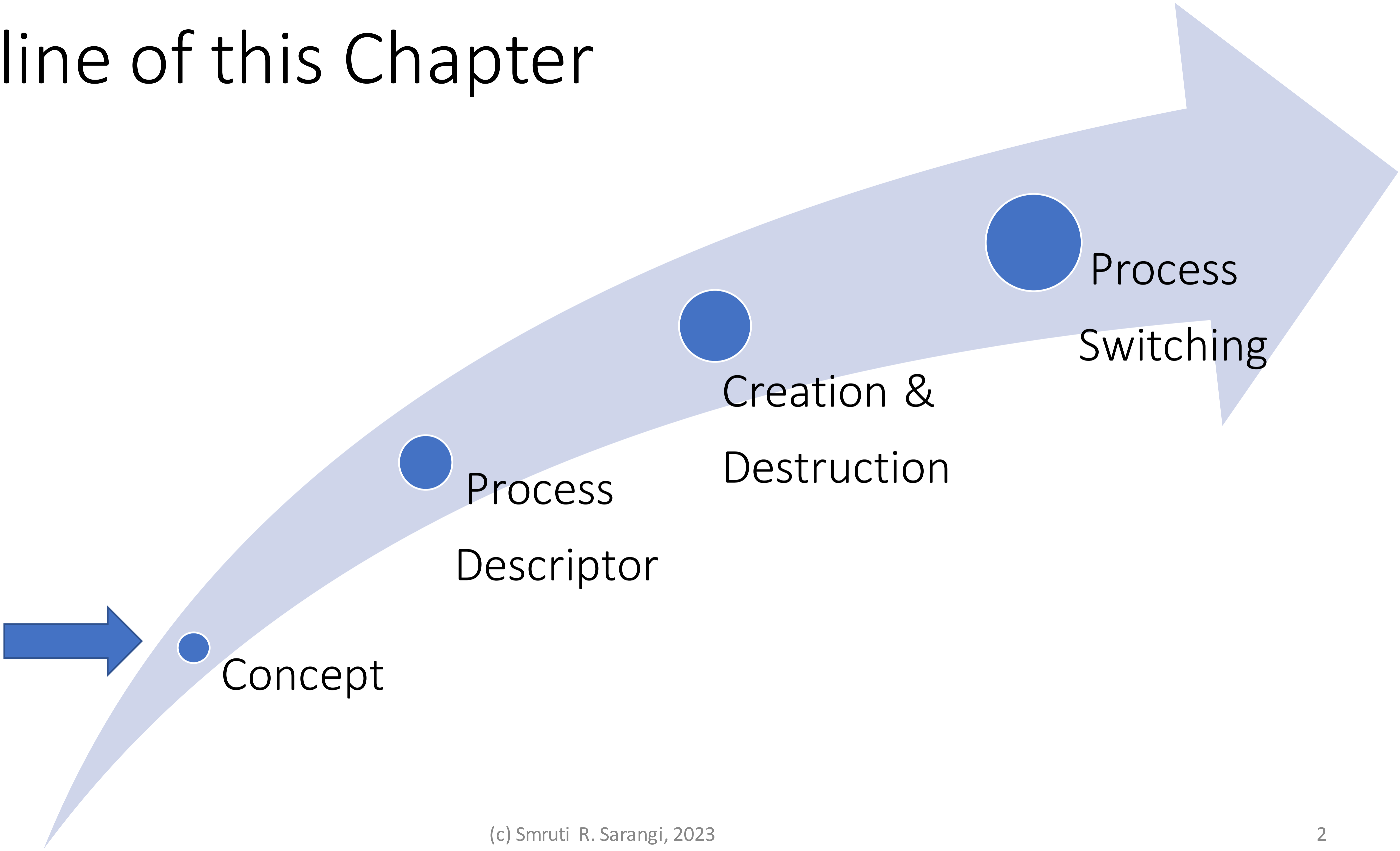
Chapter 3: Processes

Smruti R Sarangi

IIT Delhi

Outline of this Chapter



Concept

Process Descriptor

Creation & Destruction

Process Switching

Introduction to a Process

What is a process?

It is an instance of a running program. When we run a program, it acquires a life of its own and is associated with a lot of additional data structures. All of these including the state of the executing program comprise the *process*.

What does a process own?

CPU time, memory, open files, network connections,

How do processes communicate with the OS?

Processes send messages to the OS via system calls.
The OS sends messages to a process via signals (exact mechanism described later)

Types of Processes

Organized as a tree.

Stand-alone or as a lightweight process in a group

- A group of lightweight processes (threads) share part of the address space between themselves.

Single-threaded

- Contains only a single thread of execution

Multi-threaded

- Contain multiple threads of execution

The Process Descriptor

struct task_struct

include/linux/sched.h

- This is the apex data structure for storing all process-related information.
- Every process is associated with a *task_struct* data structure
- It maintains all the bookkeeping information for the process
- It is rather complex
- Reason: The main aim is to keep all process-related information in one place.

The Key Components of *task_struct*

Field	Meaning
struct thread_info thread_info	Low-level information
uint state	Process state
void * stack	Kernel stack
Priorities	prio, static_prio, normal_prio
struct sched_info sched_info	Scheduling information
struct mm_struct *mm, *active_mm	Pointer to memory information
pid_t pid	Process id
struct task_struct *parent	Parent process
struct list_head children, sibling	Child and sibling processes
File system, I/O, synchronization, and debugging fields	

The core of task_struct is thread_info (it is on its way out in future kernels)

What is thread_info?

It is a low-level data structure to store task-related information.

What is a low-level data structure?

Its layout is machine specific. Typically, its position in the address space and the way its fields are laid out are found to be very useful in accessing it to retrieve useful information. The contents of the data structure also abstract out details of the underlying hardware.

The *arch* Folder

- The Linux codebase has two parts: machine dependent and machine independent
- Most of the code is machine independent. Otherwise, it will become impossible to manage such a large codebase.
- We need a layer to abstract out details of the underlying machine.
- This is the job of the *arch* folder (machine dependent part) that:
  - The kernel uses generic data types such as u32 or u64. They are defined within files of the arch folder (for each architecture)
  - Map high-level primitives to assembly-level code snippets (arch. specific)
  - Provide other low-level services: booting the system, managing the memory system, power management, etc.

Let us come back to *thread_info*

struct thread_info

arch/x86/include/asm/thread_info.h

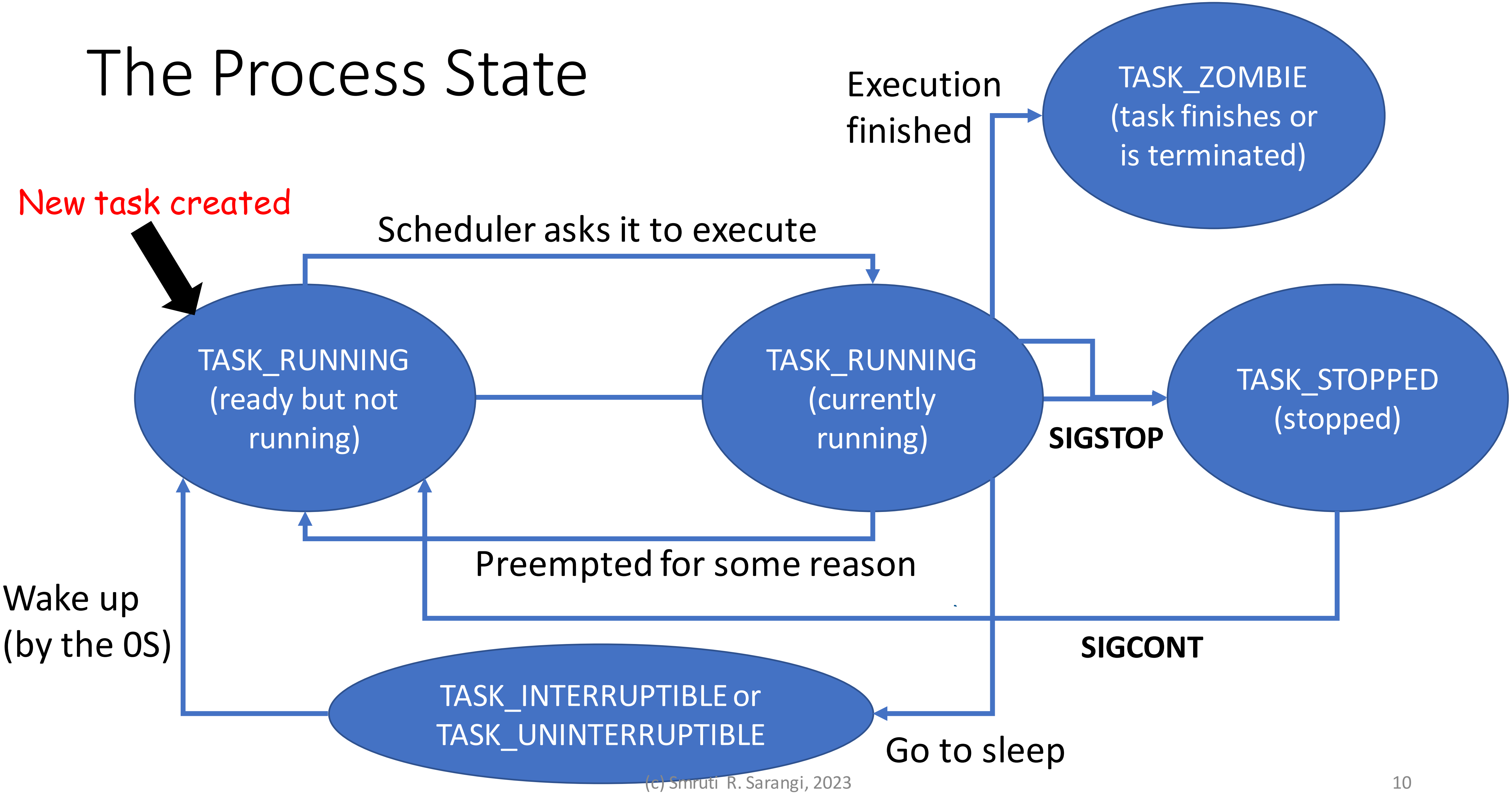
- Contains some important information about the HW state

```
struct thread_info {
        unsigned long flags;
        unsigned long syscall_work;
        u32 status;
        u32 cpu;
}
```

Flags for the state of the process, system calls, and thread synchrony, resp.

