Process Management

Lesson 04

Fall 2024 FI MU

Rado Vrbovsky
<rvrbovsk@redhat.com>

Red Hat

- User Point of View
- Kernel Point of View
- Syscalls
- Process Scheduling

Red Hat

Process From User's Point of View

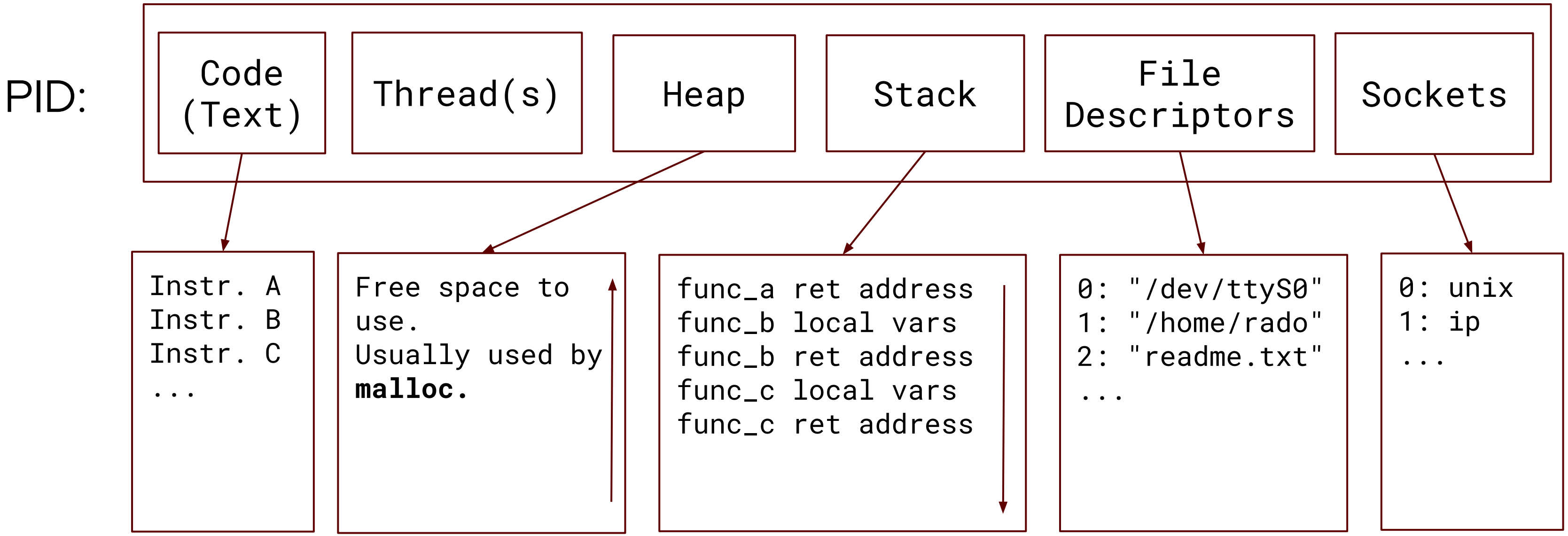
What is a process?

What is a process?

PID:

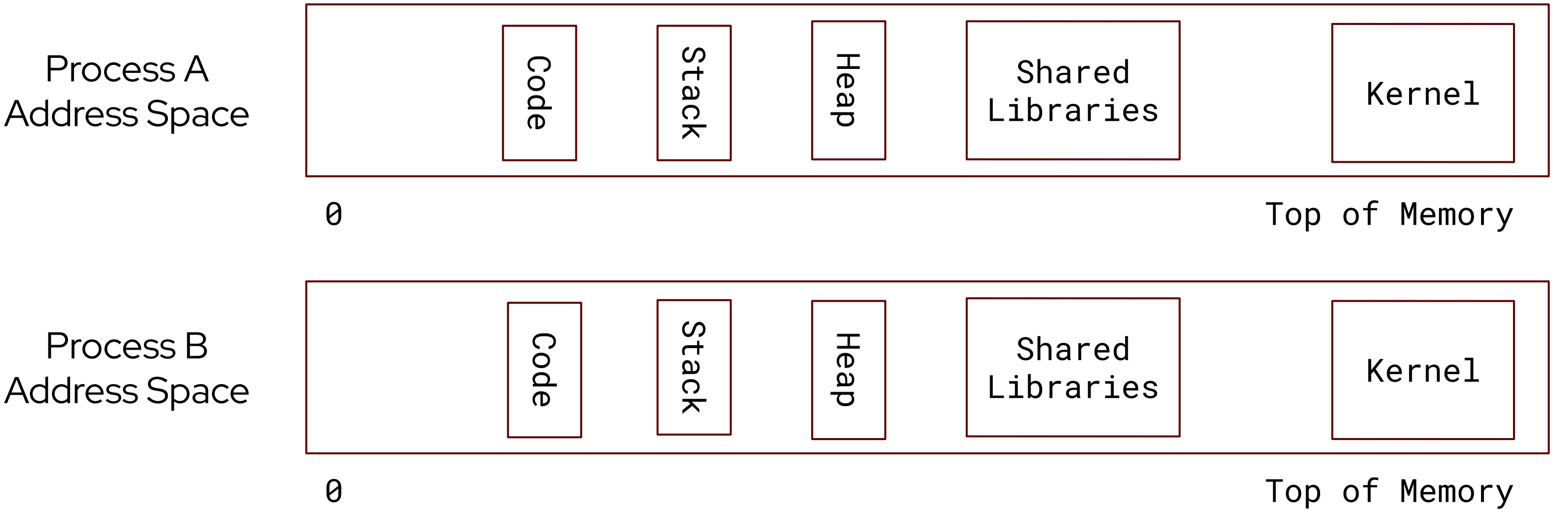
- Code (Text)
- Thread(s)
- Heap
- Stack
- File Descriptors
- Sockets

