

Jin-Soo Kim
(jinsoo.kim@snu.ac.kr)

Systems Software &
Architecture Lab.
Seoul National University

Fall 2020

Processes and Threads

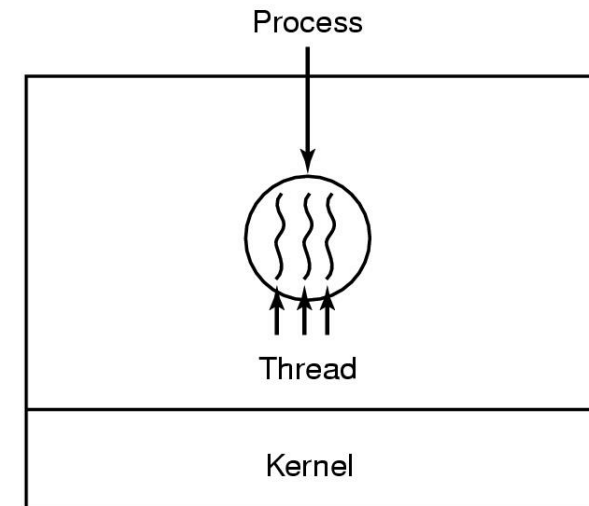
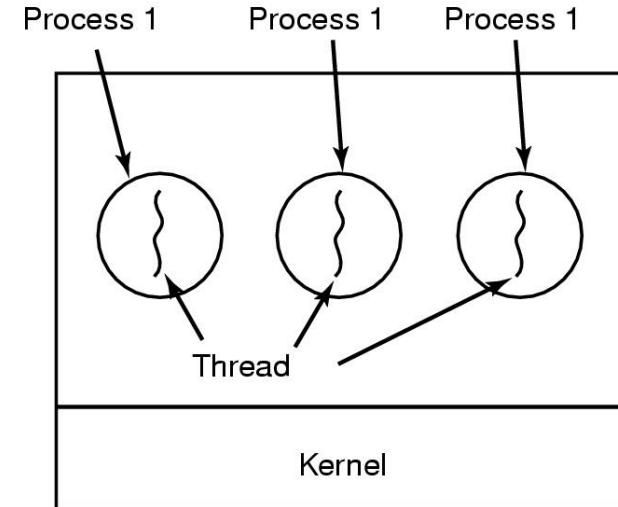


What is a Process?

- A(An) _____ of a program in execution
- Java analogy:
 - Class → “program” (static)
 - Object → “process” (dynamic)
- The basic unit of protection
- A process is identified using its process ID (PID)
- A process includes
 - CPU context (registers)
 - OS resources (**address space**, open files, etc.)
 - Other information (PID, state, owner, etc.)

What is a Thread?

- A thread of control:
A sequence of instructions being executed in a program
- A thread has its own
 - Thread ID
 - Set of registers including PC & SP
 - Stack
- Threads share an address space
 - Code, Data, and Heap
- Separate the concept of a process from its execution state



Why Threads?

- Creating concurrency is cheap
- Improves program structure
 - Divide large task across several cooperative threads
- Throughput
 - By overlapping computation with I/O operations
- Responsiveness
 - Can handle concurrent events (e.g., web servers)
- Resource sharing
- Utilization of multi-core architectures
 - Allows building parallel programs

