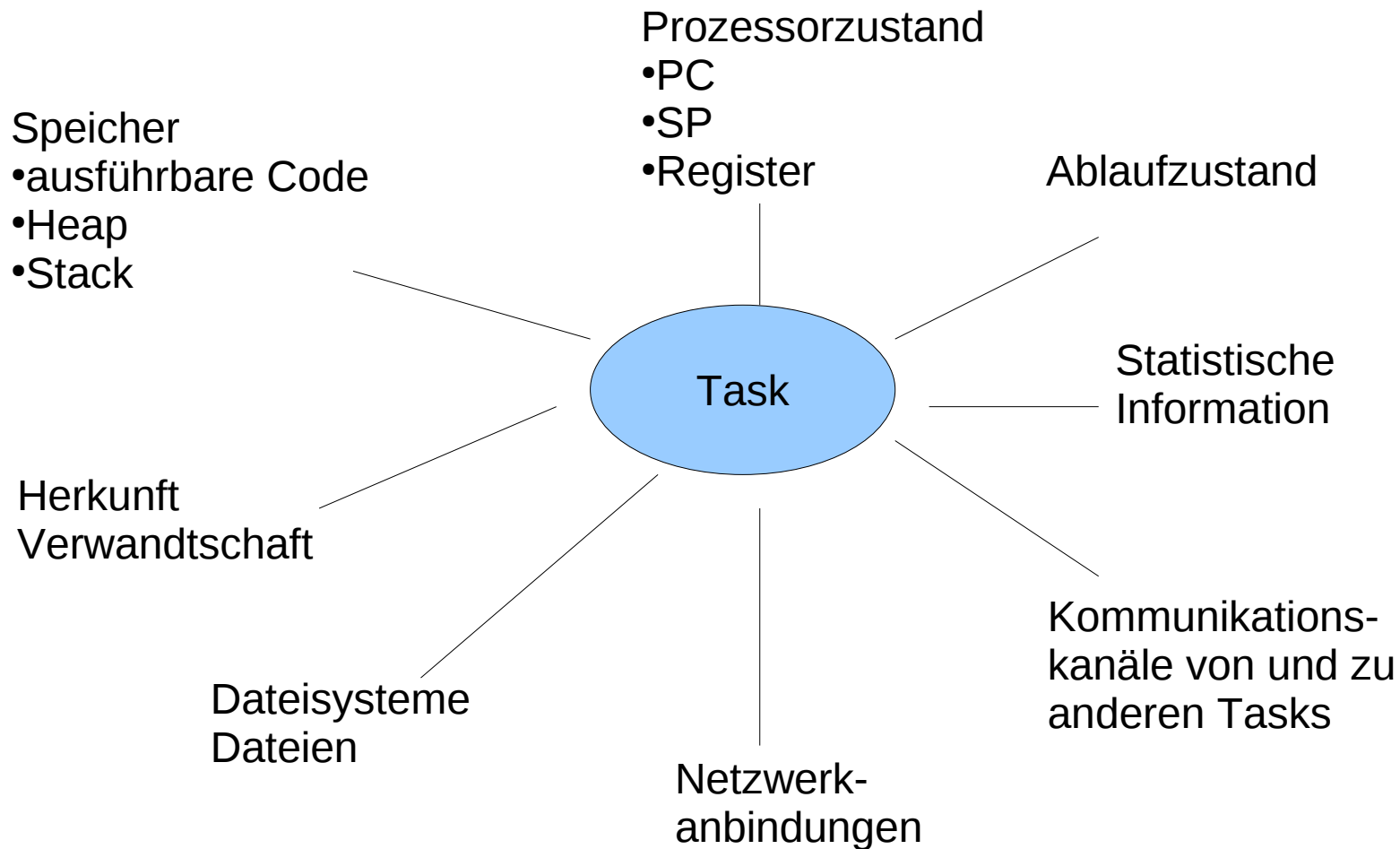
Was ist eine Task

- Ein sequentieller Arbeitsauftrag, der Prozessorzeit benötigt
- Beispiele in Unix
 - Prozesse (Instanzen laufender Programme)
 - Threads (light weight processes)
- Linuxkern: kennt nur Tasks, keine Prozesse oder Threads
- In einem multitasking Betriebssystem können mehrere Tasks (quasi) parallel ablaufen

Was gehört zu einer Task?



