Process Scheduling

Xiaoguang Wang

UIC COMPUTER SCIENCE

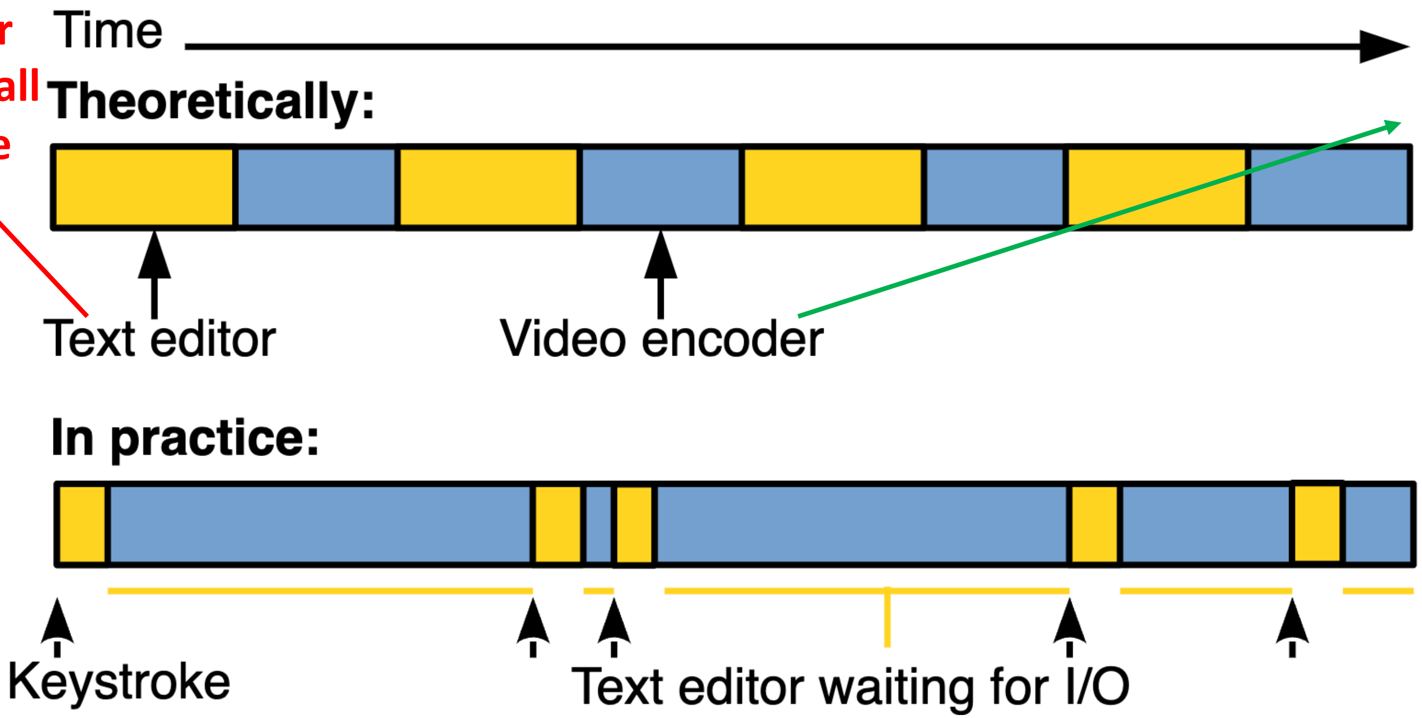
Recap: Linux CFS

Linux CFS does not use an absolute time-slice

- The time slice a process receives is a function of the load of the system (i.e., **a proportion of the CPU**)
- In addition, that time-slice is weighted by the **process priority**
- When a process **P** becomes runnable:
    - **P** will preempt the currently running process **C** if **P** consumes a smaller proportion of the CPU than **C**

