Process management

What is a process?

How Linux encapsulates a running program and its environment

Process attributes

Family relationships

- All processes have a parent
- They may have siblings or children

init is the ultimate ancestor process

Process attributes

Resources

- Address space and memory mappings
- Open files
- Namespaces
- IPC resources
- ...

Process attributes

Scheduling state

- Scheduling class
- Priorities
- Resource usage

Credentials

- Identity
- Privileges (capabilities)

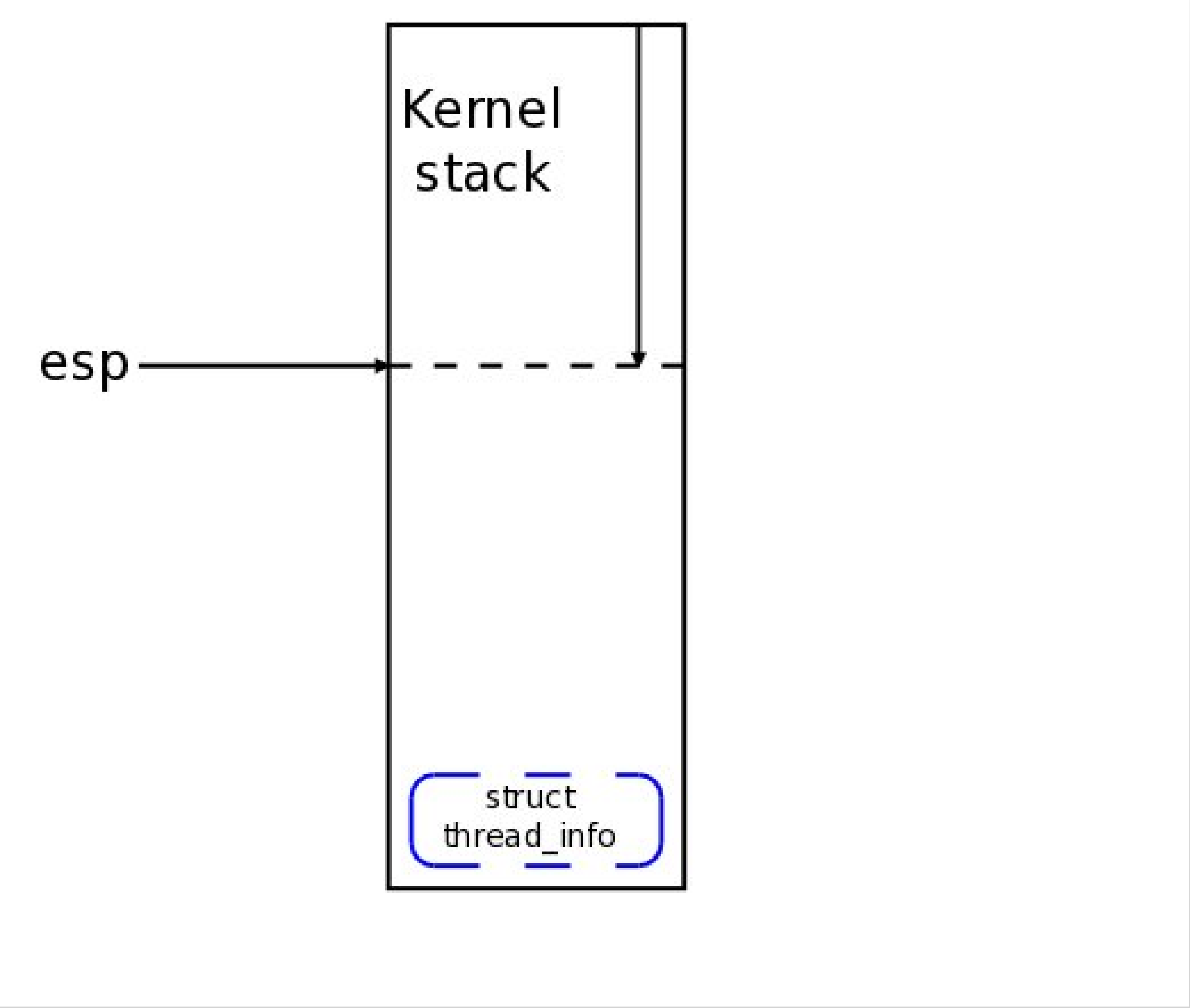
...