What is a process?

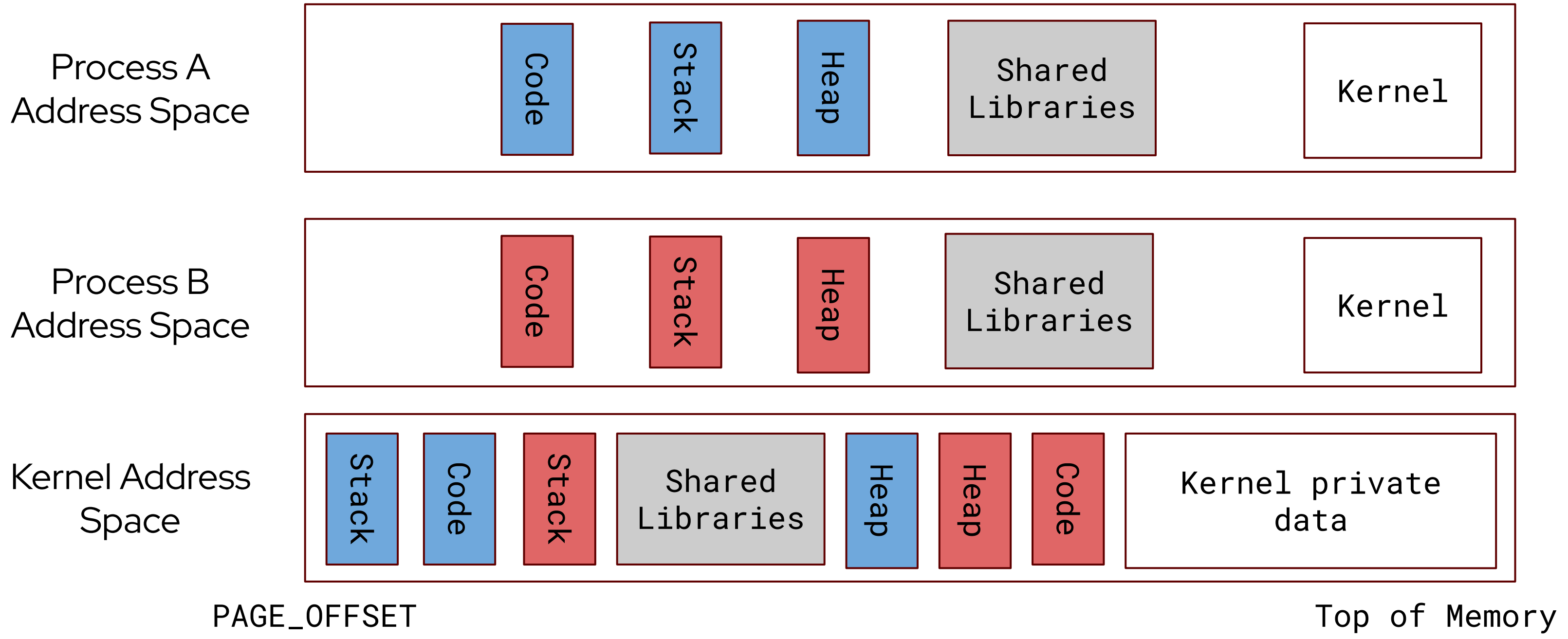


Red Hat

Process address space - Isolation



Process and kernel address space



Syscalls

```c
#include <stdio.h>

int main(void) {
	printf("Hello World!\n");
	return 1;
}
```

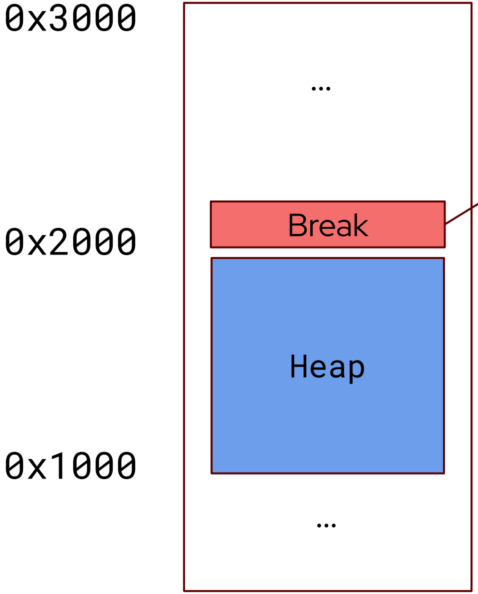
```asm
hello:
	.ascii "Hello, World!\n"
hello_len = . - hello

_start:
	mov $1, %rax		# syscall number for write
	mov $1, %rdi		# file descriptor for stdout
	mov $hello, %rsi	# pointer to the string
	mov $hello_len, %rdx	# length of the string
	syscall			# invoke syscall

	# Exit
	mov $60, %rax		# syscall number for exit
	xor %rdi, %rdi		# exit status 0
	syscall			# invoke syscall
```