Was gehört zu einer Task?

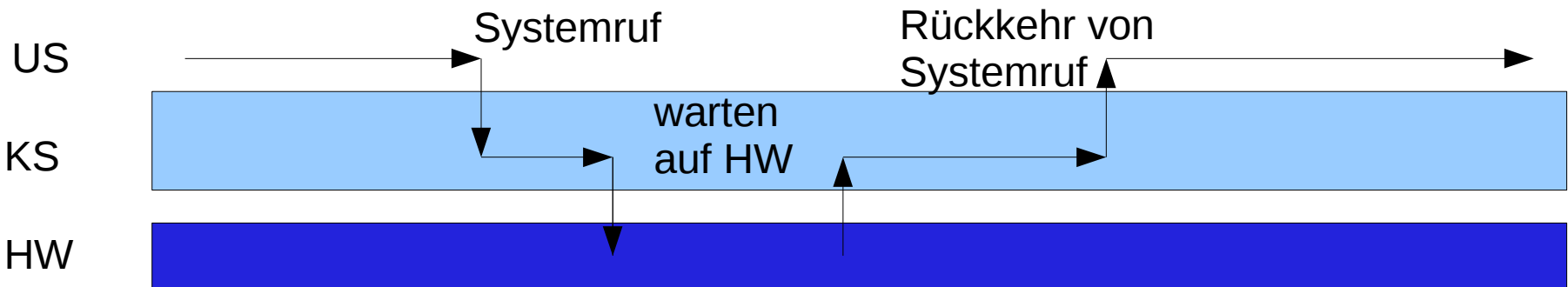


Laufzeiteigenschaften von Tasks

- Können Prozessor abgeben: warten
- Können Kontext wechseln: Nutzer <-> Kern
- sind unterbrechbar
 - Unterbrechung (Interrupt): temporärer Einschub wichtiger Aufgaben
- sind präemptif (preemptive)
 - können unterbrochen werden, ohne sofort nach der Unterbrechung wieder die CPU zu erhalten
 - z. B. durch Timerinterrupt & Zeitscheibenende

Befugnisse von Tasks

- Kerneltasks (kernel threads)
 - laufen im Kern Adressraum
 - haben Kernprivilegien
- Prozesse & Threads (Nutzertasks)
 - laufen in eigenem Adressraum ($0 \dots 2^{\text{Wortbreite}} - 1$ Bytes)
 - haben nur Nutzerprivilegien
 - bei Anforderungen die Befugnisse übersteigen: Systemruf
 - temporärer Kontextwechsel in den Kern
 - eventuell warten
 - anschließend Rückkehr in den Nutzerraum