struct thread_info

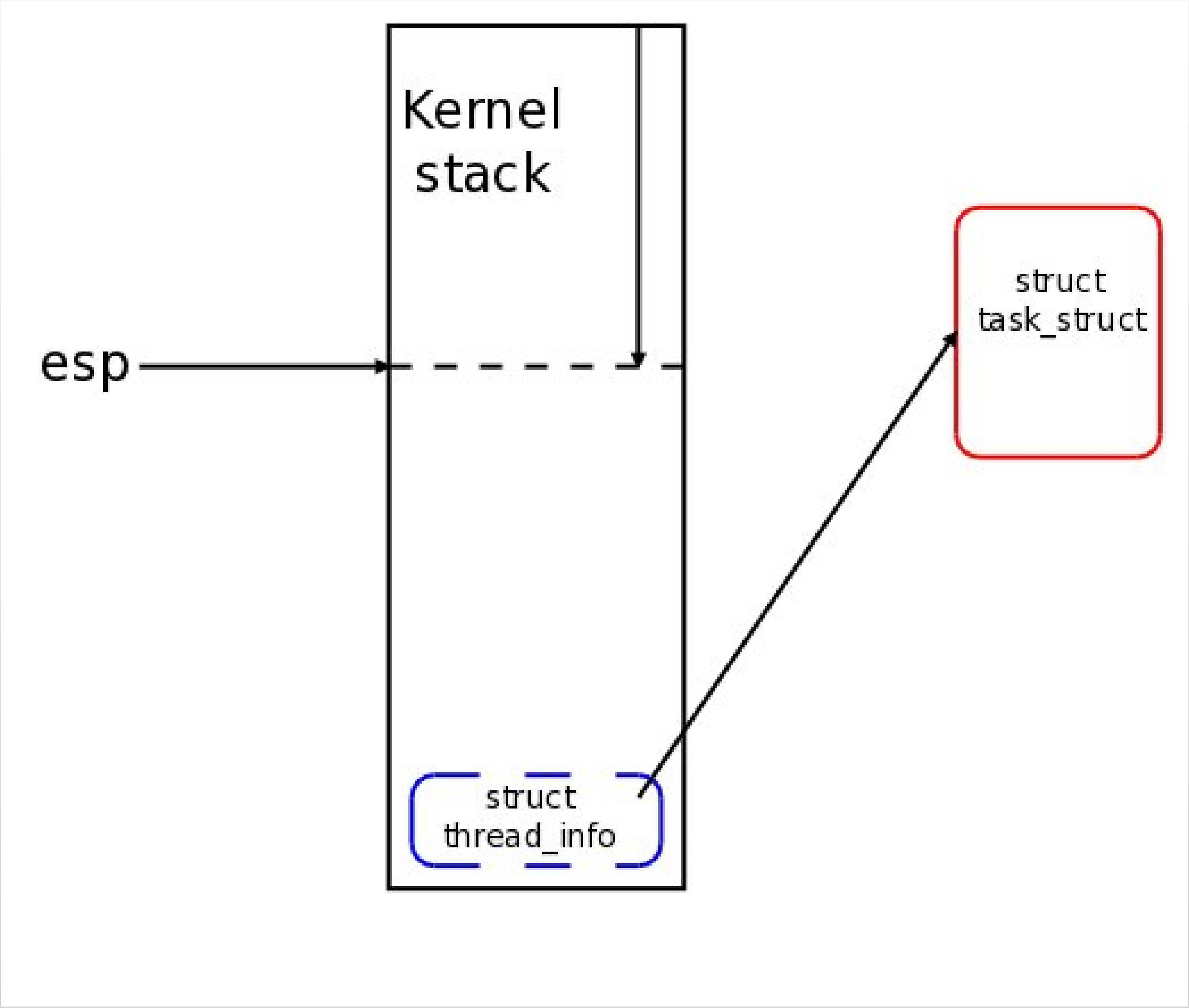
```
struct thread_info {
    struct task_struct  *task;
    struct exec_domain  *exec_domain;
    __u32        flags;
    __u32        status;
    __u32        cpu;
    int        preempt_count;
    /* ... */
}

(arch/x86/include/asm/thread_info.h)
```