Current CPU

The Process State



New task created
Scheduler asks it to execute
TASK_RUNNING (ready but not running)
TASK_RUNNING (currently running)
Execution finished
TASK_ZOMBIE (task finishes or is terminated)
SIGSTOP
TASK_STOPPED (stopped)
Preempted for some reason
SIGCONT
Wake up (by the OS)
TASK_INTERRUPTIBLE or TASK_UNINTERRUPTIBLE
Go to sleep

(c) Smruti R. Sarangi, 2023

Explanation of the Process States

- We have two running states
  - Can run (not getting an available CPU)
  - Already running
- There are two interrupted states
  - INTERRUPTIBLE → The process can be sent a message from the OS (known as a signal), and it can be woken up
  - UNINTERRUPTIBLE → The process is waiting for a particular resource to become available. It will not wake up regardless of the signal that is sent to it.
- TASK_ZOMBIE
  - A process finishes if the OS kills it or if it calls the *exit()* system call
  - Its state is however not removed. Its parent is informed with the SIGCHLD signal.
  - The parent needs to call the system call *wait()* to read the exit value of the child and then only the child process's state is cleaned up.

More about Process States

- ZOMBIE state continued …
  - The process needs to explicitly call *exit(int exitcode)* when it finishes
  - *exitcode* indicates the status of the process's execution
  - If it is 0, then it means that the process executed successfully
  - Otherwise, it means that there was an error.
  - The exit code indicates the type of the error
  - A value of `1' indicates that there was an error (not specific)
  - Any other value indicates the exact nature of the error
  - Processes are organized in a tree-like hierarchy. Every process has a parent. The parent needs to know the status of the child's execution (exit code). Hence, we maintain the child process as a zombie until the parent reads its status using variants of the *wait* system call.

The STOPPED state

- A process can be stopped/suspended
  - Send it the SIGSTOP signal: example, kill –STOP {process id}
  - The *kill* system call or command line command *kill* sends a signal to a process
  - This suspends the process
  - Another approach: Type Ctrl-Z on the terminal
    - Sends the SIGTSTP signal
    - A process can choose to ignore this
    - If it is not ignored, the process is suspended
- The process can be resumed by sending the SIGCONT signal to it
  - Use a system call to send the signal to a process
  - Use the *fg* command line utility

The Kernel Stack

Where does a process keep its information when there is a context switch? Does it need an avatar that runs in kernel mode?

Yes!

Every process is associated with a kernel stack and often a kernel thread. When a kernel thread works on behalf of the process to do some work for it, the kernel stack is used.

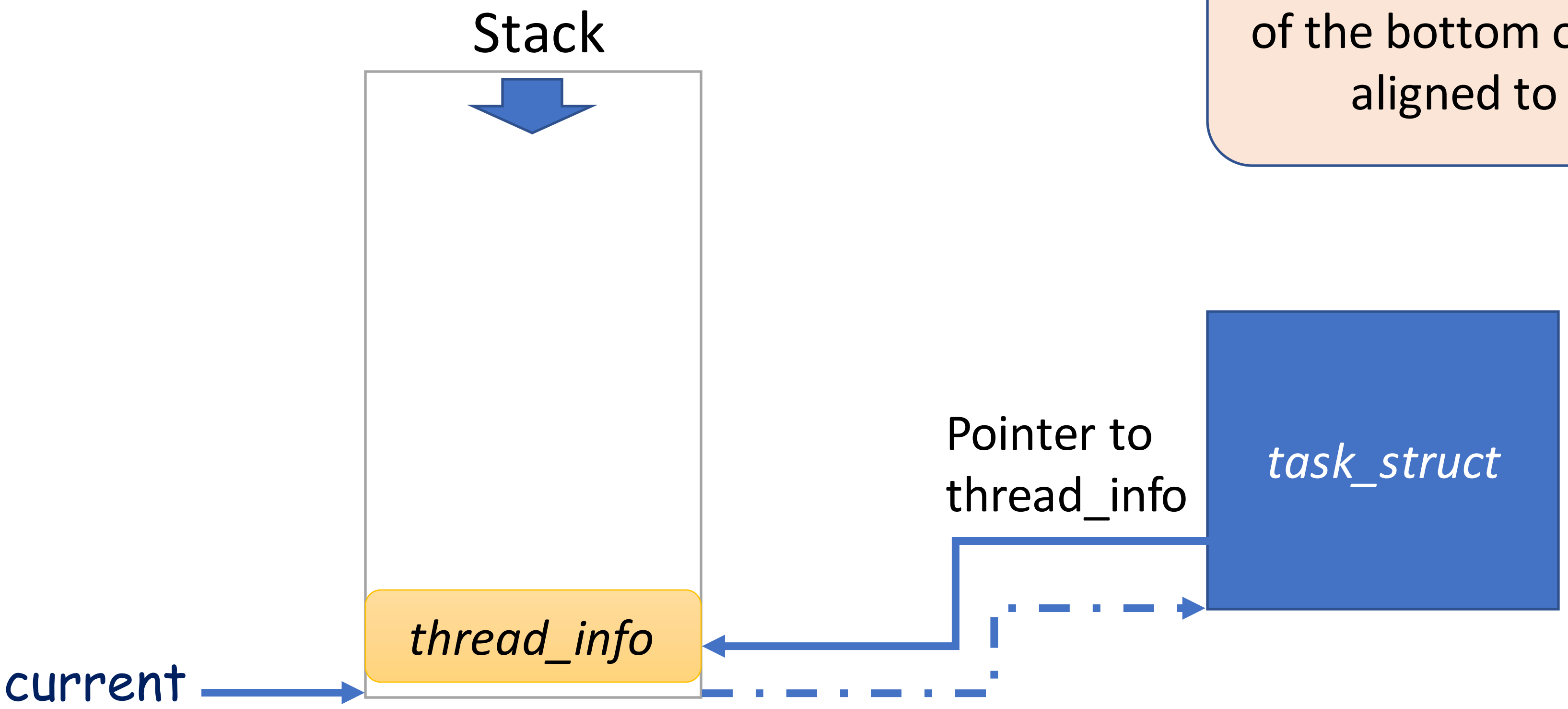
There are some limitations on the kernel stack. It cannot be arbitrarily large. In fact, no structure in the kernel can grow indefinitely and irregularly. Memory management of kernel pages is complicated.

Limitations on the Kernel Stack

- Its size is limited to 4 KB * 2 = 8 KB
- They contain useful data as long as the thread is alive or in a zombie state
- There are per-thread stacks and a few stacks that are reserved for each CPU
- The main CPU stack is an interrupt stack that is used by interrupt handlers
- What about nested interrupts?
  - Some interrupts are non-maskable interrupts (NMIs)
  - They cannot be ignored.
  - This means that if we are already running an interrupt handler, then we need to still handle these interrupts.
  - There is thus a need to switch to a new interrupt stack (for the NMI)
  - x86 processors have an interrupt stack table (IST) per CPU with 7 entries

Structure of the Stack in Old Kernels

Problem: Given an *esp*, find the address of the bottom of the stack (assume it is aligned to an 8 KB boundary)



It was at the bottom of the stack

It was easy for code to find the address of the *task_struct* via the thread_info structure

In the Current Kernel

- The *current* macro

```
DECLARE_PER_CPU(struct task_struct *, current_task);

static __always_inline struct task_struct *get_current(void)
{
    return this_cpu_read_stable(current_task);
}
#define current get_current()
```

- Store the pointer to the current task in a global variable.
- The *this_cpu_read_stable* macro reads the *task_struct* pointer from a separate per-CPU register.
- In some architectures, this points to a *thread_info* structure that in turn has a pointer to the *task_struct*

What did we learn from this part?

- The current task_struct is something that needs to be accessed very quickly and very frequently
- ❓ Where do we store it?
  - We cannot store it in a general purpose register. We have limited registers.
  - We cannot store it in a global variable. Should be CPU-specific.
  - We can store a pointer to it at the bottom of the stack. We would need additional instructions to compute the address of the pointer.
  - Store a pointer to it in a model-specific register (MSR)
  - A pointer can be stored in the local storage area on the CPU, whose address is known (used in x86)

What is actually used?

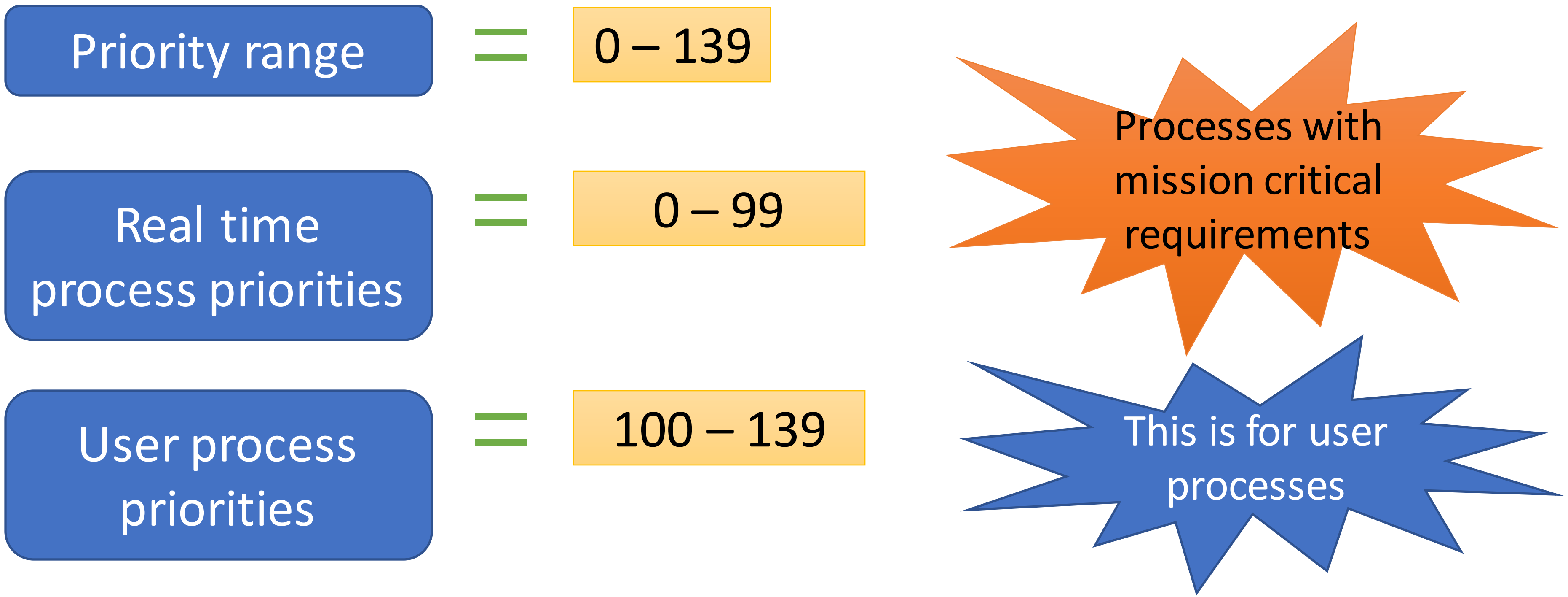
https://docs.kernel.org/core-api/this_cpu_ops.html