Syscalls

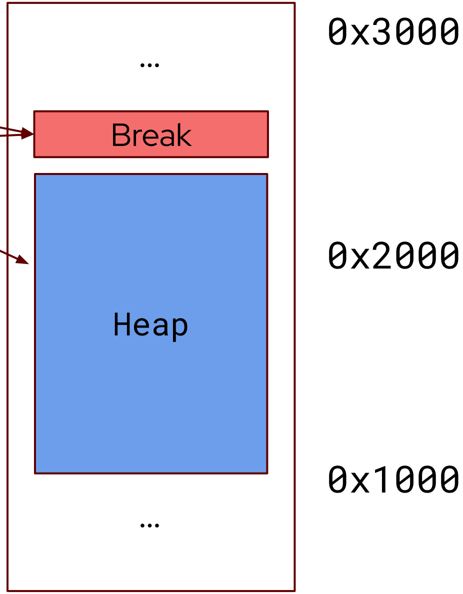
- Process Management
  - `fork`
  - `exec`
  - `clone`
  - `wait`
  - `kill`
  - `exit`
  - `nice`
  - `getpid`
  - `...`
- FS Management
  - `open`
  - `read`
  - `write`
  - `close`
  - `stat`
  - `link`
  - `unlink`
  - `...`
- Interprocess
  - `kill`
  - `signal`
  - `pipe`
  - `socket`
  - `msgget`
  - `msgrcv`
  - `semget`
  - `semop`
  - `...`
- Memory Management
  - `brk`
  - `mmap`
  - `munmap`
  - `...`

- Similar to a library call
- Called by number, not by name of the function
- Operations requiring privileged access rights are executed in a safe environment
- Over 500 syscalls

Process Break




```
brk(0x2800);
sbrk(0x800);
```



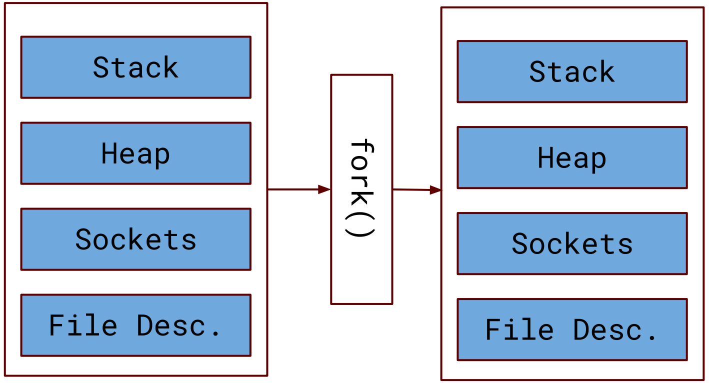
- Break - pointer to end of process data segment (heap)
- `int brk(void *addr);`
- `void *sbrk(intptr_t increment);`
- Don't use **brk** directly, use malloc