Finding struct thread_info



struct task_struct



Getting the task_struct

The current() macro

- Masks bottom bits from stack pointer
- Casts to thread_info
- Return info->task

struct task_struct

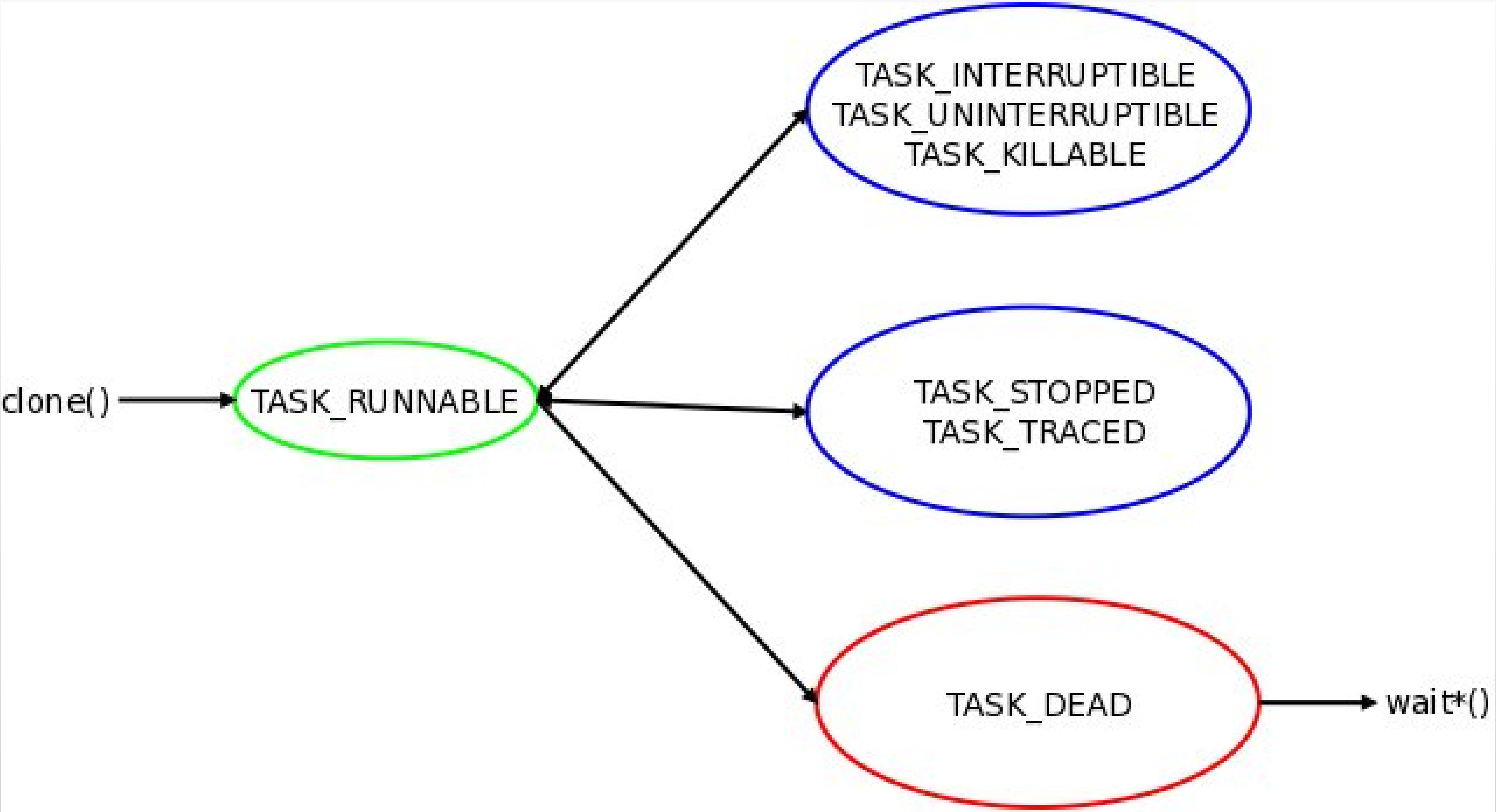
- include/linux/sched.h

What's in the task_struct

state
priorities
CPU mask
mm pointer
pid and tgid
parent pointers
child process list
sibling process list

ptrace() information
usage statistics
credentials
filesystem info
open files
namespaces
signal info
audit context info
...