- We store CPU variables in segmented memory
- The gs segment register can point to a per-cpu memory region
- Store all CPU-local variables there
- The DEFINE_PER_CPU macro in the kernel does exactly this
- The cache lines are not shared between processors
  - Leads to a higher performance (lines don’t bounce between cores)

Field	Meaning
struct thread_info thread_info	Low-level information
uint state	Process state
void * stack	Kernel stack
Priorities	prio, static_prio, normal_prio
struct sched_info sched_info	Scheduling information
struct mm_struct *mm, *active_mm	Pointer to memory information
pid_t pid	Process id
struct task_struct *parent	Parent process
struct list_head children, sibling	Child and sibling processes
File system, I/O, synchronization, and debugging fields	

Process Priorities

- Different processes have different priorities.
- This information is used by the scheduler for scheduling.



Priority range	=	0 – 139
Real time process priorities	=	0 – 99
User process priorities	=	100 – 139

Processes with mission critical requirements

This is for user processes

How do we interpret the process priorities?

- The sense of normal and real-time priorities is different

For real-time processes, higher the priority value, higher is the actual priority

This means that a task with priority 99 has the highest priority in the system

For regular processes, lower the priority value, higher is the actual priority

This means that a task with priority 139 has the lowest priority in the system

The *nice* mechanism

A user process can change its user priority by invoking the <nice> or <chrt> commands

```
> nice –n <nice value> <command>
```

Priority = 120 + <nice value>

Relevant Kernel Code

kernel/sched/core.c

```
else if (rt_policy(policy))
            prio = MAX_RT_PRIO - 1 - rt_prio;
else
            prio = NICE_TO_PRIO(nice);
```

Flip the sense if it is a real-time process

prio = 120 + nice

- Lower the value of *prio,* higher the actual priority
- Many systems allow the superuser to only issue commands with (-)ve nice values
- The scheduler typically has different queues for different *prio* values
  - Higher-priority queues get more CPU time

sched_info

include/linux/sched.h

```
/* # of times we have run on this CPU: */
unsigned long pcount;

/* Time spent waiting on a runqueue: */
unsigned long long  run_delay;

/* Timestamps: */

/* When did we last run on a CPU? */
unsigned long long last_arrival;

/* When were we last queued to run? */
unsigned long long last_queued;
```

Past run history on the CPU

How long has the task waited?

When did the task last run and when did it last enter the runqueue to run?

mm_struct

Represents an address space

- This structure contains all the information related to the memory usage of the process
- It basically functions as the memory descriptor of the process

Key components

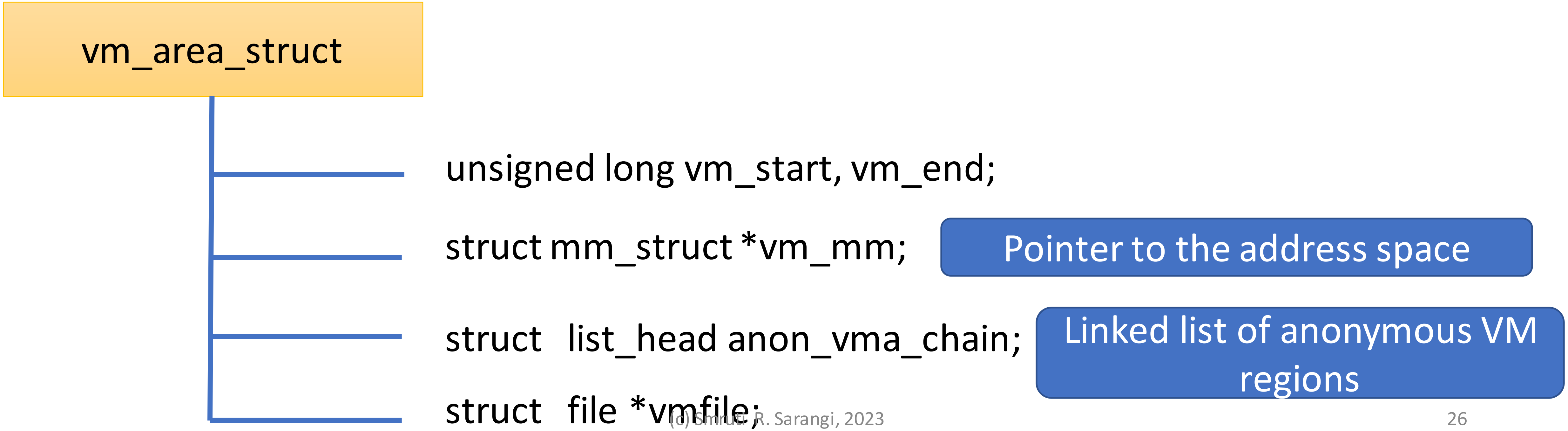
struct maple_tree mm_mt	Stores all VM regions
unsigned long task_size	Size of the VM space
pgd_t * pgd;	Pointer to the page table
int map_count	Number of VM regions
stats: total_vm, locked_vm, pinned_vm	Total pages mapped, #locked and #pinned pages

Start/end of memory regions	start_code, end_code, start_data, end_data, start_stack, ….
Owner process	struct task_struct *owner
The CPUs that the process has executed on	unsigned long cpu_bitmap[]

What does a virtual memory region look like?

include/linux/mm_types.h

- Every virtual memory region is represented by a vm_area_struct object



vm_area_struct

unsigned long vm_start, vm_end;

struct mm_struct *vm_mm;

Pointer to the address space

struct list_head anon_vma_chain;

Linked list of anonymous VM regions

struct file *vmfile;

The Maple Tree

The maple tree is the most important structure

Keeps track of VM regions

How do we locate VM regions?

https://lwn.net/Articles/845507/

Red-Black Tree vs B-Tree

? What is the best data structure for storing data about VM regions?

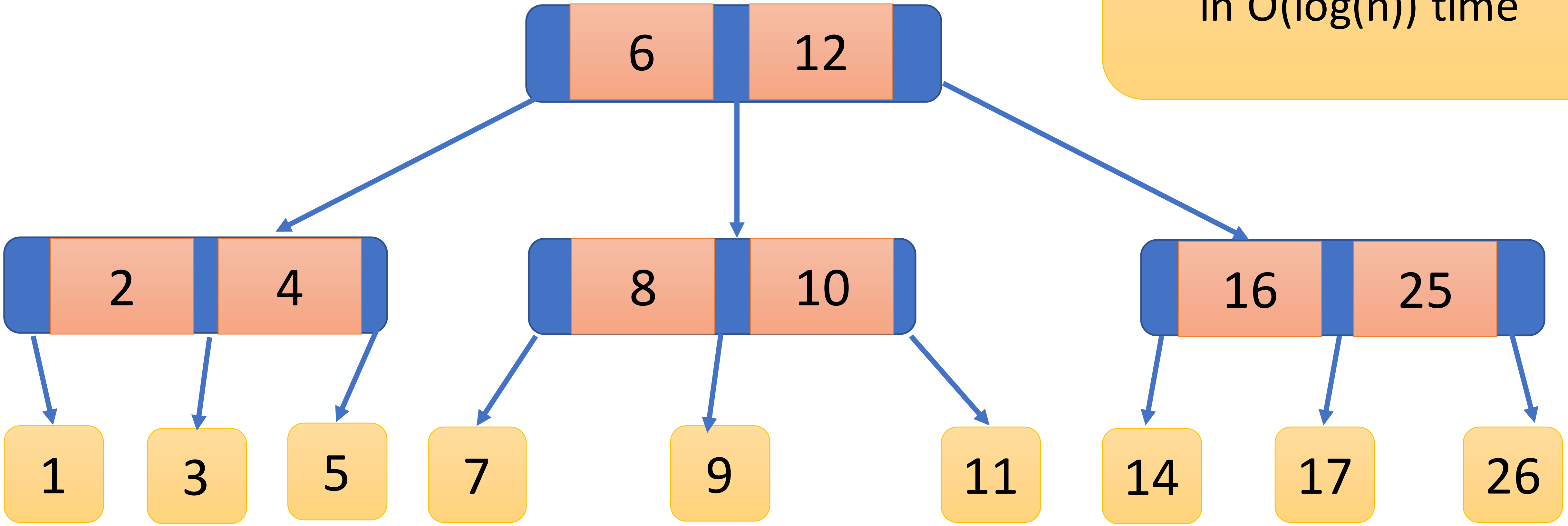
Answer: The red-**black** tree used to be the default choice. It is increasingly being replaced by the Maple tree (variant of the B-tree).

Faster and memory efficient

Hashes could do the job, but they are difficult to traverse in sorted order

What is a B-tree (structure that underlies a maple tree)?

All operations happen in O(log(n)) time



The Maple Tree

https://lwn.net/Articles/839781/

- It is a range-based B-tree
- In the Maple tree
  - Branching factor: 10 for non-leaf nodes and 16 for leaf nodes, 256-byte node size
  - Faster than traditional red-black trees
  - Optimized to fit data at cache line granularities
- Allows more parallelism
  - Different users can operate on different parts of the tree without interfering with each other.
  - They will remain isolated from each other most of the time (use less locks)
- They are used to managed "virtual memory regions"

What is anonymous and non-anonymous virtual memory?