Process Creation

Step 1. Cloning

Process A

Process B

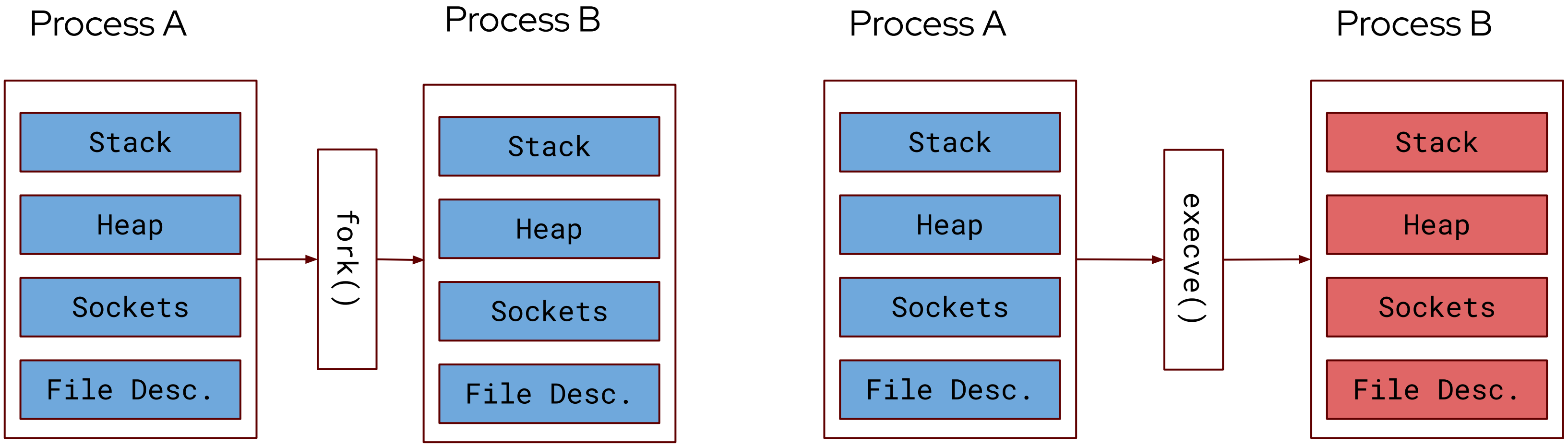


- Process A is the parent process
- Process B is the child process
- Both processes are exactly the same (stack, heap, code, file descriptors ...), except the PID, lock states and pending signals

The BSD Daemon as drawn by John Lasseter, source wikipedia.

Process Creation

Step 1. Cloning / Step 2. Execve



- Process A is the parent process
- Process B is the child process
- Both processes are exactly the same (stack, heap, code, file descriptors …), except the PID

- A new binary image is loaded from disk and completely overwrites address space of the original process

Process Creation

```
#include <stdio.h>
#include <unistd.h>

int main(void) {
      int   pid;

      pid = fork();
      if (pid == 0)
                 printf("I am a child!\n");
      else
                 printf("I am the parent!\n");

      return 0;
}
```

- Parent and child are separate processes
- They both continue executing code on the same instruction (in this example the **if** statement)

What If ...

Process Creation

```
#include <stdio.h>
#include <unistd.h>

int main(void) {
      int   pid;

      pid = fork();
      if (pid == 0)
                  printf("I am a child!\n");
      else
                  printf("I am the parent!\n");

      return 0;
}
```

- Parent and child are separate processes
- They both continue executing code on the same instruction (in this example the **if** statement)

What If …

- You could define what is the child’s entry point (what function should be executed after fork)
- Parent and child could share pieces of execution context (file descriptors, heap, stack, …)

Process Creation

```
#include <stdio.h>
#include <unistd.h>

int main(void) {
      int  pid;

      pid = fork();
      if (pid == 0)
                  printf("I am a child!\n");
      else
                  printf("I am the parent!\n");

      return 0;
}
```

- Parent and child are separate processes
- They both continue executing code on the same instruction (in this example the **if** statement)

What If ...

- You could define what is the child's entry point (what function should be executed after fork)
- Parent and child could share pieces of execution context (file descriptors, heap, stack, ...)

`clone()`

- Leveraged by the **pthread** library to create new **threads** inside processes

Process Synchronization - Signals

```
$kill -signal SIGHUP <proc_a_pid>
```

```
proc_a:

int sig_flag = 0;

void my_handler(int s) {
      sig_flag |= s;
}
int main(void) {
...
      __signalhandler ret;
      ret = signal(SIGHUP, my_handler);
...
      if (signal_flag) {
```

```
sys_kill(pid_t pid, int sig)

#define SIGHUP        1
#define SIGKILL       9
#define SIGSEGV       11
#define SIGTERM       15
#define SIGSTOP       19
...

task->sighand->
      ->action[sig - 1].sa.sa_handler
```

Process Synchronization - Signals

```
$kill -signal SIGHUP <proc_a_pid>
```

```
proc_a:

int sig_flag = 0;

void my_handler(int s) {
      sig_flag |= s;
}
int main(void) {
...
      __signalhandler ret;
      ret = signal(SIGHUP, my_handler);
...
      if (signal_flag) {
```

```
sys_kill(pid_t pid, int sig)

#define SIGHUP        1
#define SIGKILL       9
#define SIGSEGV       11
#define SIGTERM       15
#define SIGSTOP       19
...

task->sighand->
     ->action[sig - 1].sa.sa_handler
```

- There are 64 signals defined in Linux
- Signals are outdated,
  use `sigaction` instead. Or ...

Process Synchronization - Overview

Message Queues

- msgget()
- msgsnd()
- msgrcv()
- msgctl()

Shared Memory

- shmget()
- shmat()
- shmdt()
- shmctl()

Semaphores

- semget()
- semop()
- semctl()

sysvipc - System V Interprocess Communication

Sockets

- socket()
- bind()
- listen()
- accept()
- connect()
- send*()
- recv*()
- shutdown()
- close()

FIFOs

- mkfifo()
- mknod()

Signals

- kill()
- sigaction()
- signal()
- sigprocmask()
- sigpending()

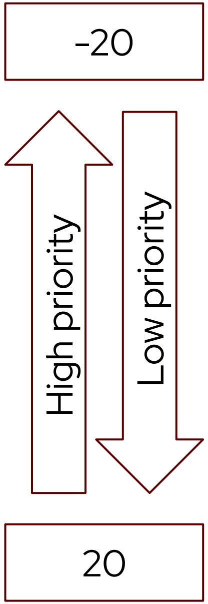
Pipes

- pipe()
- pipe2()