Recap: example in Linux CFS

Text editor waits for I/O, consuming a small portion of CPU time



Video encoder batch processes data, consuming a larger portion of CPU time

Good interactive performance

Good background, CPU-bound performance

Recap: sched_entity

`task_struct` has scheduler-related fields.

```
/* linux/include/linux/sched.h */
struct task_struct {
    /* ... */
    const struct sched_class *sched_class; /* sched_class of this task */
    struct sched_entity       se; /* for time-sharing scheduling */
    struct sched_rt_entity    rt; /* for real-time scheduling */
    /* ... */
};
struct sched_entity {
    /* For load-balancing: */
    struct load_weight  load;
    struct rb_node      run_node;
    struct list_head    group_node;
    unsigned int        on_rq;

    u64                 exec_start;
    u64                 sum_exec_runtime;
    u64                 vruntime; /* how much time a process
                                   * has been executed (ns) */
```

Recap: scheduler class

`sched_class`: an abstract base class for all scheduler classes

```c
/* kernel/sched/sched.h */
struct sched_class {
  /* Called when a task enters a runnable state */
  void (*enqueue_task) (struct rq *rq, struct task_struct *p, int flags);
  /* Called when a task becomes unrunnable */
  void (*dequeue_task) (struct rq *rq, struct task_struct *p, int flags);
  /* Choose a next task to run */
  struct task_struct * (*pick_next_task) (struct rq *rq, struct task_struct *prev,
                                          struct rq_flags *rf);
  ... ...
};
```

CFS implementation

Four main components of CFS

- Time accounting
- Process selection
- Scheduler entry point: `schedule()`, `scheduler_tick()`
- Sleeping and waking up

Time accounting in CFS

Virtual runtime: how much time a process has been executed

```c
/* linux/include/linux/sched.h */
struct task_struct {
    /* ... */
    const struct sched_class *sched_class; /* sched_class of this task */
    struct sched_entity      se; /* for time-sharing scheduling */
    struct sched_rt_entity   rt; /* for real-time scheduling */
    /* ... */
};
```

```c
struct sched_entity {
    /* For load-balancing: */
    struct load_weight   load;
    struct rb_node       run_node;
    struct list_head     group_node;
    unsigned int         on_rq;

    u64                  exec_start;
    u64                  sum_exec_runtime;
    u64                  vruntime; /* how much time a process
                                    * has been executed (ns) */
```

Time accounting in CFS

Upon every timer interrupt, CFS accounts the task's execution time

```
/* linux/kernel/sched/fair.c */
/* scheduler_tick() calls task_tick_fair() for CFS.
 * task_tick_fair() calls update_curr() for time accounting. */
static void update_curr(struct cfs_rq *cfs_rq)
{
    struct sched_entity *curr = cfs_rq->curr;
    u64 now = rq_clock_task(rq_of(cfs_rq));
    u64 delta_exec;

    if (unlikely(!curr))
        return;

    delta_exec = now - curr->exec_start; /* Step 1. calc exec duration */
    if (unlikely((s64)delta_exec <- 0))
        return;

    curr->exec_start = now;
    /* continue in a next slide ... */
}
```