- A file in the file system is defined as a contiguous array of bytes stored in a storage device like a hard drive
- A lot of files that a process uses do not exclusively belong to it.
  - The executable (opened in read-only mode)
  - Shared libraries (opened in read-only mode)
- Other files that exclusively belong to it can be mapped to the virtual memory space (easy to access). This is non-anonymous memory.
- Anonymous VM regions comprise the space on the heap that we create using *malloc* and *new*, and subsequently use

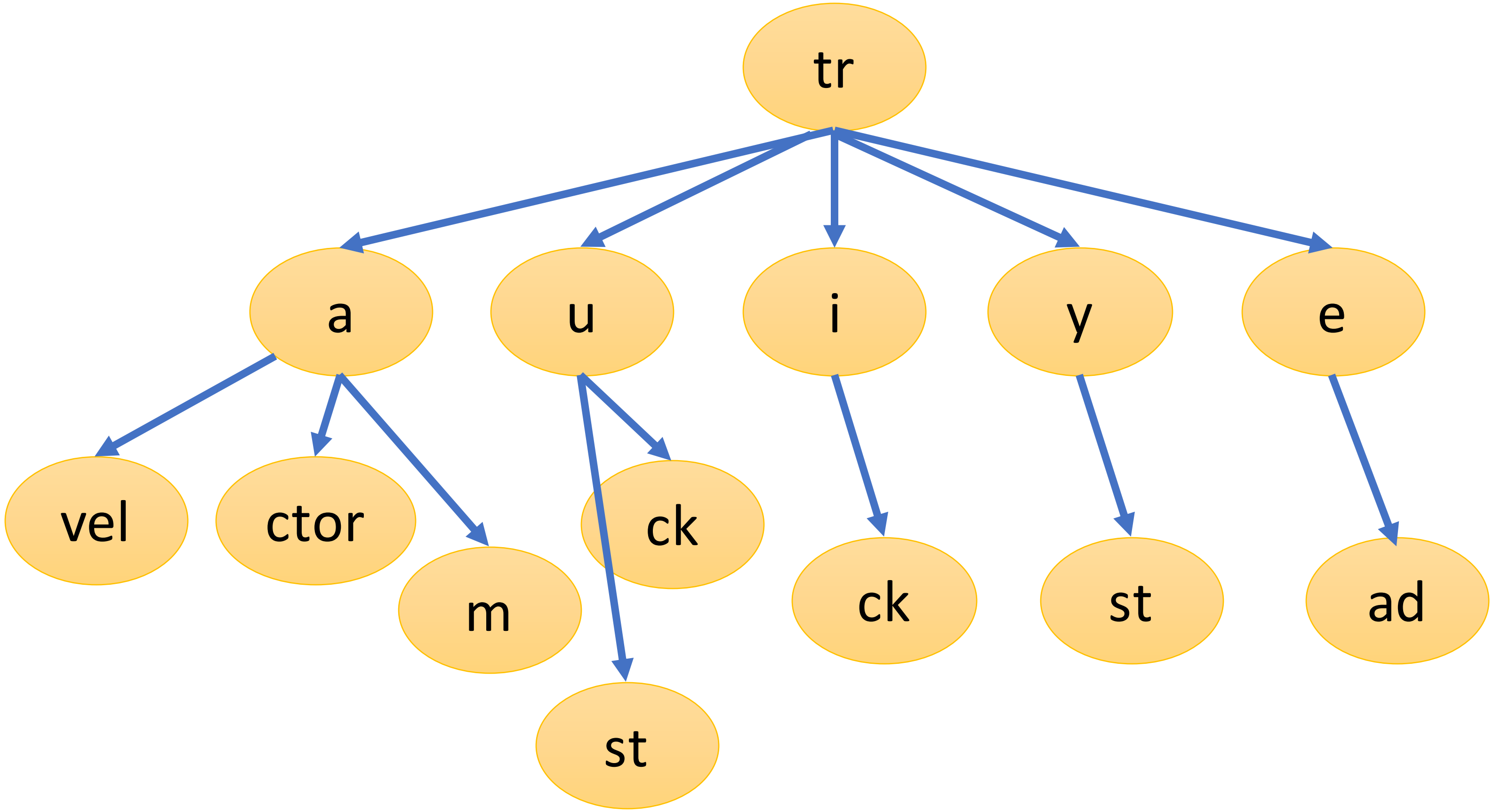
Field	Meaning
struct thread_info thread_info	Low-level information
uint state	Process state
void * stack	Kernel stack
Priorities	prio, static_prio, normal_prio
struct sched_info sched_info	Scheduling information
struct mm_struct *mm, *active_mm	Pointer to memory information
pid_t pid	Process id
struct task_struct *parent	Parent process
struct list_head children, sibling	Child and sibling processes
File system, I/O, synchronization, and debugging fields	

Radix Tree

Data structure fundamentals

strings

travel
truck
tram
trust
trick
tryst
tread
tractor



Found to be much faster than hashing-based solutions

The process id (pid)

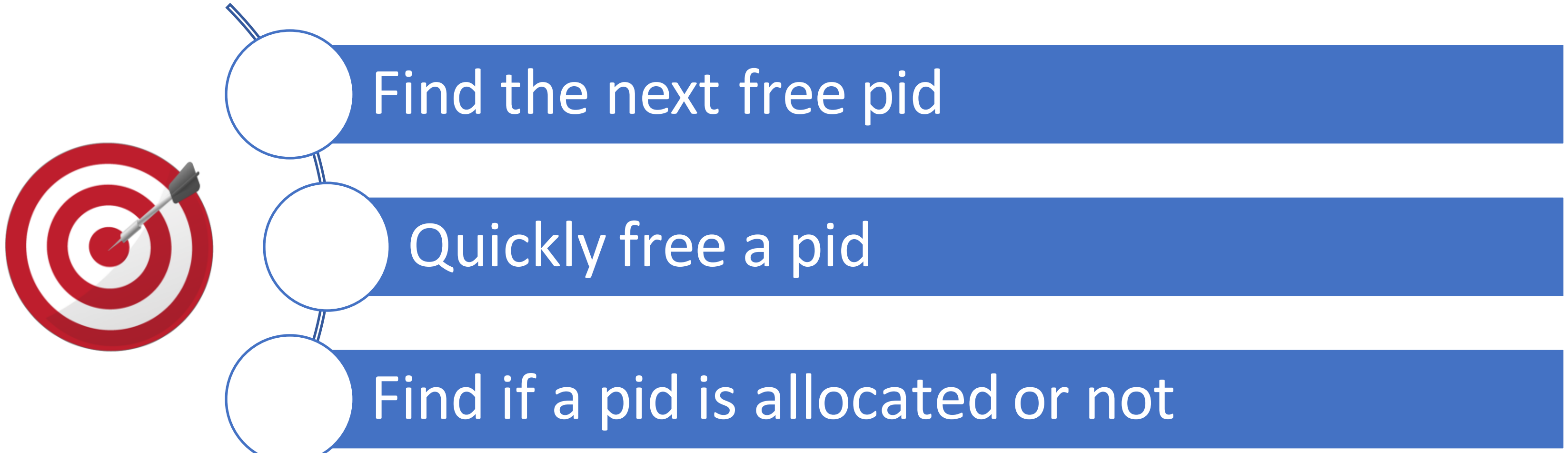
- Every process is uniquely identified by an integer: pid
- All the system calls and the kernel itself identify a process by its pid
- Processes can also be part of a group
    - This is known as a thread group
    - Every group has a tgid (thread group id)
    - All the threads in the group will have the same tgid
    - It is equal to the pid of the main thread (of the thread group)
- Linux also uses a *pid* structure (*struct pid*) to refer to a process that may have exited and its pid_t value reused.

Run the command: ps –LA

How are pids managed?

- The file /proc/sys/kernel/pid_max contains the maximum number of possible pids
- Defaults to 32,768
- There is a fundamental data structure question here.

❓ How do we manage the list of pids?



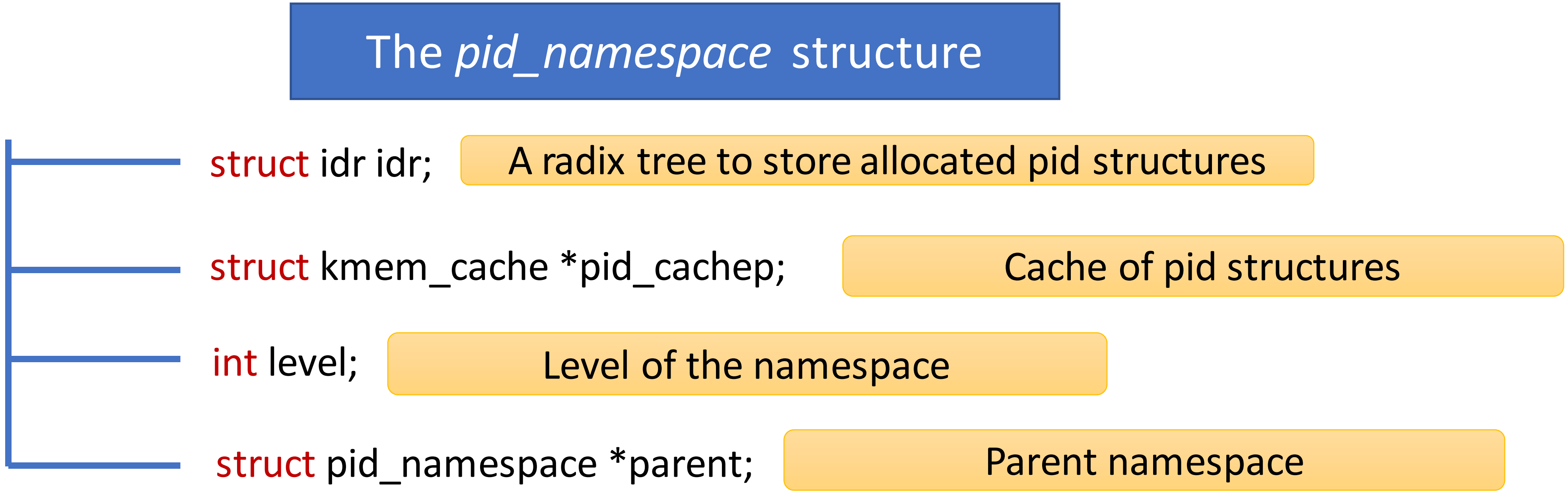
- Find the next free pid
- Quickly free a pid
- Find if a pid is allocated or not

Group *processes* into Namespaces

/include/linux/pid_namespace.h

- Overall, we divide the set of processes into namespaces
- A namespace is a set of processes that can only see each other
- Why?
  - Linux supports the notion of containers
  - A container is supposed to be an isolated “mini operating system”
  - Each container has its own process space and file system
  - Namespaces themselves have a hierarchical organization
  - A container can be suspended, resumed, and migrated
    - This means that all the constituent processes are suspended, resumed, and migrated
    - They shall continue to have the same pid numbers