Red Hat

Process Niceness

```
$ nice
0
$ strace nice
…
getpriority(PRIO_PROCESS, 0)        = 20
…
$ ps ax -o pid,ni,cmd
      PID  NI CMD
      1    0 /usr/lib/systemd/systemd --switched-root --system --deserialize=39 rhgb
      2    0 [kthreadd]
      3    0 [pool_workqueue_release]
      4  -20 [kworker/R-rcu_gp]
      5  -20 [kworker/R-sync_wq]
      6  -20 [kworker/R-slub_flushwq]
...
```



- Default value is `0`
- Use `renice` to change
- Except nice values, there is also priority value for each process

Process States and Transitions

(R) Running – Process is being executed by the CPU

(S) Interruptible sleep – Process is waiting for an event, resource to be available or completion of a syscall. Process reacts to signals and can be killed

(D) Uninterruptible sleep – Process is sleeping in an uninterruptible wait, usually waiting for a block device IO. Does not react to signals and cannot be killed

(Z) Zombie – Process has finished its execution of code, but its parent process has not collected its exit code using the wait() syscall

(T) Traced/Stopped – Process is being traced or stopped.

Process From Kernel's Point of View

What is a process?

PID:	Code (Text)	Thread(s)	Heap	Stack	File Descriptors	Sockets

==

`struct task_struct` → Task descriptor

Task Descriptor/Task Structure

- One structure per user space or kernel **thread**
  - Every process has at least one thread
- Large C language structure
  - Contains all information about thread
  - Scheduling information, memory mapping, signals, files, sockets, locks, paging tables, ...
- Macro **current**
  - Architecture specific implementation
  - Points to the task_struct that is being currently executed (e.g. called a syscall)
  - Does not have to be a user space process

struct task_struct

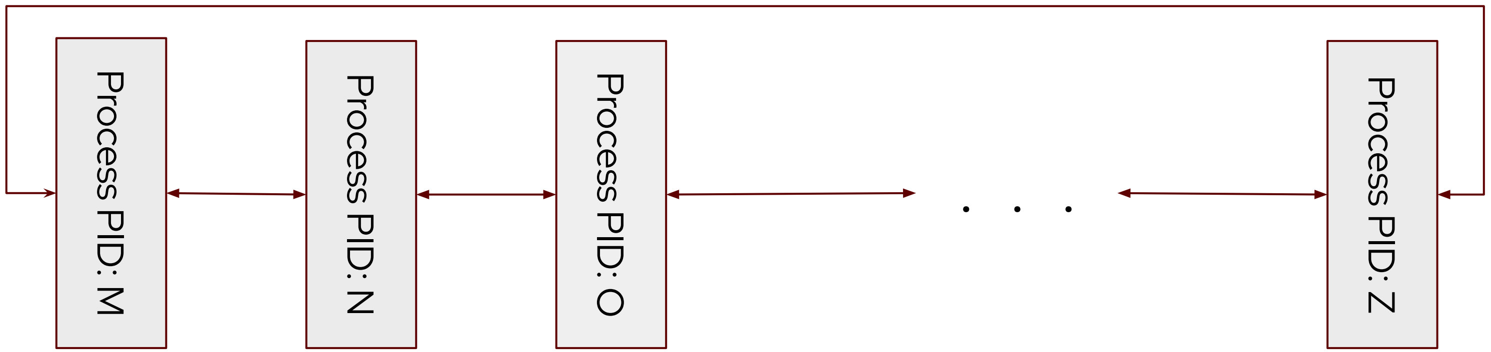
```
struct task_struct {
    ...
    pid_t                   pid;        /* Thread ID */
    pid_t                   tgid;       /* Process ID */
    ...
```

- `task_struct.pid` is the **thread** ID!
- `task_struct.gid` is the **process** ID!
  - IF (pid == tgid) → main thread
- Do not access pid and tgid directly, use
  - task_pid_nr(current)
  - task_tgid_nr(current)

Red Hat

struct task_struct - family

```
struct task_struct {
    ...
    struct task_struct __rcu   *parent;        /* Parent process */
    struct list_head           children;       /* List of children */
    struct list_head           sibling;        /* List of sibling */
    ...
    struct list_head     tasks;         /* Double linked list of all tasks */
    ...
```



struct task_struct - family

```
struct task_struct {
    ...
    struct task_struct __rcu   *parent;        /* Parent process */
    struct list_head           children;       /* List of children */
    struct list_head           sibling;        /* List of sibling */
    ...
    struct list_head      tasks;        /* Double linked list of all tasks */
    ...
```

```
#define for_each_process(p)

#define for_each_thread(p, t)

#define for_each_process_thread(p, t)
```

Red Hat

struct task_struct - state

```
struct task_struct {
     ...
     unsigned int        __state;
     ...
```

```
#define TASK_RUNNING                 0x00000000
#define TASK_INTERRUPTIBLE           0x00000001
#define TASK_UNINTERRUPTIBLE         0x00000002
...
#define EXIT_DEAD                    0x00000010
#define EXIT_ZOMBIE                  0x00000020
#define EXIT_TRACE                   (EXIT_ZOMBIE | EXIT_DEAD)
...
#define task_is_running(task)        (READ_ONCE((task)->__state) == TASK_RUNNING)
```