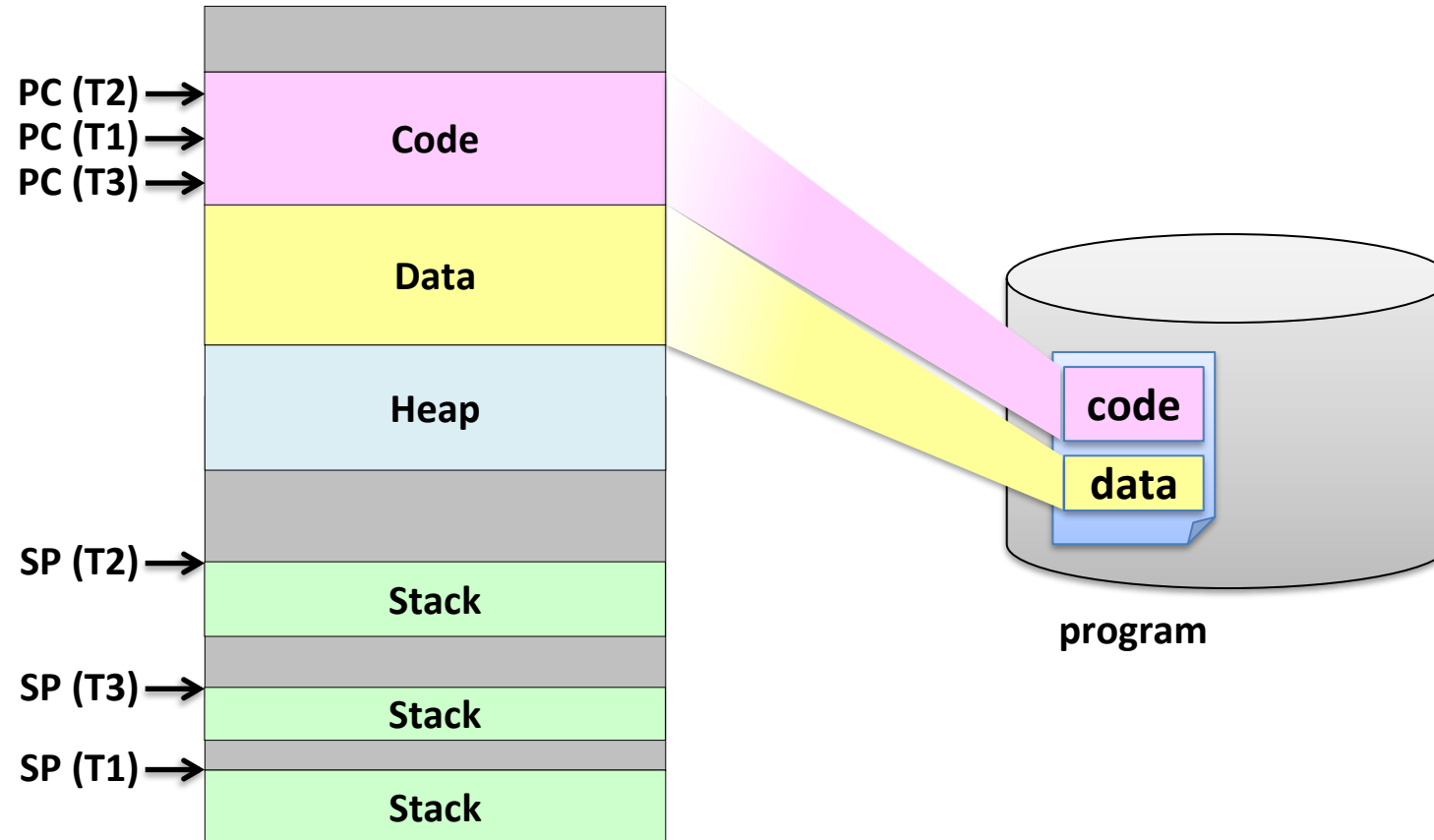
Processes vs. Threads

- A thread is bound to a single process
- A process, however, can have multiple threads
- Sharing data between threads is cheap; all see the same address space
- Threads are the unit of scheduling
- Processes are containers in which threads execute
 - PID, address space, user and group ID, open file descriptors, current working directory, etc.
- Processes are static, while threads are dynamic entities