Process states



Classic fork()

The way to make a new process

```
child_pid = fork();
```

Returns twice

- Once to parent (returning child pid)
- Once to child (returning zero)

The two processes are independent copies

Inside fork()

Copy task_struct structure
Check resource limits
Copy:

Credentials	Semaphores
Open files	Filesystem info
Signal handlers	Address space
Namespaces	Block I/O context
...	

Tracepoint [sched_process_fork]
Start new process

clone()

Sometimes processes want to share
- Memory
- Open files
- ...

The clone() system call allows sharing
- Many flags to select resources to share
- It's how threads are done on Linux
- Sharing can be undone with unshare()

fork()

...is really just a wrapper around clone()

Threads

Threads on Linux are just processes

clone() flags used to share:
- memory, files, signal handlers, ...

CLONE_THREAD flag
- Sets thread group ID
- Used for signal delivery

Per-thread stack created too

Sleeping (blocking)

There are three sleep states

- TASK_INTERRUPTIBLE
- TASK_UNINTERRUPTIBLE
- TASK_KILLABLE

Simple sleeping

```
SYSCALL_DEFINE0(pause)
{
    current->state = TASK_INTERRUPTIBLE
    schedule();
    return -ERESTARTNOHAND;
}
```

Old-style sleeping

```
/* Don't do it this way */
wait_queue_head_t waitq

while (need_to_sleep)
    sleep_on(&wait);


/* ...somewhere else... */
need_to_sleep = 0;
wake_up(&waitq);
```

Better sleeping

```
wait_event(&waitq, condition);
wait_event_interruptible(&waitq, condition);
/* A vast number of variations */


/* ...or... */
prepare_to_wait(&waitq, &wqe, state);
if (! condition)
   schedule();
finish_wait(&waitq, &wqe);
```

exit()

- When a process is done
  - Reset signals
  - Release memory
  - Tracepoint [sched_process_exit]
  - Release other resources
  - Notify parent
  - task->state = TASK_DEAD;
  - schedule();

wait*()

- Cleanup dead processes and return info
  - Tracepoint [sched_process_wait]
  - task->state = TASK_INTERRUPTIBLE
  - check for dead children
  - sleep if none
  - release task structure
  - return information

Signals

Signals can
- Change process state
- Force execution of signal handler
- Interrupt system calls

It's complex stuff!

task_struct fields

struct signal_struct *signal;

Thread-group shared information (including pending signals)

struct sigpending *pending;

Private pending signals

struct sighand_struct *sighand;

Signal handling information

sigset_t blocked;

List of blocked signals

Signal tracepoints

signal_generate

When a signal is queued for a process

signal_deliver

When a signal is delivered to a process

signal_overflow_fail

Realtime signal lost

signal_lose_info

Associated information lost

Kernel threads

- Special processes for kernel tasks
  - Run in kernel mode
  - Have no user-mode address space
- Look for [name in brackets]
- Tracepoints
  - sched_kthread_stop
  - sched_kthread_stop_ret

Control groups

A mechanism for grouping processes

Cgroups have:

- A position in a hierarchy
- A list of processes
- A set of attached “subsystems”
- A mounted control filesystem
  - Under /sys/fs/cgroup by default