`struct task_struct` - stacks

```
struct task_struct {
	...
		void			*stack;	/* kernel mode stack */

	...
```

- Userspace threads have separate stacks for userspace and kernel mode
- Kernel threads have no userspace stack
- Userspace stacks are accessible through VMA structures
- Shadow stack – Copy of user space stack
  - Created at entering syscall
  - When returning back to user space, return address to user space is compared with original stack

struct task_struct - affinity

```
struct task_struct {
	...
	cpumask_t			cpus_mask;	/* CPU affinity mask */

	...
```

- Bitmask of individual CPUs where the thread is allowed to run
- Individual threads can be bound, or denied to run on specific CPUs
- Can be modified using syscalls sched_getaffinity, sched_setaffinity, or user space tool taskset

```
$ taskset -p 1
pid 1's current affinity mask: ff
```

Red Hat

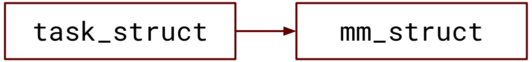
`struct task_struct` - scheduler

```
struct task_struct {
	struct thread_info		thread_info;
	...
	const struct sched_class	*sched_class;
	...
	struct thread_struct		thread;
}
```

- `task_struct.thread_info`
  - Per thread structure, contains a `flag` field, telling scheduler if thread should be preempted
  - Defined always as first item
- `task_struct.thread`
  - Architecture specific, on x86 contains CPU state when thread is preempted
  - Defined always last

Red Hat

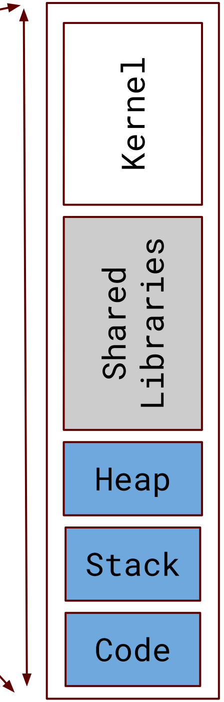
Memory Space Descriptor mm_struct



- Userspace mapping, NULL for kernel threads

```
struct task_struct {
    struct mm_struct    *mm {
        ...
        unsigned long start_code, end_code, start_data, end_data;
        unsigned long start_brk, brk, start_stack;
        unsigned long arg_start, arg_end, env_start, env_end;
        ...
        struct linux_binfmt *binfmt;
        ...
```