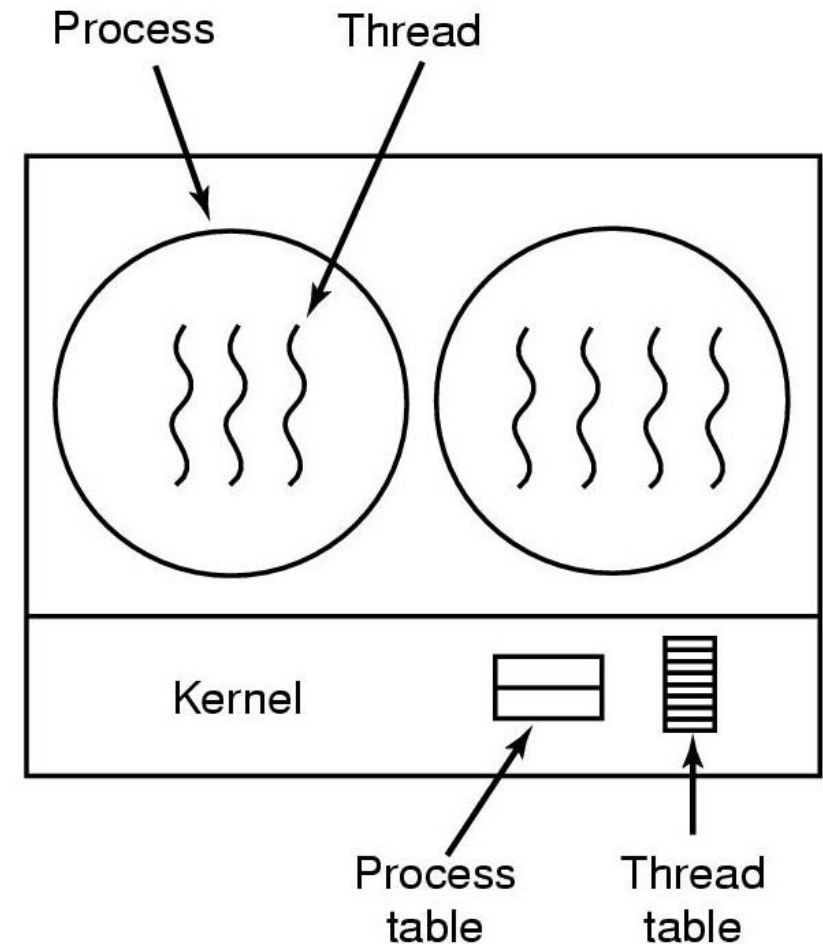
Image source: <https://dribbble.com/shots/1395795-factory-cross-section-progress-4>

Address Space with Threads



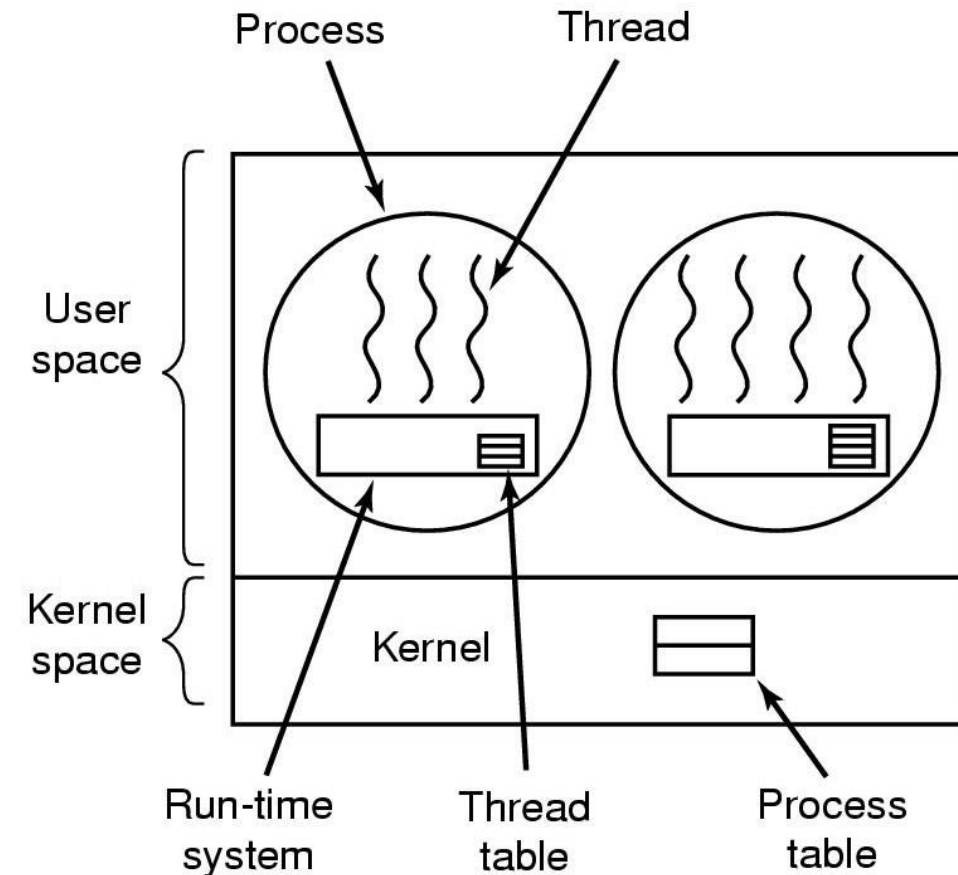
Kernel-level Threads

- OS-managed threads
 - OS manages threads and processes
 - All thread operations are implemented in the kernel
 - Thread creation and management requires system calls
 - OS schedules all the threads
 - Creating threads are cheaper than creating processes
 - Windows, Linux, Solaris, Mac OS X, AIX, HP-UX, ...