Cgroup subsystems

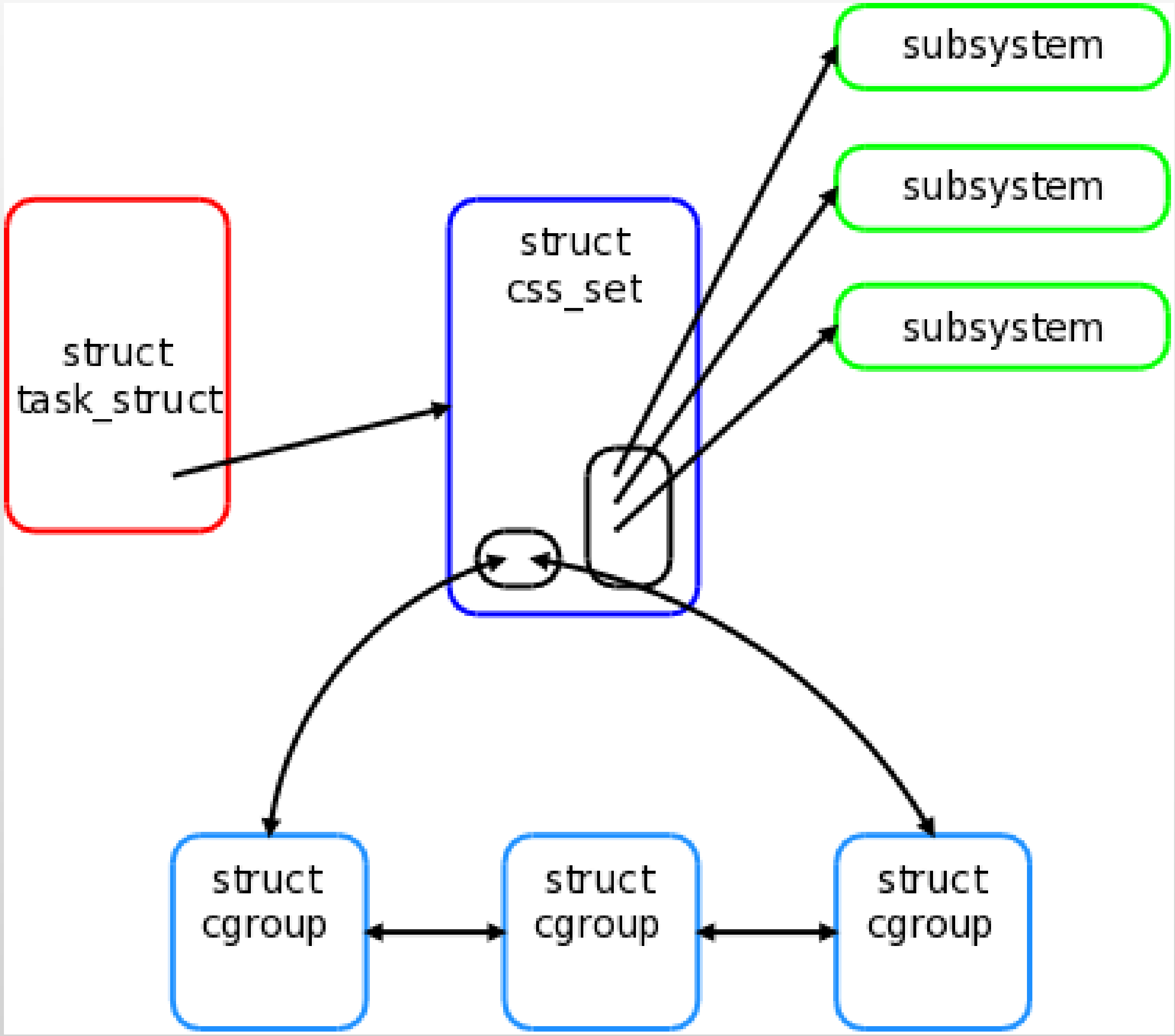
A means for affecting process behavior

- CPU affinity
- CPU scheduling
- Block I/O bandwidth control
- Namespaces
- User-space (systemd)

Systemd cgroup hierarchy

```
`-- system
    |-- abrtd.service
    |-- atd.service
    |-- avahi-daemon.service
    |-- backups.mount
    |-- backups-vena.mount
    |-- console-kit-daemon.service
    |-- crond.service
    |-- cups.service
    |-- dbus.service
    |-- fsck@.service
    |-- getty@.service
    |   |-- tty2
    |   |-- tty3
    |   |-- tty4
    |   |-- tty5
    |   `-- tty6
```

Associating cgroups and tasks



Cgroup notes

Code in kernel/cgroup.c
- No tracepoints in cgroup code

See a process's info under
- /proc/*pid*/cgroup

Under /sys/fs/cgroup you can
- Create new groups
- Query membership
- Move processes between groups

Scheduling

CPU scheduling has two goals

- Maximize system throughput
- Minimize latency

They are often contradictory!