Fields of the *pid_namespace* structure



The *pid_namespace* structure

struct idr idr; — A radix tree to store allocated pid structures

struct kmem_cache *pid_cachep; — Cache of pid structures

int level; — Level of the namespace

struct pid_namespace *parent; — Parent namespace

- Every namespace has a parent
- Hence, it has a level (the root namespace has level 1)
- Use a cache of pid structures
- Use a radix tree to find the next pid number

The pid Structure (abridged view)

```
struct upid {
        int nr;
        struct pid_namespace *ns;
};

struct pid
{
        refcount_t count;
        unsigned int level;

        /* lists of tasks that use this pid */
        struct hlist_head tasks[PIDTYPE_MAX];

        /* wait queue for pidfd notifications */
        struct upid numbers[1];
};
```

pid number

Pointer to the namespace

Tasks that use this pid → represents a task group

Array of *upids* (one per level)

Allocating a *pid* structure

- Call the *alloc_pid* function defined in kernel/pid.c
- Use as software cache
  - The namespace has an element called *pid_cachep* that is a cache of pid structures
  - Fetch an entry from the software cache
  - This is a fast process. There is no need to allocate a new *pid* structure
- Note that a process may be a part of many namespaces
  - Its namespace and all ancestor namespaces
  - Allocate a *pid* number in each ancestor namespace
- At each level, keep adding the *pid structure* to the radix tree at each level

How do you use a radix tree here?

- Store all the processes in a radix tree
- The key is the process id, and the value is the ptr to the *pid* structure
- This works like a hashtable. Faster than a real hashtable in practice.
- A radix tree works well when the keys share prefixes
  - This is indeed the case with process ids (think about it …)

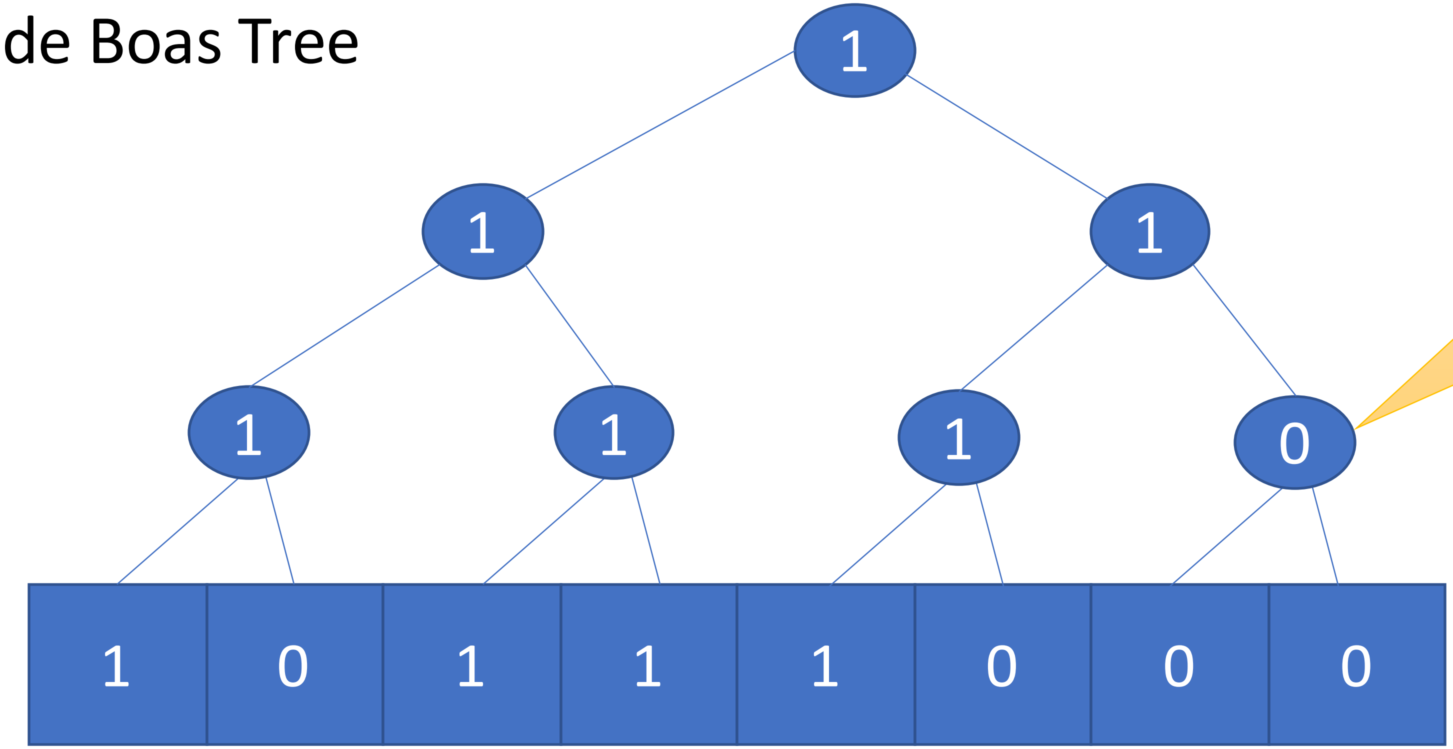
Next Problem: Find a free process id

- Create a bitmap: 1 bit per process.
- If there is a maximum of *K* processes, then we have a large K-bit bitmap
- To find a free entry in the bitmap
  - Maintain a bitmap of all process ids in a given range (1 if free, 0 if not free)
  - Problem: Given a starting index, find the next index that is free
  - Linux uses sequential search that has some smart features
  - Traverse long word by long word (not bit by bit)
  - It uses the built in *bsf* instruction to find the first 1 bit set in a long word

Function: radix_tree_find_next_bit in /lib/radix-tree.c

This process can be accelerated

- Van Emde Boas Tree



Does this sub-tree have a free entry?

We can also store multiple bits in each leaf node or internal node.

Possible to find the next free entry in O(log(n)) time

Incorporate the Van Emde Boas Tree in the Radix Tree as Linux Does

- Idea:
  - Split the bitmap among the leaf nodes of the radix tree. Each internal node contains a single bit indicating if the sub-tree rooted at it has a free entry or not.
  - We can start allocating a new *pid* from 0 or from a given *process's* pid such as the parent process
  - Let us refer to this point as the *starting point*
  - Once you reach the right leaf in the radix tree (the *starting point*), start searching in the bitmap chunk towards greater indices until you find a free entry.
  - If you don't find one, then search the next bitmap chunk (greater values), and so on till a free entry is found.

Radix tree
+
Van Emde Boas tree
=
Linux IDR tree

Field	Meaning
struct thread_info thread_info	Low-level information
uint state	Process state
void * stack	Kernel stack
Priorities	prio, static_prio, normal_prio
struct sched_info sched_info	Scheduling information
struct mm_struct *mm, *active_mm	Pointer to memory information
pid_t pid	Process id
struct task_struct *parent	Parent process
struct list_head children, sibling	Child and sibling processes
File system, I/O, synchronization, and debugging fields	

File System, I/O, and Debugging Fields

- struct fs_struct *fs;

Pointer to the file system that this process uses.

- struct files_struct *files;

List of files opened by the process

- struct signal_struct *signal;

List of registered signal handlers for the process.
A signal handler for a signal is a function that is called when a process receives the signal from the OS.

Few more I/O related fields

- struct bio_list *bio_list;

Block device information (like hard disks)

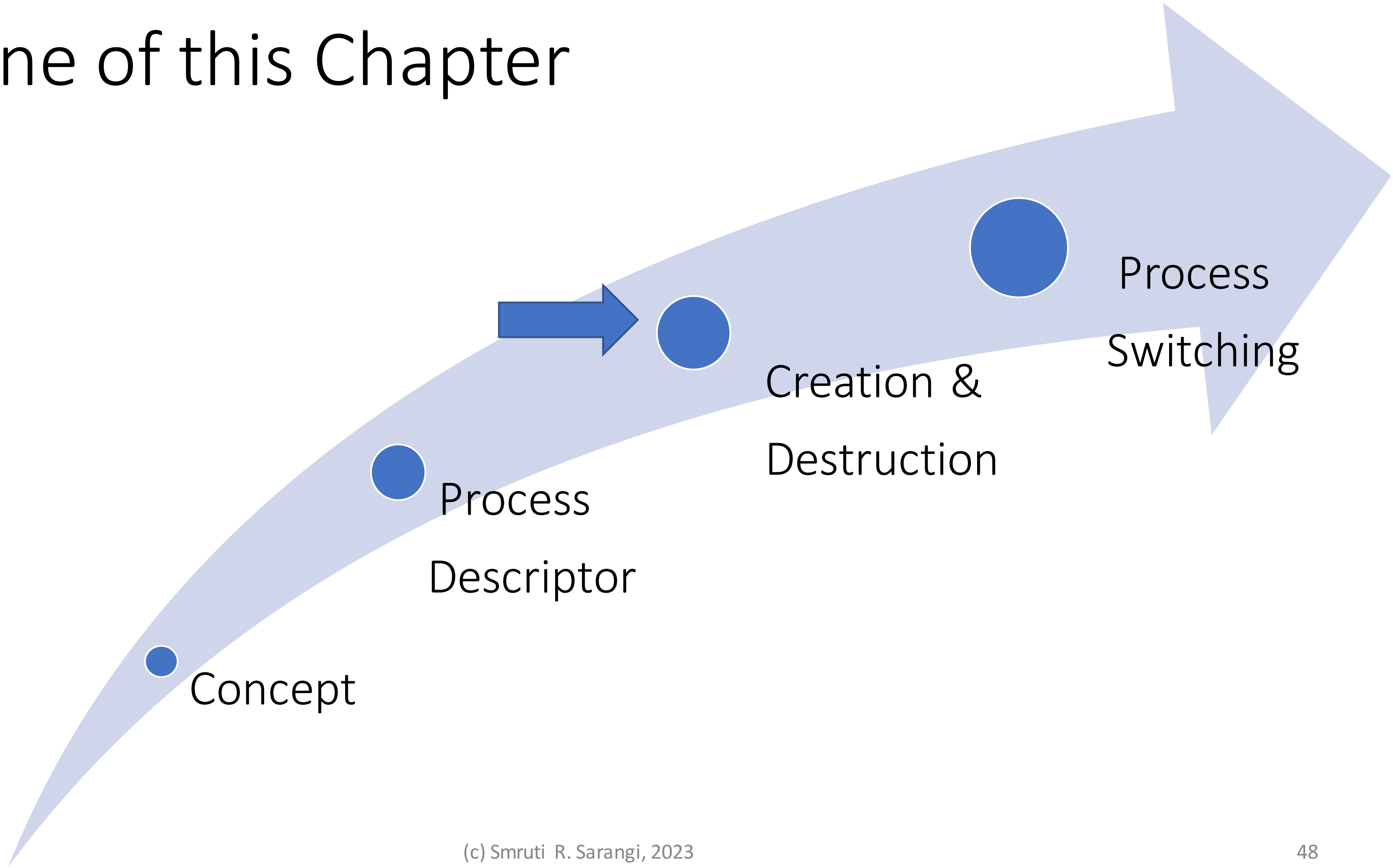
- struct io_context *io_context;