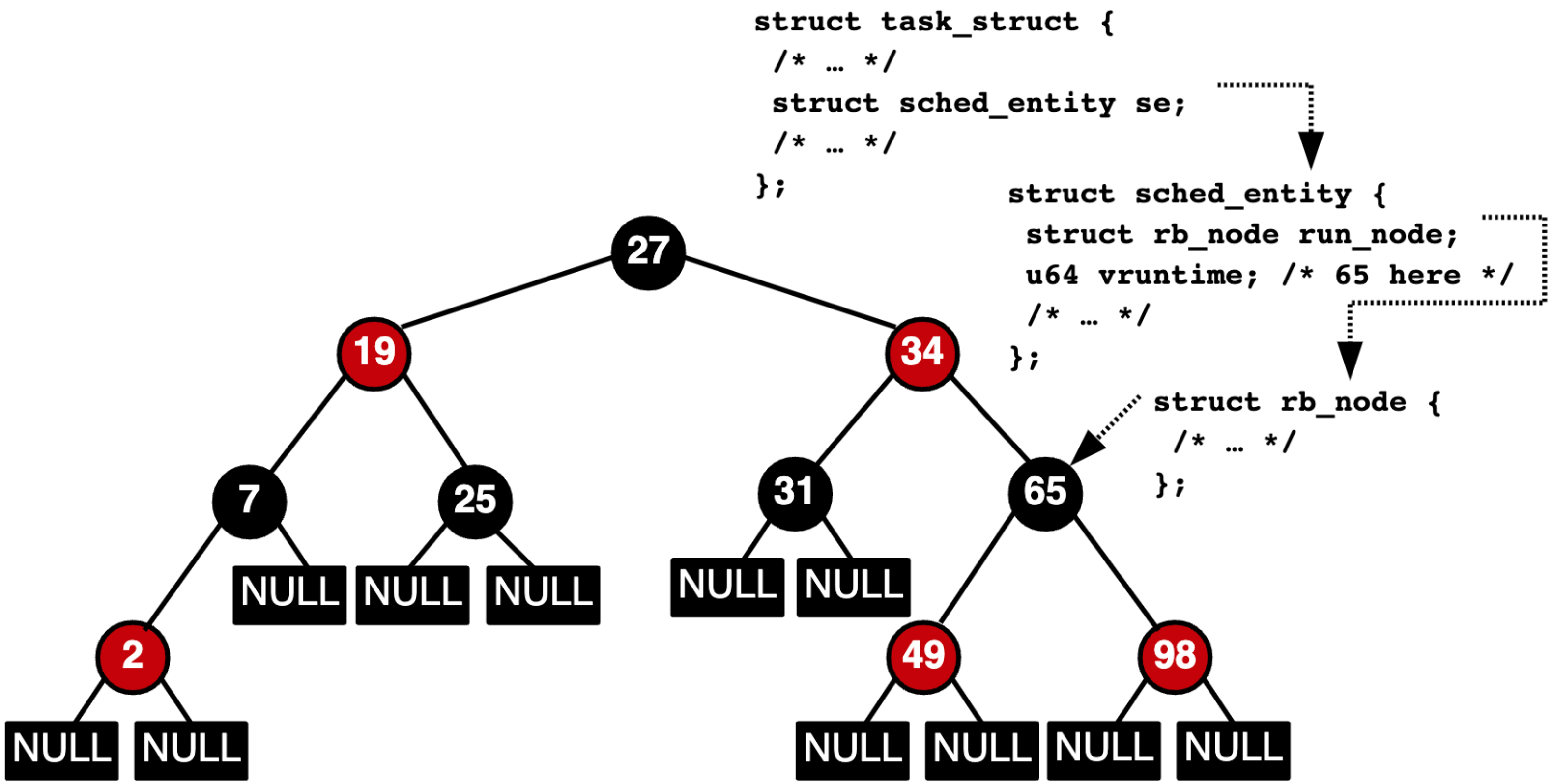
Time accounting in CFS

```
/* continue from the previous slide ... */

schedstat_set(curr->statistics.exec_max,
              max(delta_exec, curr->statistics.exec_max));

curr->sum_exec_runtime += delta_exec;
schedstat_add(cfs_rq->exec_clock, delta_exec);

/* update vruntime with delta_exec and nice value */
curr->vruntime += calc_delta_fair(delta_exec, curr); /* CODE */
update_min_vruntime(cfs_rq);

if (entity_is_task(curr)) {
    struct task_struct *curtask = task_of(curr);

    trace_sched_stat_runtime(curtask, delta_exec, curr->vruntime);
    cpuacct_charge(curtask, delta_exec);
    account_group_exec_runtime(curtask, delta_exec);
}

account_cfs_rq_runtime(cfs_rq, delta_exec);
```

Process selection in CFS

CFS maintains a `rbtree` of tasks indexed by `vruntime` (i.e., `runqueue`)