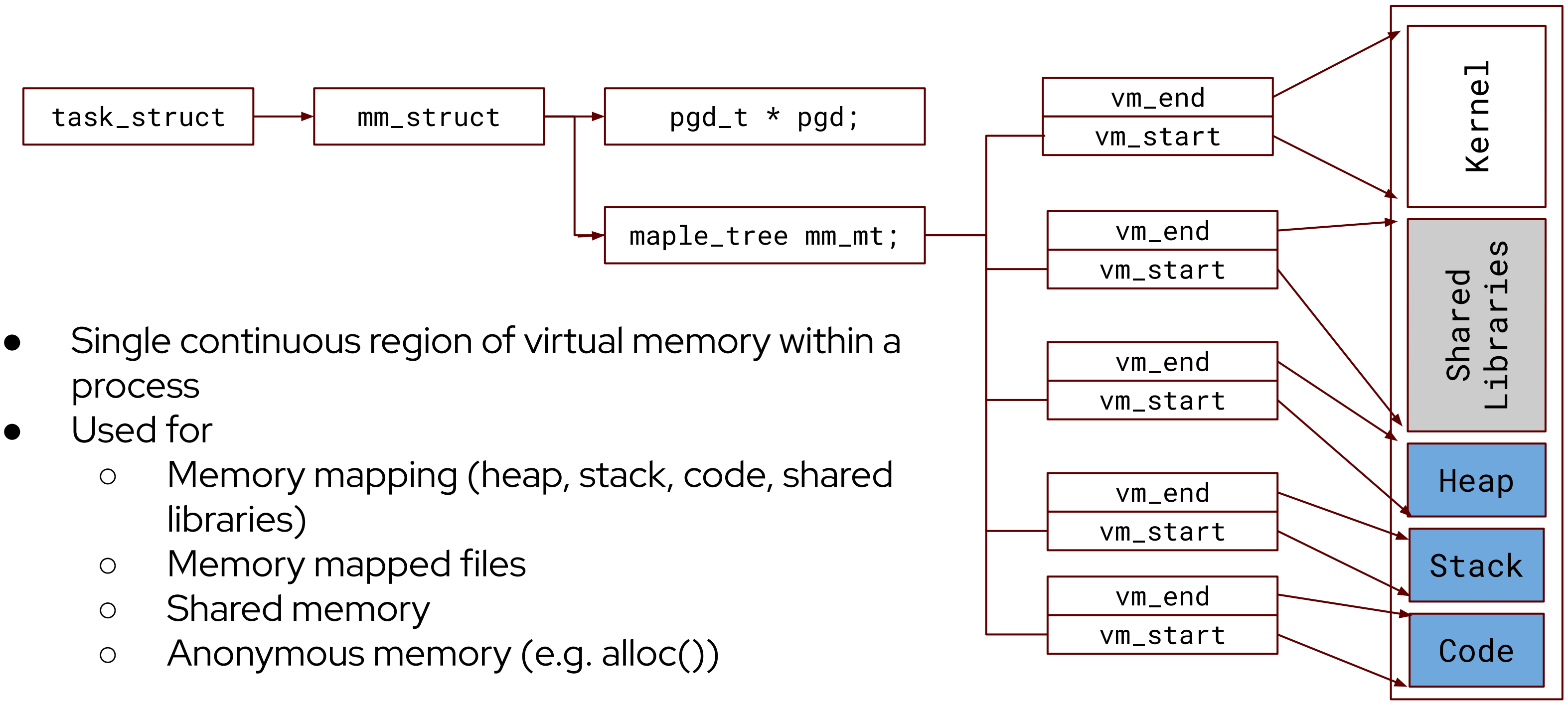
Memory Space Descriptor mm_struct



- Top level page directory for each process
  - Multilevel page table hierarchy to translate linear address to physical address

```
pgd_t *pgd = current->mm->pgd; // Get the PGD for the
current process

p4d_t *p4d = pgd_offset(pgd, address); // Get the P4D entry
pud_t *pud = p4d_offset(p4d, address); // Get the PUD entry
pmd_t *pmd = pud_offset(pud, address); // Get the PMD entry
pte_t *pte = pmd_offset(pmd, address); // Get the PTE entry
```

Virtual Memory Space Descriptor vm_area_struct



- Single continuous region of virtual memory within a process
- Used for
  - Memory mapping (heap, stack, code, shared libraries)
  - Memory mapped files
  - Shared memory
  - Anonymous memory (e.g. alloc())

Syscalls

Syscalls - Uname

```
$ uname -a
Linux fedora33-kw 6.8.11-200.fc39.x86_64 #1 SMP PREEMPT_DYNAMIC Sun May 26
20:05:41 UTC 2024 x86_64 GNU/Linux
```

```
DECLARE_RWSEM(uts_sem);                                  // Uname and hostname semaphore

SYSCALL_DEFINE1(newuname, struct new_utsname __user *, name)     // Syscall macro
{
     struct new_utsname tmp;                             // System information structure

     down_read(&uts_sem);                                // Take the semaphore
     memcpy(&tmp, utsname(), sizeof(tmp));               // Copy data
     up_read(&uts_sem);                                  // Release the semaphore
     if (copy_to_user(name, &tmp, sizeof(tmp)))          // Copy buffer to user space
               return -EFAULT;
     return 0;                                           // Return OK
}
```

Syscalls - Macros

```
#define SYSCALL_DEFINE1(name, ...) SYSCALL_DEFINEx(1, _##name, __VA_ARGS__)
…
#define SYSCALL_DEFINE6(name, ...) SYSCALL_DEFINEx(6, _##name, __VA_ARGS__)


#define SYSCALL_DEFINEx(x, sname, ...)                              \
     SYSCALL_METADATA(sname, x, __VA_ARGS__)                        \
     __SYSCALL_DEFINEx(x, sname, __VA_ARGS__)
```

- `SYSCALL_METADATA` - Data for tracing events
- `__SYSCALL_DEFINEx` - Complex machinery of macros and GCC extensions to create the syscall implementation

Red Hat

Syscalls - Entries

```
0     common  read                      sys_read
1     common  write                     sys_write
2     common  open                      sys_open
```

```
$ sh ./scripts/syscalltbl.sh --abis common,64 arch/x86/entry/syscalls/s
     yscall_64.tbl arch/x86/include/generated/asm/syscalls_64.h
```

```
__SYSCALL(0, sys_read)
__SYSCALL(1, sys_write)
__SYSCALL(2, sys_open)
```

```
#define __SYSCALL(nr, sym) case nr: return __x64_##sym(regs);
```

Red Hat

Syscalls - Table

```
long x64_sys_call(const struct pt_regs *regs, unsigned int nr)
{
     switch (nr) {
     #include <asm/syscalls_64.h>
     default: return __x64_sys_ni_syscall(regs);
     }
};
```

```
__SYSCALL(0, sys_read)
__SYSCALL(1, sys_write)
__SYSCALL(2, sys_open)
```

Copying data to and from user space

Copy simple values:

- `get_user(x, ptr); // Get a simple variable from user space.`
- `put_user(x, ptr); // Write a simple value into user space.`
    - `x - Variable to store result`
    - `ptr - Source/Destination address, in user space.`

Copy data:

- `copy_from_user(void *to, const void __user *from, unsigned long n);`
- `copy_to_user(void __user *to, const void *from, unsigned long n);`