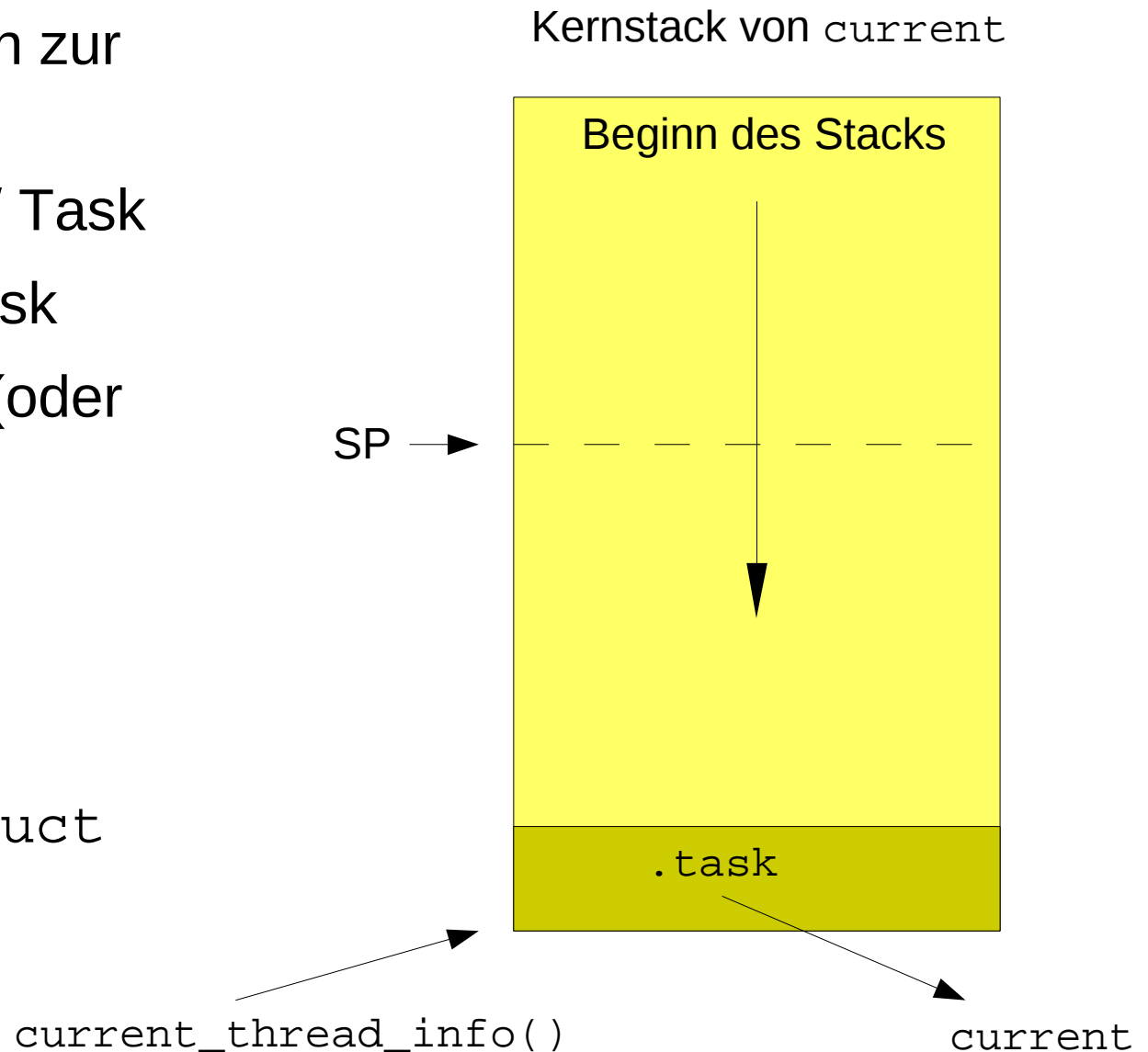


Identifizierung von Tasks im Kern

- Prozessdescriptor vom Typ `struct task_struct`
 - definiert in `include/linux/sched.h`
 - enthält alle wichtigen Daten zu einer Task (~1,7kB für 32bit Architekturen)
 - alloziert via slab/slub allocator
 - aktuelle Task: `current`
- `struct thread_info`
 - definiert in `include/asm/thread_info.h`
 - „zeigt“ auf Stack & `task_struct`
 - aktuelle task: `current_thread_info()`
- Prozessidentifikator (PID) vom Typ `pid_t`:
 - früher max. 32768 pids, jetzt default
 - Erhöhung via `/proc/sys/kernel/pid_max` möglich