- Always pick a task with the smallest `vruntime`, the left-most node



Process selection in CFS

Quiz: There are four processes running in user-space. The properties of these four processes are as follows.

pid	scheduling class	status	vruntime	nice
100	SCHED_NORMAL	TASK_RUNNING	4000	1
200	SCHED_NORMAL	TASK_RUNNING	3000	-2
300	SCHED_NORMAL	TASK_INTERRUPTIBLE	2000	2
400	SCHED_NORMAL	TASK_UNINTERRUPTIBLE	1000	-1

Q: Which process will be scheduled by CPU scheduler? Why?

Process selection in CFS

When CFS needs to choose which runnable process to run next:

- The process with the smallest `vruntime` is selected
- It is the leftmost node in the tree

```
/* kernel/sched/fair.c */
struct sched_entity *__pick_first_entity(struct cfs_rq *cfs_rq)
{
  struct rb_node *left = cfs_rq->rb_leftmost;
  if (!left)
    return NULL;
  return rb_entry(left, struct sched_entity, run_node);
}
```

Add a task to a runqueue

When a task is woken up, it is added to a runqueue

```
/* linux/kernel/sched/fair.c */
void enqueue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int flags)
{
    bool renorm = !(flags & ENQUEUE_WAKEUP) || (flags & ENQUEUE_MIGRATED);
    bool curr = cfs_rq->curr == se;

    /* Update run-time statistics */
    update_curr(cfs_rq);

    update_load_avg(se, UPDATE_TG);
    enqueue_entity_load_avg(cfs_rq, se);
    update_cfs_shares(se);
    account_entity_enqueue(cfs_rq, se);
    /* ... */

    /* Add this to the rbtree */
    if (!curr)
        __enqueue_entity(cfs_rq, se);
```

Add a task to a runqueue

```
static void __enqueue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se)
{
    struct rb_node **link = &cfs_rq->tasks_timeline.rb_node;
    struct rb_node *parent = NULL;
    struct sched_entity *entry;
    int leftmost = 1;
    /* Find the right place in the rbtree: */
    while (*link) {
        parent = *link;
        entry = rb_entry(parent, struct sched_entity, run_node);
        if (entity_before(se, entry)) {
            link = &parent->rb_left;
        } else {
            link = &parent->rb_right;
            leftmost = 0;
        }
    }
    /* Maintain a cache of leftmost tree entries (it is frequently used): */
    if (leftmost)
        cfs_rq->rb_leftmost = &se->run_node;
    rb_link_node(&se->run_node, parent, link);
    rb_insert_color(&se->run_node, &cfs_rq->tasks_timeline);
```

Remove a task from a runqueue

When a task goes to sleep, it is removed from a runqueue

```
/* linux/kernel/sched/fair.c */
void dequeue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int flags)
{
    /* Update run-time statistics of the 'current'. */
    update_curr(cfs_rq);
    update_load_avg(se, UPDATE_TG);
    dequeue_entity_load_avg(cfs_rq, se);
    update_stats_dequeue(cfs_rq, se, flags);
    clear_buddies(cfs_rq, se);

    /* Remove this to the rbtree */
    if (se != cfs_rq->curr)
        __dequeue_entity(cfs_rq, se);
    se->on_rq = 0;
    account_entity_dequeue(cfs_rq, se);
```

Remove a task from a runqueue

```
static void __dequeue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se)
{
    if (cfs_rq->rb_leftmost == &se->run_node) {
        struct rb_node *next_node;

        next_node = rb_next(&se->run_node);
        cfs_rq->rb_leftmost = next_node;
    }

    rb_erase(&se->run_node, &cfs_rq->tasks_timeline);
}
```