Process Scheduler

Red Hat

Scheduler

- Divide CPU resources between competing consumers (user/kernel threads)
- Smallest scheduled unit is a thread (every process has at least one thread)
- Thread state machine is defined using flags
- Threads being executed or are ready to be executed are stored in a structure named **runqueue**
- Sleeping threads are stored in **waitqueue**
- Each CPU has its own runqueues
- Waitqueue is created by device drivers and the kernel, there can be many wait queues

Context Switch / Process Swap

Threads leave the CPU in one of two ways:

- Voluntary
  - Thread is waiting for an IO operation to finish
  - Thread is waiting for a lock to be opened
  - Thread decides to sleep
- Involuntary
  - Scheduling: When the CPU scheduler decides to switch to a different thread based on scheduling policies (e.g. processes exceeded its scheduled allocation of CPU time)
  - Preemption: When a higher-priority thread becomes ready to run and preempts the currently executing thread.

Context Switch / Process Swap

- Architecture specific
- Expensive operation
  - Saving CPU state of current thread (previous)
  - Installing MM settings of the new (next) thread
  - Restoring CPU state of the new (next) thread
    - `context_switch(...)`

Scheduler Policies

- Linux scheduler consists of several scheduling policies
- Scheduling policy == scheduling algorithm
- Every thread in the system is associated with only one policy
- Current scheduling policies
  - SCHED_DEADLINE
  - SCHED_FIFO, SCHED_RR
  - SCHED_NORMAL, SCHED_BATCH
  - SCHED_IDLE

Scheduling Classes

- Abstraction classes that hold the individual scheduling policies
- New classes can be added and removed to source code depending on need
- Each scheduling class has a different model how to select eligible tasks/threads, each scheduling class maintains its own runqueue

```
struct sched_class {
…
      void (*enqueue_task) (struct rq *rq, struct task_struct *p, int flags);
      void (*dequeue_task) (struct rq *rq, struct task_struct *p, int flags);
…
      struct task_struct *(*pick_next_task)(struct rq *rq);
…
      void (*task_tick)(struct rq *rq, struct task_struct *p, int queued);
…
}
```

Stop Scheduler Class

- Does not have a policy
- Highest priority
- Can preempt everything and is preempted by nothing
- Available only on SPM
- One kernel thread per CPU
  - "migration/N"
- Used by task migration, CPU Hotplug, RCUs, ftrace, kernel live patching

(Early) Deadline Scheduler Class

- Policy SCHED_DEADLINE
- The task with the earliest deadline will be served first
- User has to set 3 parameters
  - Period – activation pattern of the real time task
  - Runtime – amount of CPU time that the application needs
  - Deadline – maximum time in which the result must be delivered
- Used for periodic real time tasks e.g. multimedia, industrial control

Real Time Scheduler Class

- Used for short latency sensitive tasks
- Two policies
- SCHED_FIFO
  - AKA POSIX scheduler
  - Runqueue is a FIFO pipe
  - Thread will run until it voluntary yields the CPU
  - Real time aggressive
- SCHED_RR
  - 100ms time slice by default
  - Round Robin scheduler
  - Realtime moderately aggressive

CFS - Completely Fair Scheduler

- Most common used scheduler, used for the rest of the all tasks in the system
- Introduced by Ingo Molnar in 2007, for long time the only scheduler
- Scheduling policies
    - SCHED_NORMAL - Normal Unix tasks, default scheduler
    - SCHED_BATCH - Low priority, non interactive jobs
- Implemented with red-black trees
- Tracks virtual runtime of tasks (amount of time a task has run) in nanoseconds
- Tasks with shortest vruntime runs first, left most node in the RB tree
- Priority is used to set tasks weight, slower will vruntime increase
- Kernel will reset all the vruntime values in RB tree when starting a new scheduling epoch

Idle Scheduler

- Lowest priority scheduling class
- No scheduling policies
- One kernel thread (idle) per CPU
  - “swapper/N”
- Idle thread runs only when nothing else is runnable on a CPU
- Puts the CPU in a deep sleep state and is woken when there is a thread to run
- There is always only one task in idle class

The Extensible Scheduler

- Scheduling policy SCHED_EXT
- Introduced recently (Jan 2023, The future is now!)
- Idea of “plugable schedulers”
- Not really a scheduler itself, but a framework
- Uses eBPF technology
  - Runtime load schedulers from userspace
  - Without need to recompile the kernel
  - Allows safe experimentation
  - Library of schedulers for niche applications (e.g. service, specific game, …)

Scheduler Code

- `schedule()` → `__schedule()` → `__pick_next_task()`
- Classes are ordered by the task priority they cover, classes with higher priority are being queried first
- __pick_next_class returns a pointer to the task_struct it self which will be executed

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

      BUG(); /* The idle class should always have a runnable task. */
}
```

Thread Scheduling

- Thread state machine is defined using flags
  - task_struct.thread_info.flas |= TIF_NEED_RESCHED
    - `set_tsk_need_resched(struct task_struct *tsk)`

- Who is calling the scheduler?
  - Executed in context of **current** process
  - Return from syscall
  - Return from interrupt

Red Hat

Thank you!

Questions?