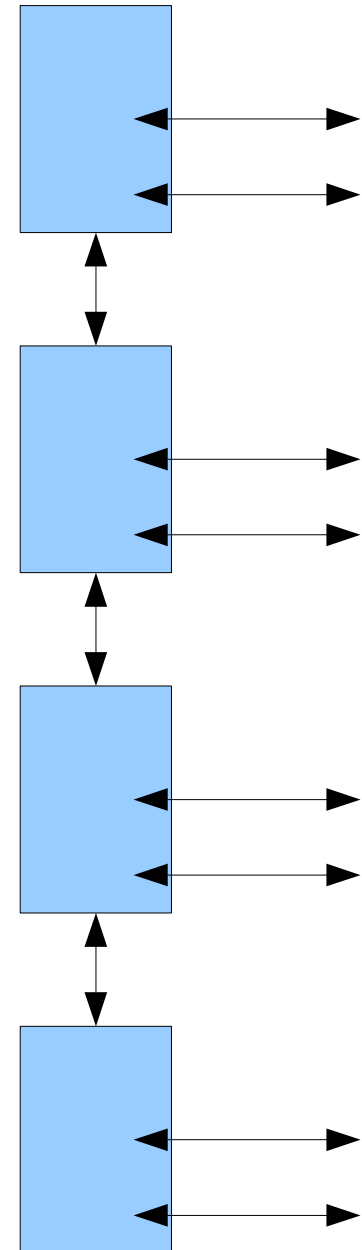
Kernstack & Thread_Info

- Stack, der Task im Kern zur Verfügung steht
 - früher 8 bzw 16 kB / Task
 - neu 4 bzw 8 kB / Task
 - enthält am unteren (oder oberen) Ende `thread_struct`
- `thread_struct`
 - architekturabhängig
 - zeigt auf `task_struct`
 - `exec_domain`
 - CPU Referenz
 - preempt count



Task Struktur Verkettungen

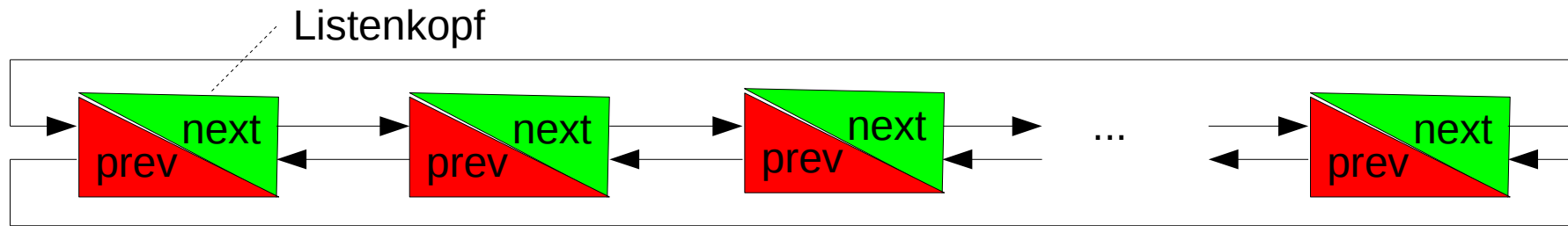
- doppelt verkettete Listen
 - tasks
 - ptrace_children, ptrace_list
 - children, sibling
 - cpu_timers
- Einzelzeiger
 - real_parent
 - parent
 - group_leader



Exkurs: Listen im Kern I

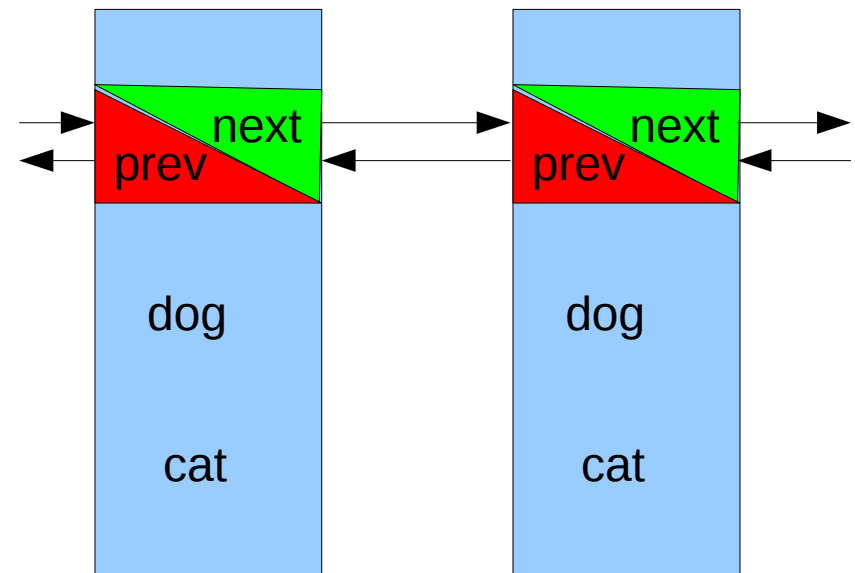
- doppelt verkettete Listen: `<linux/list.h>`

```
struct list_head {  
    struct list_head *next, *prev;  
};
```



z. B.

```
struct my_struct {  
    struct list_head mylist;  
    long dog;  
    void *cat;  
};
```