I/O subsystem state of the associated processes

The *ptrace* Mechanism

- It allows the parent process to observe and control the execution of a child process
- This is the crucial piece of technology that allows debuggers to run
- One process can pretty much take over another process
- A process can be traced
  - If the process is being traced then the tracking process gets the control
  - The task_struct structure has a field *unsigned int ptrace;*
  - The flags in this field enable the ptrace functionality
- Whenever there is an event of interest like a *fork* or *syscall*
  - The traced process stops
  - A SIGTRAP signal is sent to the tracking process
  - It runs a signal handler
  - This can inspect the state of the tracked process, and change its system call params

Outline of this Chapter



Concept

Process Descriptor

Creation & Destruction

Process Switching

Process Creation and Destruction

- Creating and destroying processes are essential to running and finishing programs
- Linux's approach may appear weird at the beginning
- It will start making sense gradually …
- The first process that the kernel runs has a pid 0 (the *idle* process)
- Then it runs the *init* process with a pid 1 (after booting)

The *fork* mechanism

The basic idea is that the kernel creates only one process during boot time: the *init* process

All child processes are created by essentially cloning the parent process. For example, the first few processes clone *init*. A fork is a type of cloning.

start_kernel and init

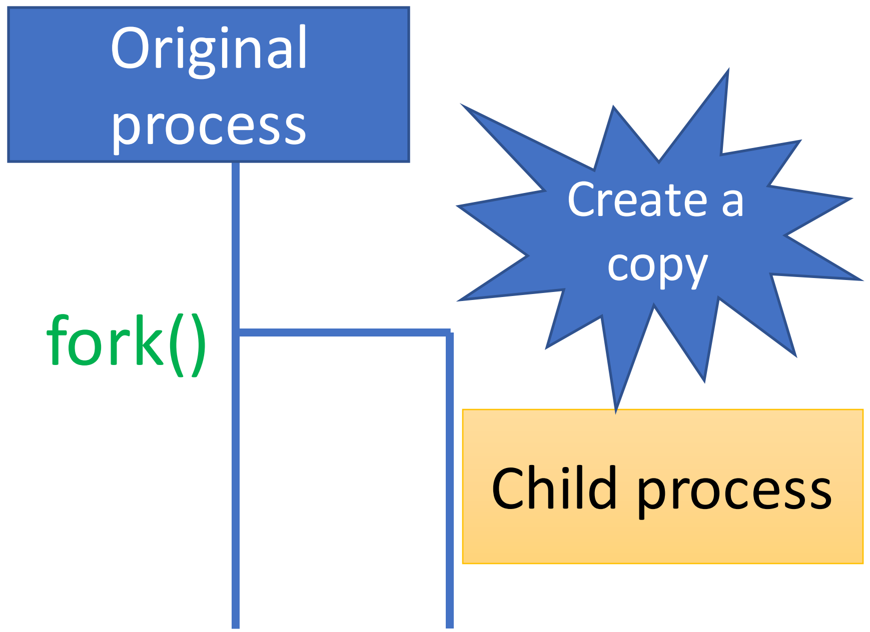
- The *start_kernel* function forks the *init* process
- The *init* process transitions from kernel mode to user-space mode
- This happens by a call to *execve* (in user space) discussed later
- A rare instance where the parent process is in kernel space, and the forked process is in user space
- The user space now starts executing …

The *fork* system call

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

int main( void ) {
    int pid = fork();

    if (pid == 0) {
        printf( "I am the child \n" );
    } else {
        printf( "I am the parent: child = %d\n", pid );
    }
}
```



pid = 0 for the child

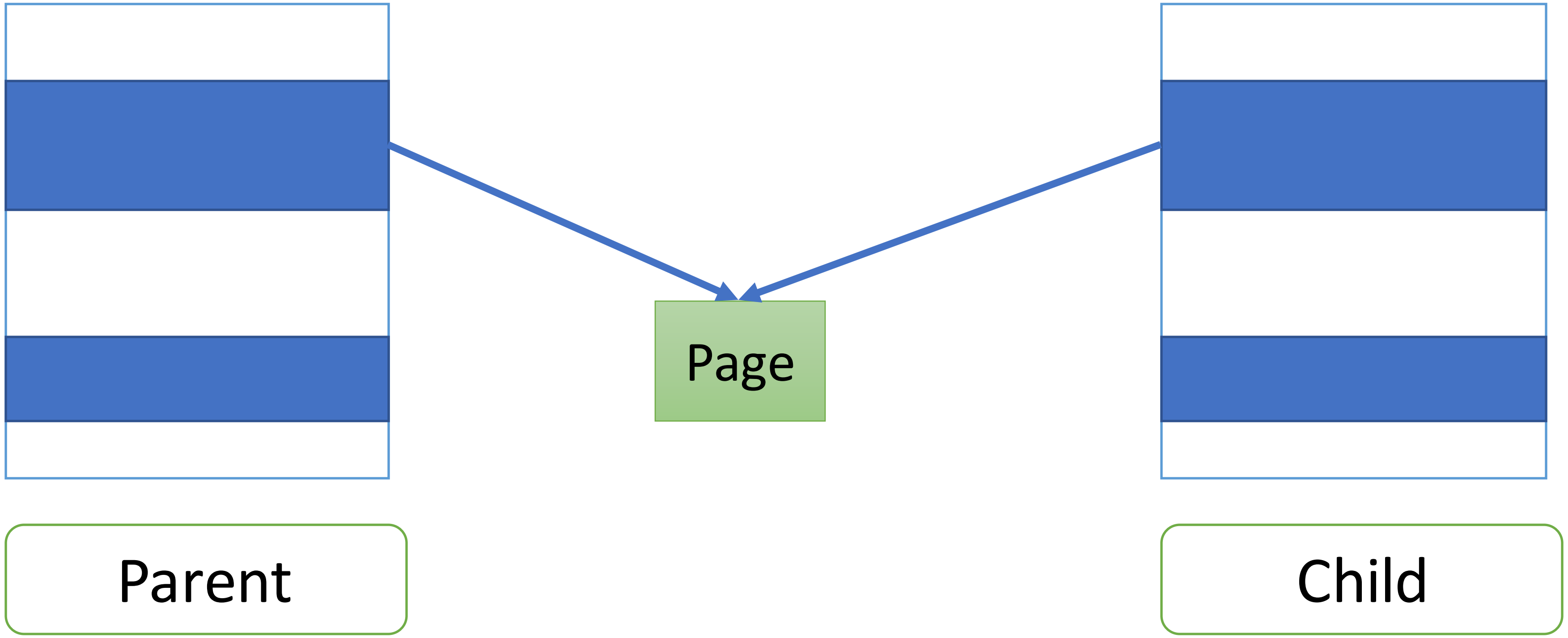
The parent gets the pid of the child

More about the *fork* function call

- A new child process is created.
- The process id (pid) of the child is the return value of the fork function call (for the parent)
- What about the child process?

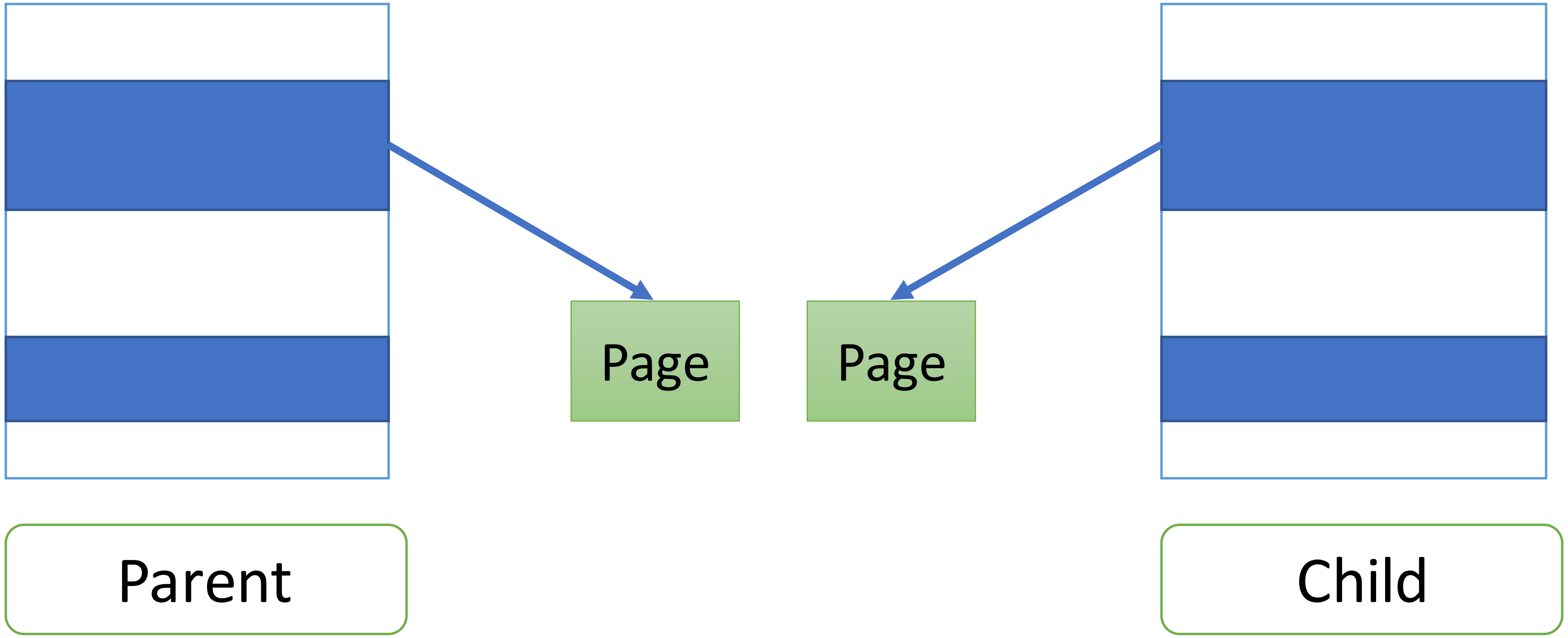
- All the memory regions of the parent are copied.
- The process state is copied
- The program counters are set to the same value
    - They are albeit in different processes (different address spaces)
    - Both point to the next instruction after the fork system call
    - The child gets 0 as the return value of the fork system call

Copying the address space



- Just copy the page tables.
- The address space is effectively copied.

Create a copy when there is a write



- If there is a write by any process (parent or child), just create a copy of the page, and map it to the chid process.
- This is known as the copy-on-write mechanism (saves a lot of memory)

How do you know that there is a write?

- Every TLB entry and page table entry has additional bits to store page-wise permissions
- For example, read-only pages have the READONLY bit set
- Let us have one more read-only permission bit. Let us call it P2.
- When we *fork* a process, we set the value of P2 to 1 (read only) for all the pages of both the parent and child process
- When the parent or child process try to write to a page with its P2 bit set and the READONLY bit set to 0
  - We create a copy of the page
  - The P2 bits for both the pages (original and the copy) are set to 0

The *excecvp* system call

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

#define PWDPATH "/usr/bin/pwd"

int main( void ) {
    char *argv[2] = {"pwd",NULL};
    int pid = fork();

    if (pid == 0) {
        execvp (PWDPATH,argv);
    } else {
        printf( "I am the parent: child = %d\n", pid );
    }
}
```