Entry point: schedule()

```c
/* __schedule() is the main scheduler function. */
static void __sched notrace __schedule(bool preempt)
{
    struct task_struct *prev, *next;
    struct rq_flags rf;
    struct rq *rq;
    int cpu;

    cpu = smp_processor_id();
    rq = cpu_rq(cpu);
    prev = rq->curr;

    /* pick up the highest-prio task */
    next = pick_next_task(rq, prev, &rf);

    if (likely(prev != next)) {
        /* switch to the new MM and the new thread's register state */
        rq->curr = next;
        rq = context_switch(rq, prev, next, &rf);
    }
```

Entry point: schedule()

```
/* Pick up the highest-prio task: */
static inline struct task_struct *
pick_next_task(struct rq *rq, struct task_struct *prev, struct rq_flags *rf)
{
    const struct sched_class *class;
    struct task_struct *p;

    /* ... */
again:
    for_each_class(class) {
        /* In CFS, pick_next_task_fair() will be called.
         * pick_next_task_fair() eventually calls __pick_first_entity() */
        p = class->pick_next_task(rq, prev, rf);
        if (p) {
            if (unlikely(p == RETRY_TASK))
                goto again;
            return p;
        }
    }
    /* The idle class should always have a runnable task: */
    BUG();
```

Scheduler classes and policies

high

Stop

Deadline

- SCHED_DEADLINE

Real Time

- SCHED_FIFO
- SCHED_RR

low

Fair

high

- SCHED_NORMAL
- SCHED_BATCH
- SCHED_IDLE

low

Idle

Real-time scheduling policies

Linux provides two soft real-time scheduling classes

- `SCHED_DEADLINE`; `SCHED_FIFO`, `SCHED_RR`
- Best effort, no guarantee

Real-time task of any scheduling class will always run before non-realtime ones (CFS: `SCHED_NORMAL`, `SCHED_BATCH`)

- `schedule()` → `pick_next_task()` → `for_each_class()`

Real-time scheduling policies

`SCHED_FIFO`

- Tasks run until it blocks/yield
- Only a higher priority RT task can preempt it
- Round-robin for tasks of same priority

`SCHED_RR`

- Same as `SCHED_FIFO`, but with a fixed time slice

```
$ cat /proc/sys/kernel/sched_rr_timeslice_ms
100
```

Real-time scheduling policies

SCHED_DEADLINE

- Real-time policies mainlined in v3.14 enabling predictable RT scheduling
- Early deadline first (EDF) scheduling based on a period of activation and a “worst-case execution time” (WCET) for each task

CFS on multi-core machines

Per-CPU runqueues (rbtrees)

- To avoid costly accesses to shared data structures

Runqueues must be kept balanced

- e.g., dual-core with one long runqueue of high-priority processes, and a short one with low-priority processes
  - High-priority processes get less CPU time than low-priority ones

A load balancer runs periodically based on priority and CPU usage

Preemption and context switch

A **context switch** is the action of swapping the process currently running on the CPU to another one

Performed by `context_switch()`, which is called by `schedule()`

- Switch the address space through `switch_mm()`
- Switch the CPU state (registers) through `switch_to()`

```
/* kernel/sched/core.c */
schedule() -> __schedule() -> context_switch()
-> switch_mm(), switch_to()
```

Preemption and context switch

Then, when `schedule()` will be called?

- A task can voluntarily relinquish the CPU by calling `schedule()`
- A current task needs to be preempted if
    1. it runs long enough (i.e., its `vruntime` is not the smallest anymore)
    2. a task with a higher priority is woken up

Scheduling-related system calls

sched_getscheduler, sched_setscheduler

nice

sched_getparam, sched_setparam

sched_get_priority_max, sched_get_priority_min

sched_getaffinity, sched_setaffinity

sched_yield

Scheduling-related system calls

Source code on blackboard: `lec08-scheduler1-code.tar.gz`

```c
/* Get the scheduling class */
ret = sched_getscheduler(pid);
if(ret == -1)
    handle_err(ret, "sched_getscheduler");
printf("sched_getscheduler returns: %d\n", ret);

/* Get the priority (nice/RT) */
ret = sched_getparam(pid, &sp);
if(ret == -1)
    handle_err(ret, "sched_getparam");
printf("My priority is: %d\n", sp.sched_priority);

/* Set the priority (nice value) */
ret = nice(1);
```