Exkurs Listen im Kern II

- Initialisieren
 - `struct my_struct *p;`
 - `INIT_LIST_HEAD(&p->mylist);` oder
 - `p->mylist=LIST_HEAD_INIT(p->mylist)`
- Operationen ($O(1)$)
 - `list_add(struct list_head *new,
 struct list_head *head);`
 - `list_add_tail(*new, *head)`
 - `list_add_rcu(*new, *head)`
 - `list_del(*entry)`
 - `list_empty(*head)`
 - ...

Exkurs: Listen im Kern III

- Element einer Liste

```
list_entry(list_head_ptr, struct_type, member_name)
```

- Traversieren von Listen

```
struct list_head *p; struct my_struct *my;
```

```
list_for_each(p, mylist) {
```

```
    /* p zeigt auf ein Listenelement */
```

```
    my=list_entry(p, struct my_struct, mylist)
```

```
    /* my zeigt auf die Struktur. die p enthält */
```

```
}
```

- ähnlich `list_for_each_prev()`

Traversieren von Tasklisten

- `struct task_struct *task; struct list_head *list;`

- Traversiere alle Kinder einer Task

```
list_for_each(list, &current->children.sibling) {  
    task=list_entry(list, struct task_struct, sibling);  
    /* task is now one of current's children */  
}
```

- Traversiere all Vorfahren einer Task

```
for(task=current; task!=init_task; task=task->parent) ;
```

- Nächste Task

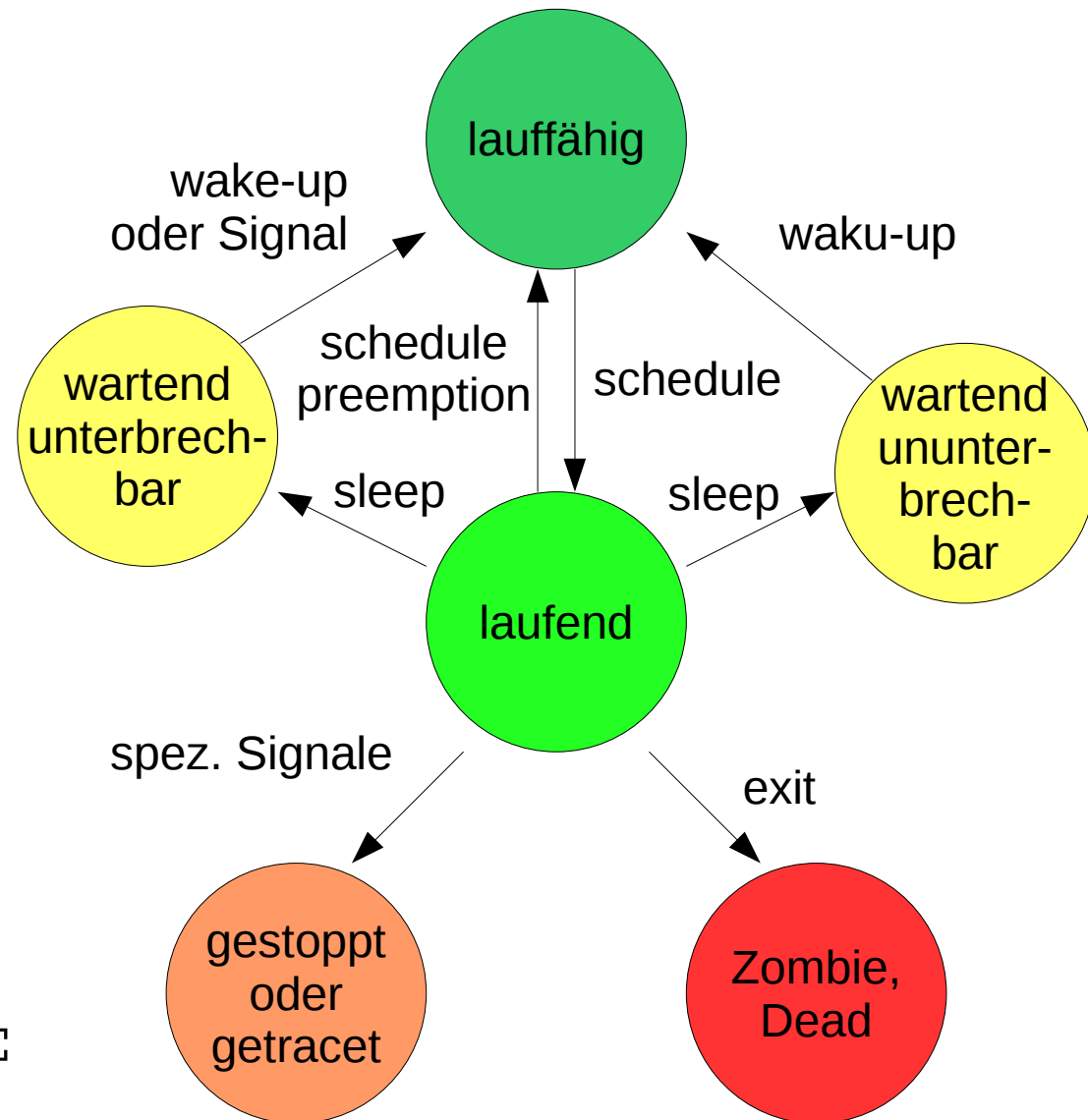
```
list_entry(task->task.next, struct task_struct, tasks)
```