The child may decide to create its brand new avatar.

The child process executes a new command. Replaces itself → contents of its memory space

The *exec* family of system calls

- Clean up the memory space of a process
- Load the starting memory state of the executable specified in the *exec system call*:
  - Setup the text, data, and bss sections.
  - Initialize the stack and heap sections
- Maintain the other resources that the process used to own: open files, network connections, etc.
- Start executing from the beginning of the text section

The *fork* and *clone* calls (in detail)

/kernel/fork.c

- There are many variants of the *fork* and *clone* system calls
- All of them finally end up in the *copy_process* function

struct task_struct copy_process (struct pid *pid, …, …., …)*

1. Duplicate the current task_struct
   I. Allocate a new task
   II. Duplicate the architectural state (e.g: floating-point state)
   III. Setup the kernel stack
   IV. Add other bookkeeping information
   V. Set the time that the child task has run to zero
   VI. Assign this task to a CPU
   VII. Allocate a pid

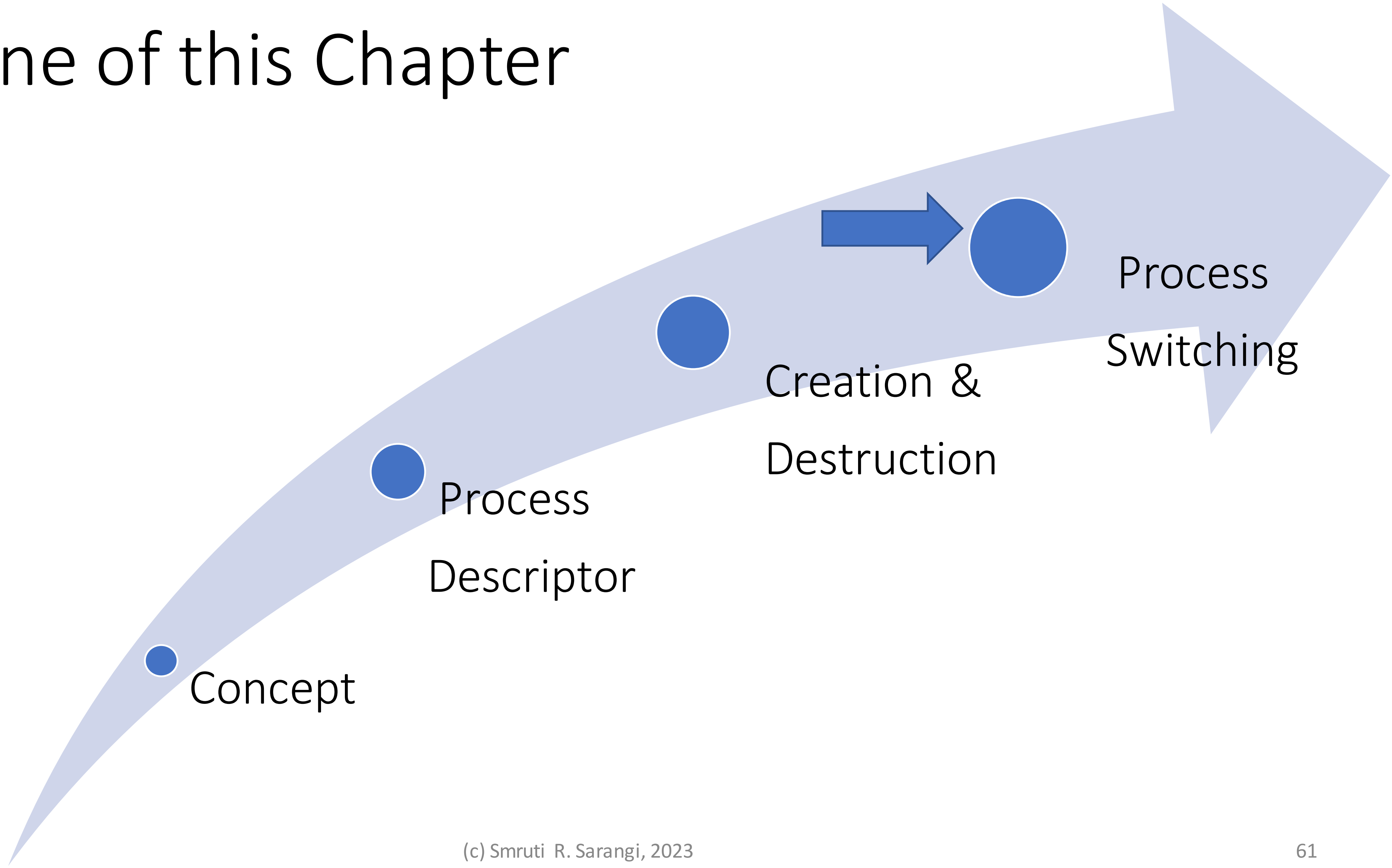
Continuation …

2. Copy of all the information about open files, network connections, I/O, and other resources from the original task
   I. Copy connections to open files
   II. Copy the reference to the current file system
   III. Signal handler information
   IV. Copy the virtual address memory map (in the mm_struct)
   V. Copy namespaces and I/O permissions
3. Fix the relationships
   i. Add the new task to the children list of the parent
   ii. Fix the parent and sibling list of the new task
   iii. NOTE: In a multi-threaded process, only the calling thread is forked

Kernel Threads

- Linux distinguishes between user threads, I/O threads, and kernel threads
- It defines special functions like *kernel_clone* to create kernel threads
- The *flags* field in the *task_struct* has this information
- All the kernel threads are descendants of kthreadd (process id: 2)
- It is like *init* for kernel threads
- They are created using *kthread_create ()* (defined in kernel/kthread.c)
- Used primarily for periodic book-keeping tasks, timers, interrupt handling, I/O device interfacing, etc.

Outline of this Chapter



Concept

Process Descriptor

Creation & Destruction

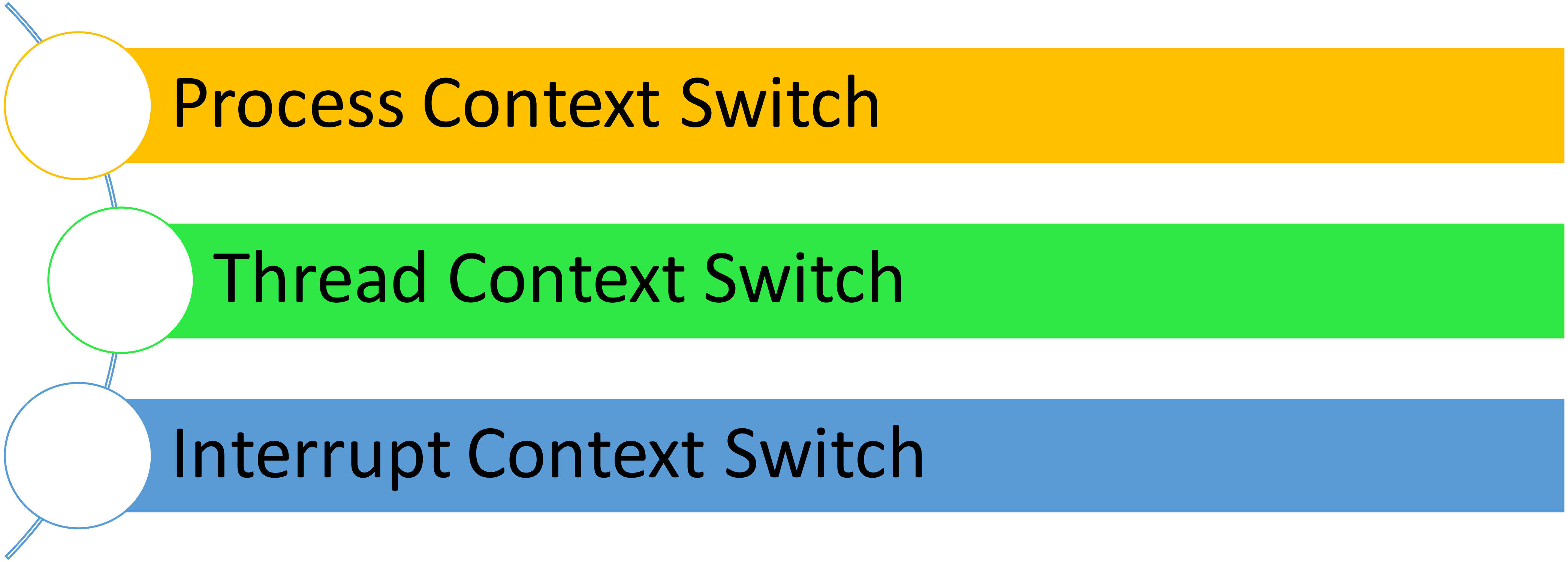
Process Switching

General Principles

Internals of the Context Switch Process

- Every process has a hardware context
- It is the value of all the registers that are associated with the process
  - General-purpose registers
  - Program counter (also known as the instruction pointer)
  - Segment registers
  - Privileged registers such as CR3 (starting address of the page table)
  - ALU and floating-point unit flags
- The hardware context needs to be saved and restored
- The pointers to the page table and the contents of the TLB need to be changed
- The software context (open files, network connections) is comparatively much easier to manage: does not need to be stored and restored

Types of Context Switches



Process Context Switch

Thread Context Switch

Interrupt Context Switch

Context Switching Types

Process Context Switch

- Consider a regular user process that enters the kernel after a system call
- There is no need to create a new kernel process
- Let the same process continue, albeit in “kernel mode”
- We still need to save the PC and registers (and restore them later)
- This is a soft switch from user to the kernel mode and vice versa
- However, this is less complicated than a full-scale process switch
- The virtual address space can be the same as long as we use different virtual addresses in kernel mode, and use a kernel-specific stack.