Scheduling-related system calls

```c
/* Switch scheduling class to FIFO and the priority to 99 */
sp.sched_priority = 99;
ret = sched_setscheduler(pid, SCHED_FIFO, &sp);
if(ret == -1)
    handle_err(ret, "sched_setscheduler");

/* Set the RT priority */
sp.sched_priority = 42;
ret = sched_setparam(pid, &sp);
if(ret == -1)
    handle_err(ret, "sched_setparam");
printf("Priority changed to %d\n", sp.sched_priority);
```

Scheduling-related system calls

```c
/* Get the max priority value for SCHED_RR */
max_rr_prio = sched_get_priority_max(SCHED_RR);
if(max_rr_prio == -1)
    handle_err(max_rr_prio, "sched_get_priority_max");
printf("Max RR prio: %d\n", max_rr_prio);

/* Get the min priority value for SCHED_RR */
min_rr_prio = sched_get_priority_min(SCHED_RR);
if(min_rr_prio == -1)
    handle_err(min_rr_prio, "sched_get_priority_min");
printf("Min RR prio: %d\n", min_rr_prio);
```

Scheduling-related system calls

```
cpu_set_size = sizeof(cpu_set_t);
CPU_ZERO(&cs);  /* clear the mask */
CPU_SET(0, &cs);
CPU_SET(1, &cs);
/* Set the affinity to CPUs 0 and 1 only */
ret = sched_setaffinity(pid, cpu_set_size, &cs);
if(ret == -1)
    handle_err(ret, "sched_setaffinity");
```

Scheduling-related system calls

```c
/* Get the CPU affinity */
CPU_ZERO(&cs);
ret = sched_getaffinity(pid, cpu_set_size, &cs);
if(ret == -1)
    handle_err(ret, "sched_getaffinity");
assert(CPU_ISSET(0, &cs));
assert(CPU_ISSET(1, &cs));
printf("Affinity tests OK\n");
```

```c
/* Yield the CPU */
ret = sched_yield();
if(ret == -1)
    handle_err(ret, "sched_yield");
```