- Traversiere alle Tasks

```
for_each_process(task) {  
    printk(„%s[%d]\n“, task->comm, task->pid);  
}
```

Task Zustände

- lauffähig: `TASK_RUNNING`
- wartend
 - & kann Signale empfangen: `TASK_INTERRUPTIBLE`
 - & kann keine Signale empfangen: `TASK_UNINTERRUPTIBLE`
 - 2.6.14: `TASK_NONINTERACTIVE`
- angehalten: `TASK_STOPPED`
- ge-trace-t: `TASK_TRACED`
- beendet durch `exit`: `EXIT_ZOMBIE`
- fertig zum Aufräumen: `EXIT_DEAD`



Task-Erzeugung

- z. B. Programmstart in Unix

```
pid=fork();  
if (pid == 0) {  
    /* execute child code */  
    /* program start: */  
    execv(...);  
} else if ( pid > 0 ) {  
    /* execute parent code */  
else {  
    /* error */  
}
```

- `fork()` kopiert Prozess
- `exec(v(e))()` startet neues Programm in frisch kopierten Prozesskontext
- d.h. bei fork-exec Kombination ist die Kontext Kopie durch `fork()` umsonst!
- moderne Unixe:
 - copy on write (COW)
 - kopiere Seiten erst bei Schreibzugriff

Taskerzeugung im Kern: do_fork()

- `fork()`, `vfork()`, `clone()` werden auf `do_fork()` in `linux/kernel/fork.c` abgebildet
- `do_fork()`
 - Parameter `clone_flags` beschreibt welche Betriebsmittel von Eltern und Kind Task gemeinsam genutzt werden.
 - alloziert neue pid
 - ruft `copy_process()` auf
 - ruft `wake_up_new_task()` auf.

Clone Flags

CLONE_VM set if VM shared between processes

CLONE_FS set if fs info shared between processes

CLONE_FILES set if open files shared between processes

CLONE_SIGHAND set if signal handlers and blocked signals shared

CLONE_PTRACE set if we want to let tracing continue on the child too

CLONE_VFORK set if the parent wants the child to wake it up on mm_release

CLONE_PARENT set if we want to have the same parent as the cloner

CLONE_THREAD Same thread group?

CLONE_NEWNS New namespace group?