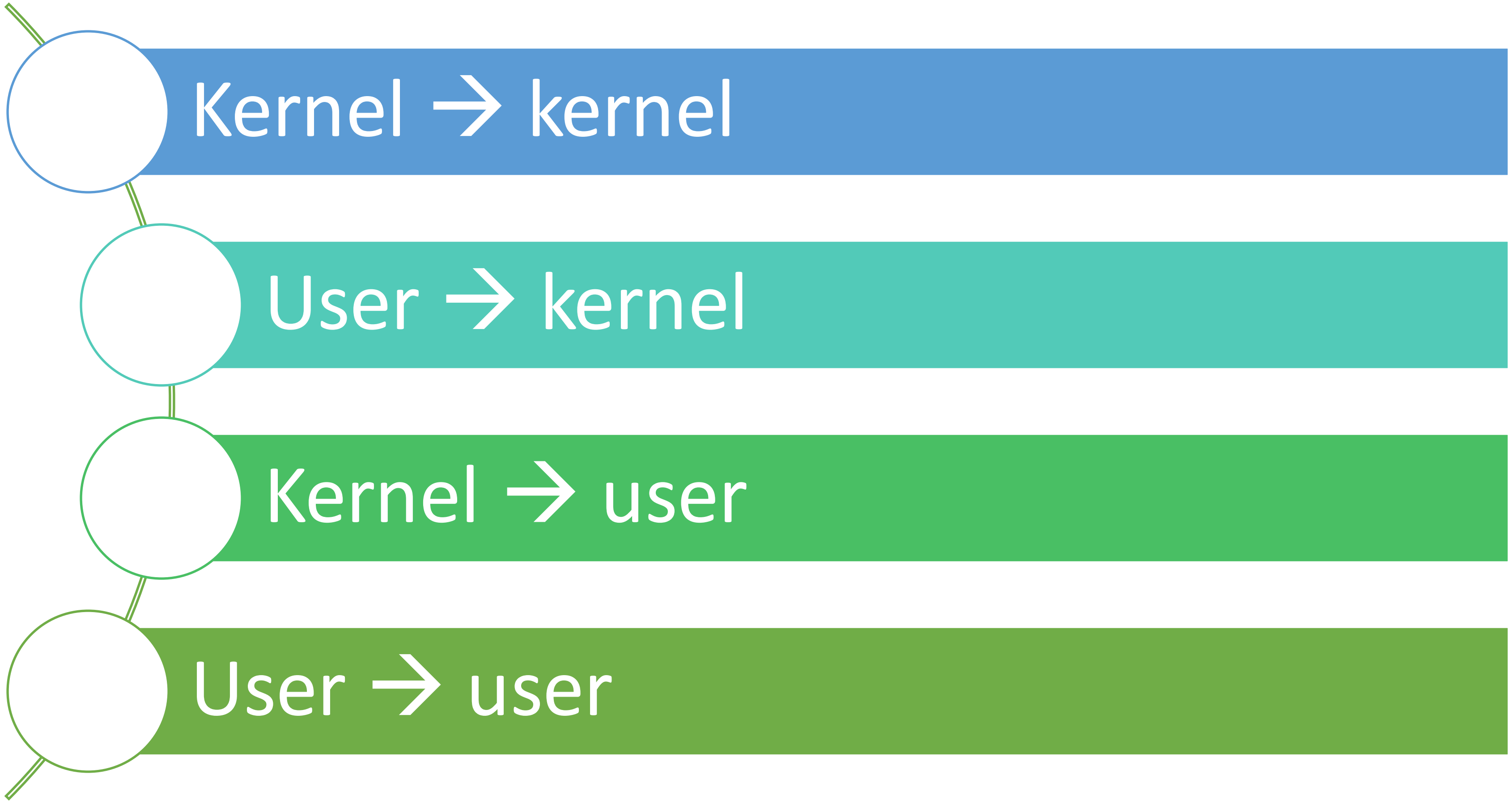
Thread Context Switch

- A thread is the atomic unit of scheduling in the OS
- A thread however cannot own resources, the thread group however can
- Switching threads that belong to the same thread group is easier than a full process context switch
    - Replace only those parts of the virtual ↔ physical mapping that are private to a thread, notably the stack
    - Do the same with registers (general-purpose registers, flags and the PC)
    - Change the *current* pointer to refer to the *task_struct* of the new thread

Interrupt Context Switch

- Whenever, a HW interrupt arrives, we need to service it quickly
- We cannot continue to run the same thread/process
- The interrupt handler may need a new kernel thread of its own or may continue to use the same thread (one that was interrupted)
- Most interrupt handlers typically consist of two parts:
  - Top half – Short piece of code subject to many restrictions. Does basic interrupt processing.
  - Bottom half – This is a full-fledged kernel task that can execute later. It often does the bulk of the interrupt processing
- ❌ The top half accesses variables in a separate virtual address space. Hence, changing the TLB mappings is not required.

Four Types of Switches in the Linux Kernel



- Kernel → kernel
- User → kernel
- Kernel → user
- User → user

Details of the Context Switch Process

Store the State (Basic Operations)

/arch/x86/entry/entry_64.S

- The job of the functions and macros in this file are to store the state of the executing thread
- *entry_syscall_64* function is the only entry point for system calls on 64-bit x86 machines.
  - Note the SYM_CODE_START directive. Declares a function written in the assembly language
  - We need to be very careful in saving the state
  - Some model specific registers (MSRs) are typically used

Steps for Saving the Context after a *syscall*

- The hardware stores *rip* (PC) to *rcx*, and stores *rflags* in *r11*
    - If it is an interrupt, then MSR registers and dedicated memory areas perform the same role. In x86, the values of *rip, CS* (code segment), and *rflags* are pushed to the stack by HW.
- Call the ***swapgs*** instruction to store the contents of the *gs* register in a pre-specified address (stored in an MSR)
- Store the stack pointer (*rsp*) in a dedicated memory region (in the task state segment (TSS))
- Set the stack to the kernel stack

Continuation …

- Push *DS, rsp (from TSS), r11*, *CS,* and *rcx* onto the kernel stack
- Push the rest of the general-purpose registers to the kernel stack

Need to disable interrupts on the local processor during this process

Follow the reverse process while returning from the kernel

sysret and *iret* instructions

- *sysret* is the opposite of *syscall*
  - Transfers the contents of *rcx* to *rip*
  - Transfers the contents of *r11* to *rflags*
- What happens if an interrupt arrives between setting *rsp* (user stack) and executing sysret?
  - The interrupt handler can still execute.
  - Let it use its separate stack (recall the interrupt stack table)

- *iret*
  - Restore the values of *rip, CS,* and *rflags* from the stack
  - Setting the value of *rip* is equivalent to a jump to the user program

Additional Context

/arch/x86/include/asm/processor.h

thread_struct

- Cache of TLS (thread local storage) descriptors (base-limit form)
- Stack pointer
- es, ds, fs, and gs segment register values
- I/O permissions: io_bitmap
- Floating-point unit state

Then what ….

- Once the context is saved, there is often no need to create a separate kernel thread
- The same thread can continue to execute using the kernel stack (of course)
- Service the interrupt of system call
- Check if there is additional work to do (that has a higher priority)

exit_to_user_mode_loop in /kernel/entry/common.c

- If there is work to be done, then call the scheduler (schedule() function)
- The scheduler will find the appropriate task to run next
- It calls the *context_switch* function

Context Switch Process

/kernel/sched/core.c

- context_switch (run queue, prev task, next task)
  - prepare_task_switch
  - arch_start_context_switch
  - switch the memory structures
  - switch_to (switch the register state and stack)
  - finish_task_switch

Basic Steps

- *prepare_task_switch* → Set the state of the *prev* task
  - It is not running any more
- Switch the memory maps (mm_struct structures)
  - The TLB contents need to be changed

Switching to the kernel

```
if (! next->mm)
        next->active_mm = prev->active_mm;

    if (prev->mm)
            mmgrab (prev->active_mm);
    else
            prev->active_mm = NULL;
}
```

Use the *mm* of the previous task

Increase the reference count

Coming from user space

Coming from the kernel

Switching to User Space

- From Userspace
  - Manage states of interrupt queues and do other book-keeping activities
- From the kernel
  - prev->active_mm = NULL

Call the __switch_to function in
/arch/x86/kernel/process_64.c

__switch_to *function*

- Extract the *thread_struct* structures
- Load the TLS (thread local state)
  - *fs* and *gs* segment registers
  - Load the rest of the segment registers
- Change the *current* ptr and the stack pointer
- Set the floating-point unit state
- Restore the state of model specific registers

finish_task_switch

- Set the state of the *prev* task and *next* task
- Load the *kmap* for the task
  - Maps user space pages to the kernel address space

Interesting Trivia

- You will often find statements of the form:
  - *if (likely (<some condition>) { …. }*   OR
  - *if (unlikely (<some condition>) { …. }*
- They are hints to the branch predictor of the CPU
  - This branch is most likely to be taken

- You will often find statements of the form:
  - static *__latent_entropy* struct task_struct *copy_process (…){…}
  - We are using the value of the task_struct* pointer as a source of randomness
  - Many such random sources are combined to create cryptographic keys

srsarangi@cse.iitd.ac.in