Summary: task = process | thread

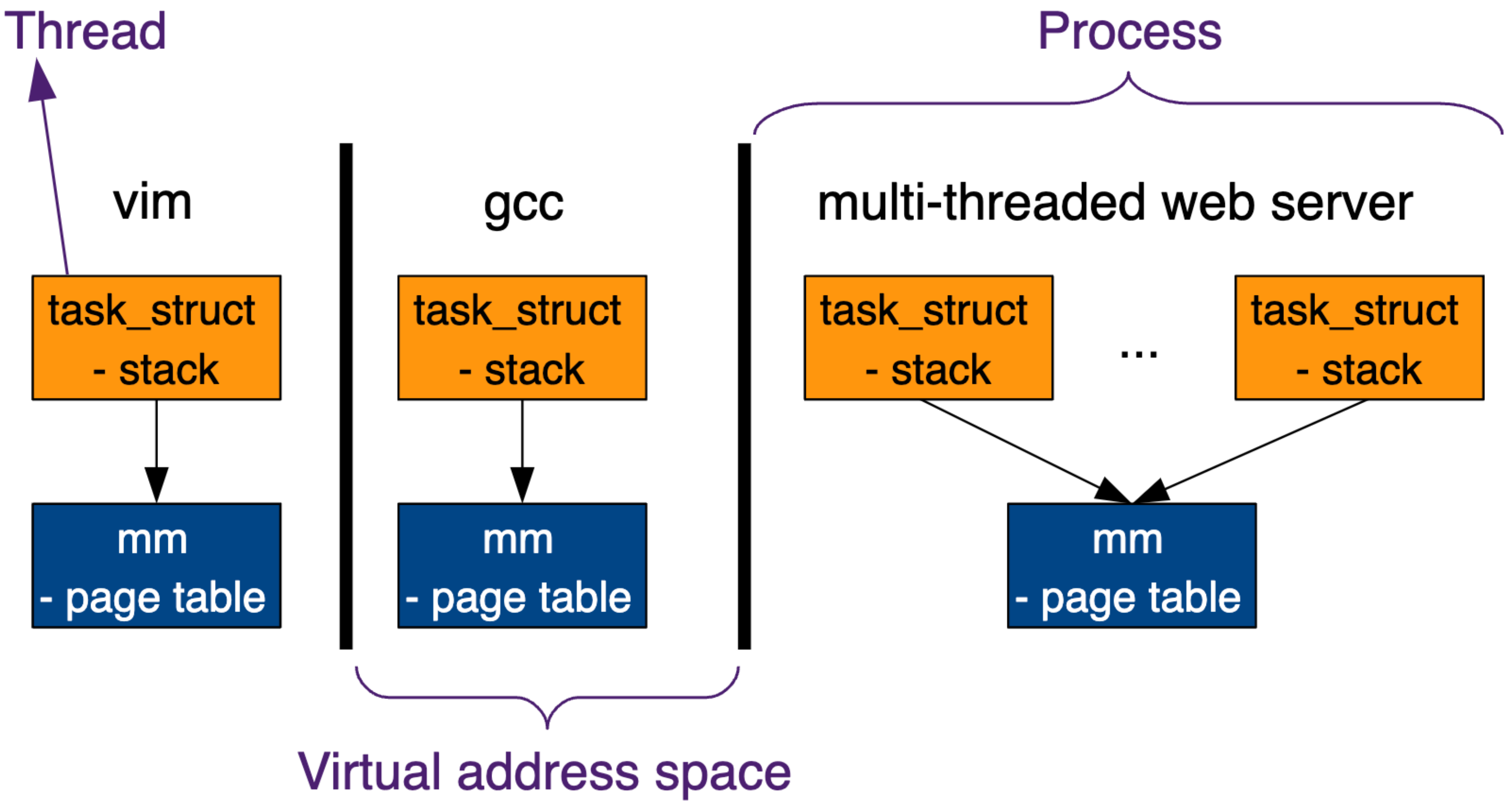
`struct task_struct`

- a process or a thread

`struct mm`

- a virtual address space

task = process | thread

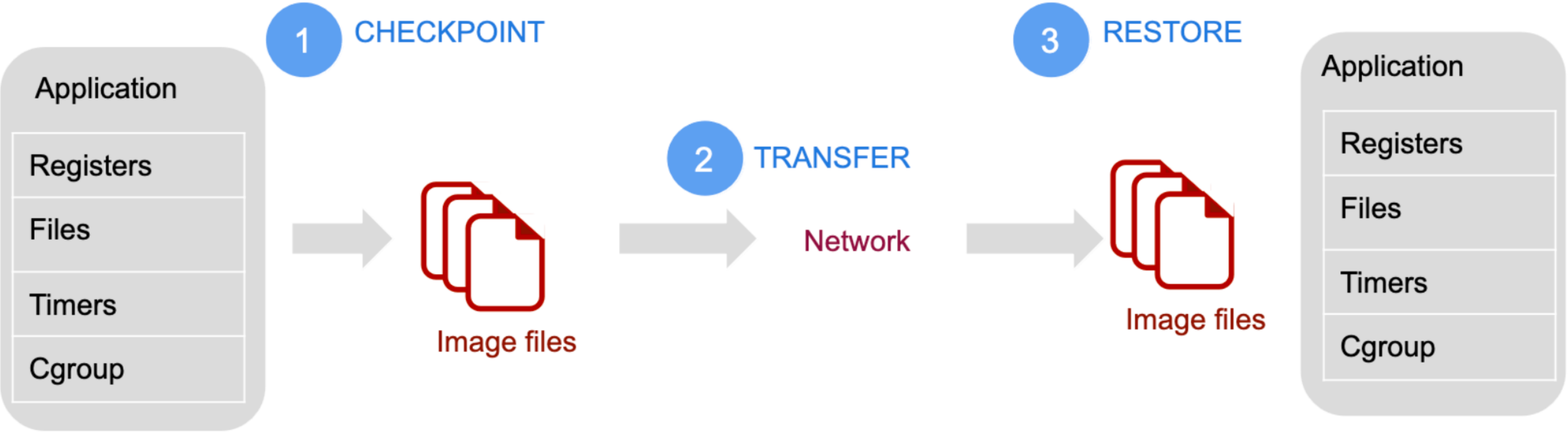


Other topics about processes?

How can we capture a process?

- Checkpoint/Restore In Userspace, or CRIU (https://criu.org/Main_Page)
- freeze a process (or a running container) and checkpoint its state to disk
- the data saved can be used to restore the process and run it exactly as it was at the time of the freeze

Linux CRIU



Application
Registers
Files
Timers
Cgroup
1
CHECKPOINT
Image files
2
TRANSFER
Network
3
RESTORE
Image files
Application
Registers
Files
Timers
Cgroup

Linux CRIU use cases

Container live migration

Slow-boot services speed up

Seamless kernel upgrade

Move "forgotten" applications into "screen"

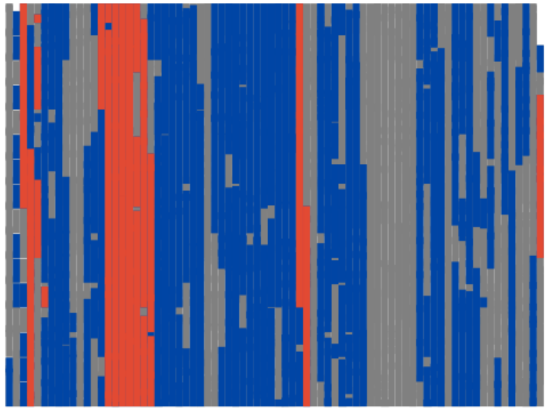
… …

https://criu.org/Usage_scenarios

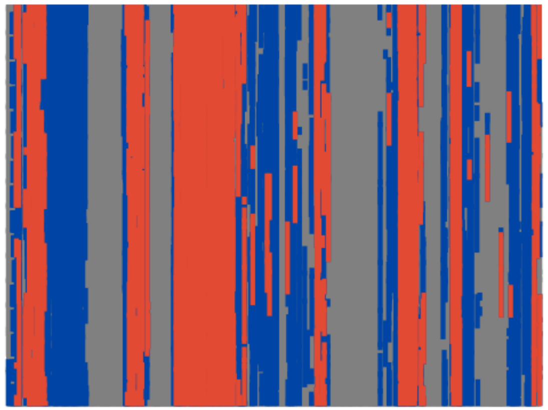
Other use case of CRIU?

Transform a process

- Can we transform a process and disable some unused code features?
  - e.g., initialization code



(a) *605.mcf_s*



(b) *Lighttpd*

Visualization of process memory footprints for executed basic blocks (blue and red), unused basic blocks (gray), and initialization-related basic blocks (red) in SPEC INT2017 605.mcf_s benchmark and Lighttpd web server.

Other use case of CRIU?

Transform a process

- Transform a process to run across the architecture boundary
- https://github.com/dapper-project/demo

Any other thoughts on transforming a process state?

Next steps

Assignment 5:

- hash table, rbtree, and Xarray (Due Feb 9th)

Final project proposal: (Due Feb 9th)

Paper reading assignment:

- OS scheduling with nest: keeping tasks close together on warm cores, EuroSys'22
- Due Feb 12th

Further readings

LKD3: Chapter 4

The Linux scheduler: A decade of wasted cores, EuroSys16

The Battle of the Schedulers: FreeBSD ULE vs. Linux CFS, USENIX ATC18