CLONE_SYSVSEM share system V SEM_UNDO semantics

CLONE_SETTLS create a new thread local storage (TLS) for the child

CLONE_PARENT_SETTID set the TID in the parent

CLONE_CHILD_CLEARTID clear the TID in the child

CLONE_DETACHED Unused, ignored

CLONE_UNTRACED set if the tracing process can't force CLONE_PTRACE on this clone

CLONE_CHILD_SETTID set the TID in the child

CLONE_STOPPED Start in stopped state

Taskerzeugung in Kern: `copy_process()`

`copy_process()` in `linux/kernel/fork.c`

- Plausibilitäts-Checks, Sicherheits-Checks
- `dup_task_struct()`
- check resource limits
- Initialisiere Statistik, Timers
- kopiere diverse Strukturen
- `sched_fork()`
 - CPU Zuweisung,
 - `TASK_RUNNING` ohne Einfügen in run-queue
 - Zeitscheibenallokation
- ...
- return Zeiger auf neue Task

Prozess vs. Threads

- Linux hat keinen expliziten Support für Threads
 - im Gegensatz zu Solaris und Windows
- In Linux werden sowohl Threads als auch Prozesse auf Tasks abgebildet
- Threads werden mit dem Systemruf `clone()` erzeugt
 - Clone hat Parameter, der beschreibt welche Betriebsmittel gemeinsam genutzt werden
 - `clone_flags`: `CLONE_VM` | `CLONE_FS` | `CLONE_FILES` | `CLONE_SIGHAND`
- Siehe auch „man clone“

Task Termination

- Prozesse rufen den `exit()` Systemruf auf
 - evtl implizit ans Ende des Programms einkompiliert/gelinkt
- `do_exit()` aus `kernel/exit.c`:
 - set `PF_EXITING` in `current->flag`
 - lösche Timer: `del_timer_sync()`
 - Accounting
 - Gib AR frei: `__exit_mm`
 - lösche Semaphore: `exit_sem()`
 - räume Taskressourcen auf: `__exit_files()`, `__exit_fs()`, `exit_namespace()`, ...
 - Übergabe des Exitcodes
 - Signalisiere Elternteil in `exit_notify()`:
 - `current->state=EXIT_ZOMBIE`
 - rufe Scheduler auf: `schedule()`

Waisenkinder

- Was passiert mit einem Prozess, dessen Elternprozess aufhört?
- Wird er auf ewig Zombie bleiben?
- Lösung: „Reparenting“
- `do_exit()` -> `exit_notify()` -> `forget_original_parent()`
 - Suche nach neuem Elternprozess:
 - in gleicher Threadgruppe oder
 - Init Prozess
 - Reparenting aller Kinderprozesse

Löschen des Task Descriptors

- nach `do_exit()` existiert die `task_struct` noch
- sie wird evtl. noch vom Elternteil benötigt:
 - die `wait()` Funktionsfamilie gibt `pid` & Exitcode des abgeschlossenen Kindprozesses zurück
- `sys_wait4()` Systemruf in `kernel/exit.c`
 - `do_wait()` -> `wait_task_zombie()` -> `release_task()`
 - dekrementiere Zahl der Prozesse eines Users
 - `__exit_signal()`,
 - `__unhash_process()`: lösche task aus `pidhash`
 - evtl: löschen aus `ptrace` Liste
 - `put_task_struct()`
 - gib `uid` frei
 - gib Kernstack Seiten frei
 - dealloziere `task_struct`

Kern Threads

- werden benutzt um Kernoperationen im Hintergrund laufen zu lassen
 - Kernprivilegien
 - kein eigener Adressraum
 - keine Kontextwechsel in Nutzerraum
 - schedulable & präemptif
- Beispiele: pdfflush, ksoftirqd
- Erzeugung von Kern Threads:
 - `int kernel_thread(int (*fn)(void *), void * arg, unsigned long flags);`
 - `do_fork(flags | CLONE_VM | CLONE_UNTRACED, 0, ®s, 0, NULL, NULL);`
 - `CLONE_KERNEL = (CLONE_FS | CLONE_FILES | CLONE_SIGHAND)`

Systemstart: initiale Prozesse

- Prozess 0 – „swapper process“
 - kernel thread wird in `start_kernel()` (in `init/main.c`) von Grund aufgebaut
 - ...
 - `rest_init()`
 - `kernel_thread(init, NULL, CLONE_FS | CLONE_SIGHAND);`
 - `schedule();`
 - `cpu_idle();`
- `init()` in `init/main.c` hat `pid=1`
 - ...
 - `run_init_process("/sbin/init");`
 - `execv(...);`