Once upon a time

We had the O(1) scheduler

- Fast runqueue management
- Lots of interactivity heuristics

Problems:

- Difficult code
- Poor interactivity

Completely fair scheduling

- The core idea:
  - Dump all the heuristics
  - If there are N runnable processes
    - Each get 1/N of the available CPU time

Implementation

```
struct sched_entity {
    struct rb_node     run_node;
    struct list_head group_node;
    unsigned int    on_rq;

    u64         exec_start;
    u64         sum_exec_runtime;
    u64         vruntime;
    u64         prev_sum_exec_runtime;
    /* ... */
}
```

vruntime

- The amount of CPU time the process has used
  - ...sort of

- Time to pick a new task to run?
  - Grab the runnable task with the smallest vruntime

Preemption

When should tasks be preempted?

- Too rarely: bad latencies
- Too often: bad throughput

CFS approach: try to bound latency

- /proc/sys/kernel/sched_latency_ns
- The period in which all tasks should run
- Default: 6ms * (1 + log2(ncpus))

How long should a process run?

"Time slice" is dynamic:

- sched_latency_ns / (# running tasks)

...but only to a point

- sched_min_granularity_ns
- The smallest a time slice will go
- Default: 750μs

Thus:

- Latency will suffer as load gets high

Scheduling classes

- The scheduler supports multiple classes
  - A strict priority arrangement
- Three classes supported now:
  - Realtime (both RR and FIFO)
  - CFS (SCHED_OTHER)
  - Batch

pick_next_task()

To choose the next process to run:

```
for_each_class(class) {
    p = class->pick_next_task(rq);
    if (p)
        return p;
}
```

Wakeups

- task->state = TASK_RUNNING
- Adjust vruntime
  - Adjust to new runqueue minimum
  - Possibly give credit for sleep
- Put the task into the run queue
- Possibly preempt running task
  - /proc/sys/kernel/sched_wakeup_granularity_ns
  - 1ms * (1 + log2(ncpus))

Why struct sched_entity?

...instead of storing the info in the task_struct directly?

Why struct sched_entity?

Control groups

Imagine:

- Alice runs one movie player
- Bob runs 9 compilers
- ...
- Alice gets grumpy

Group scheduling

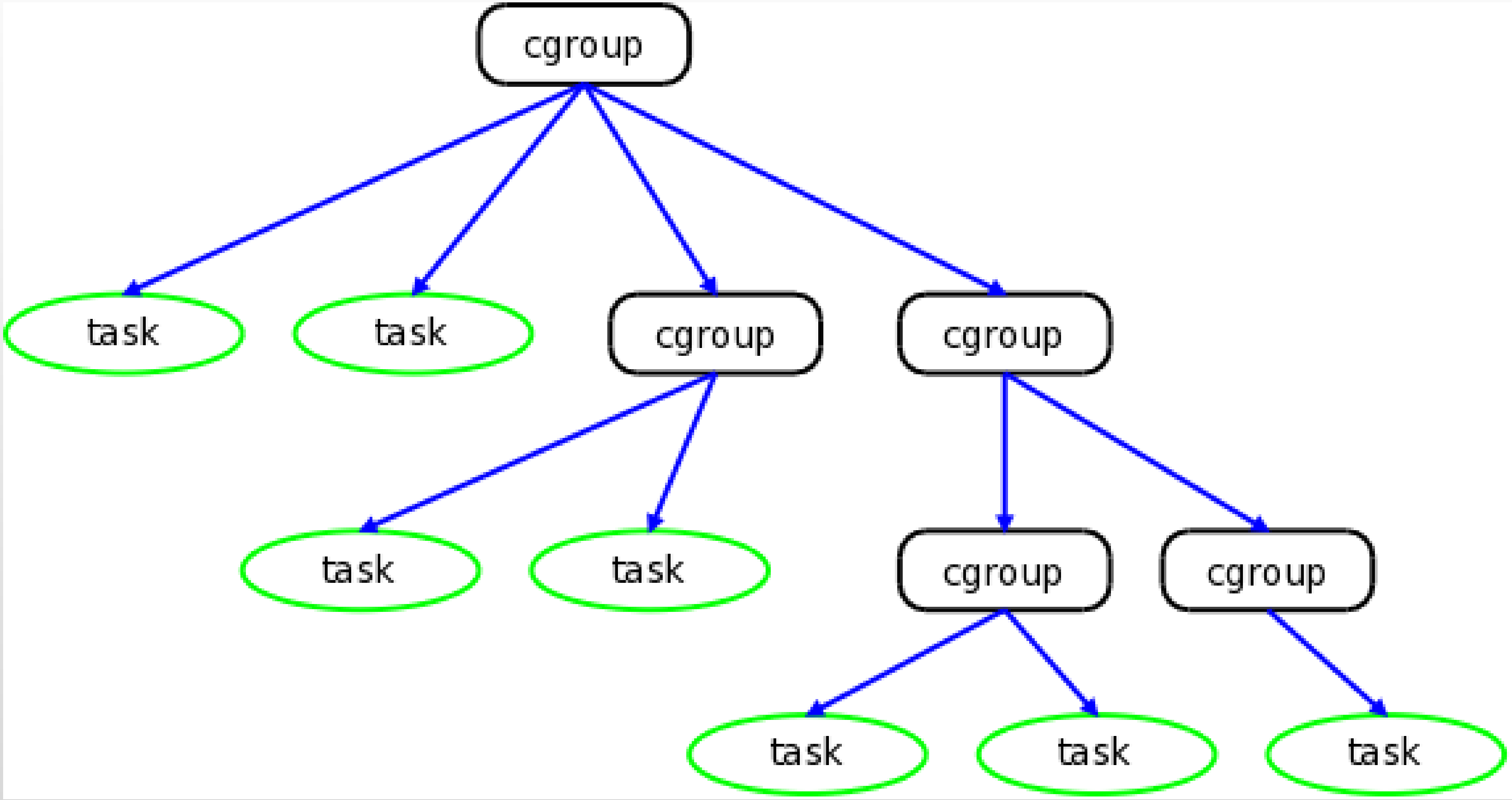
Give Alice and Bob their own groups

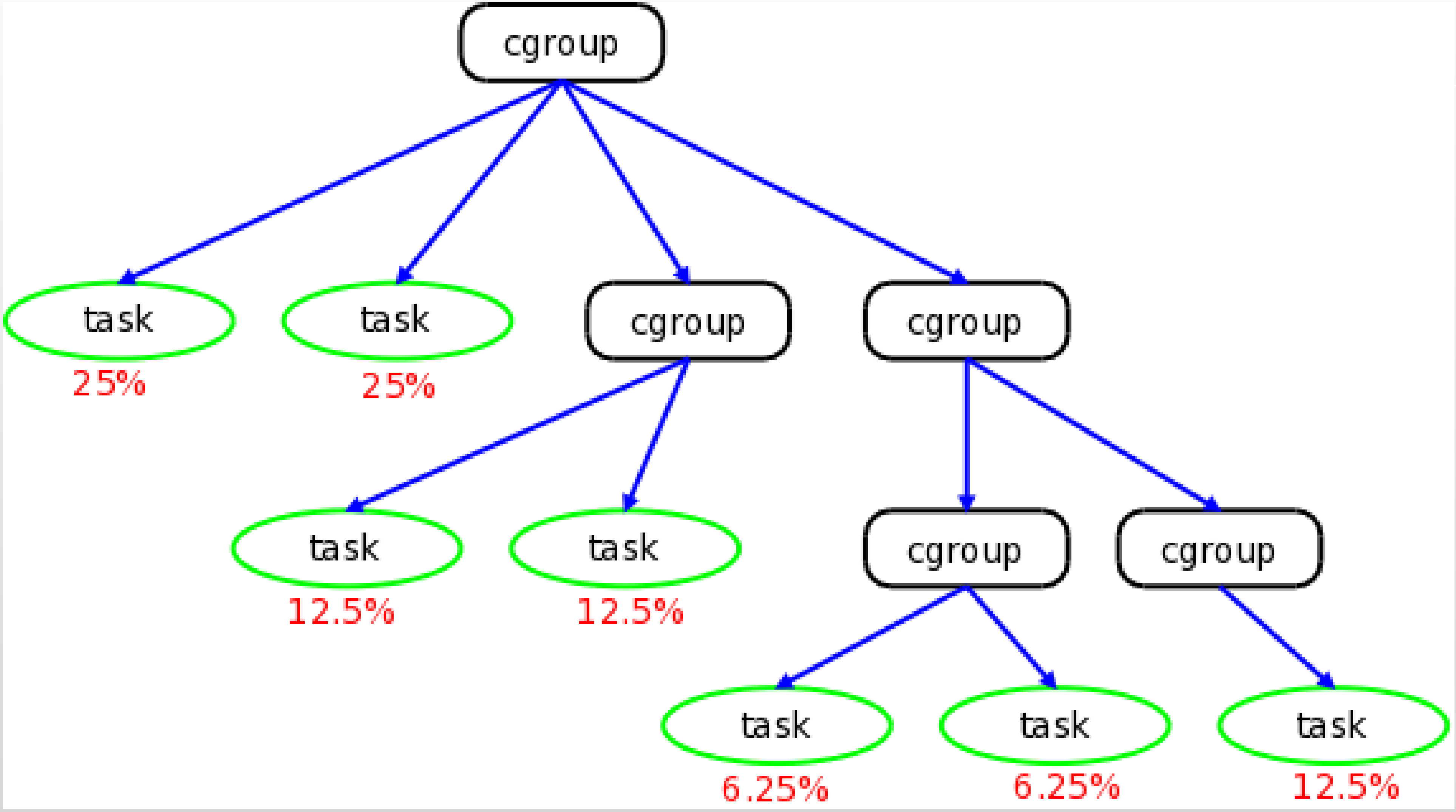
The groups are scheduled

- 50% for each

Each user's processes compete

- ...within their group


cgroup
task
task
cgroup
cgroup
task
task
cgroup
cgroup
task
task
task



cgroup
task
25%
task
25%
cgroup
cgroup
task
12.5%
task
12.5%
cgroup
cgroup
task
6.25%
task
6.25%
task
12.5%

Balancing

The scheduler must:

- Keep approximately equal load on all CPUs
- Minimize migrations
  - Especially across NUMA nodes
- Respect CPU affinity
- Respect power management goals

It's complicated!

Scheduler tracepoints

sched_migrate_task
sched_switch
sched_wakeup
sched_wakeup_new
...

Credentials

Who does a process represent

What special capabilities does it have?

All found in task->cred

struct cred

```
struct cred {
	uid_t		uid;
	gid_t		gid;
	uid_t		suid;
	gid_t		sgid;
	uid_t		euid;
	gid_t		egid;
	uid_t		fsuid;
	gid_t		fsgid;
	unsigned	securebits;
	kernel_cap_t	cap_inheritable;
	kernel_cap_t	cap_permitted;
	kernel_cap_t	cap_effective;
	kernel_cap_t	cap_bset;
	void		*security;
	/* ... */
};
```

Capabilities

Finer-grained privileges

sort of

They include:

CAP_SYS_ADMIN

CAP_DAC_OVERRIDE

CAP_KILL

CAP_NET_BIND_SERVICE

CAP_SYS_RESOURCE

...

Capability sets

Effective

- Capabilities which can be used now

Permitted

- Capabilities which could be enabled

Inheritable

- Those which can be passed to other programs

Bounding

- The maximum anybody can have

Capability checks

Typical code

```
if (!capable(CAP_SOMETHING))
    return -EPERM;
```

Capability use is not traced

- But `PF_SUPERPRIV` is set in the process flags

Processes: related topics

Process address space
Resource accounting
Personalities
ptrace()
...

Questions?