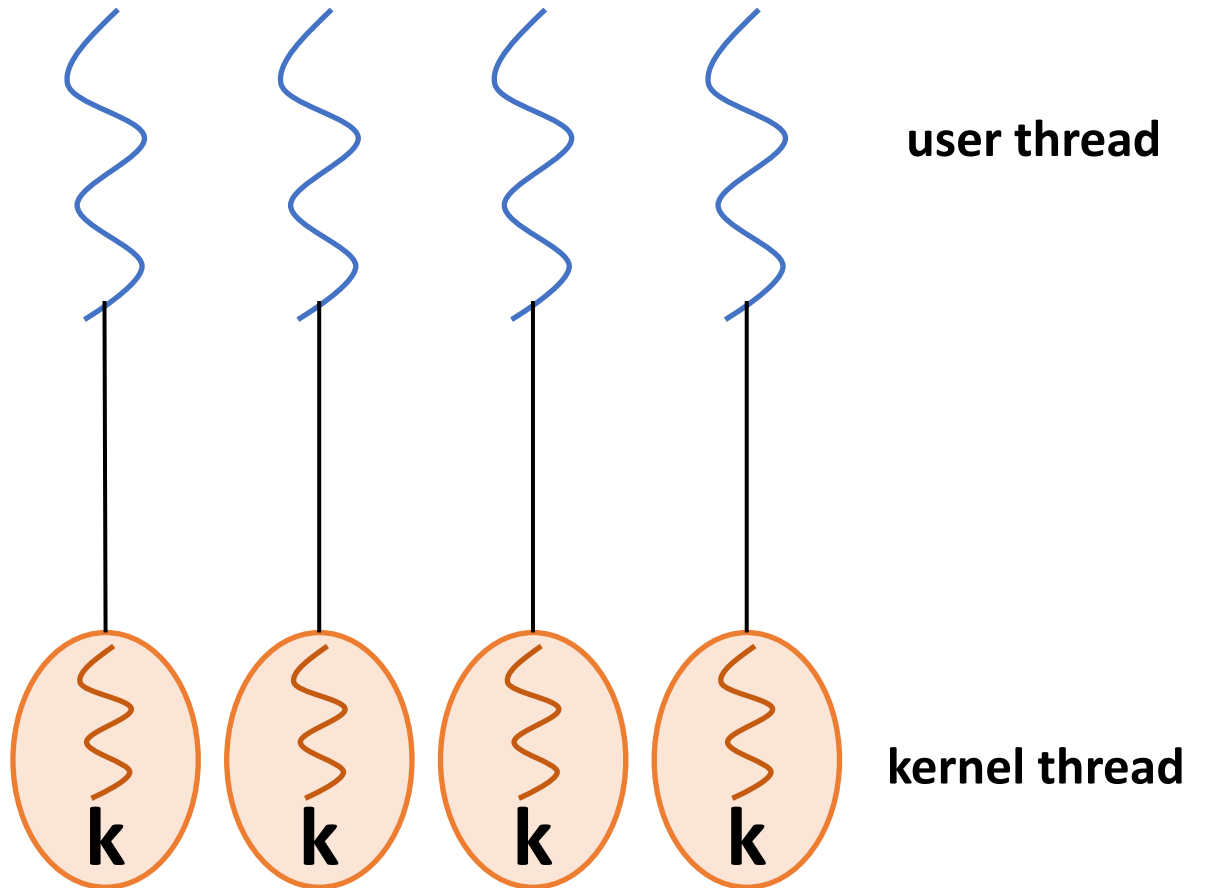
User-level Threads

- Threads are implemented at the user level
 - A library linked into the program manages the threads
 - Threads are invisible to the OS
 - All the thread operations are done via procedure calls (no kernel involvement)
 - Small and fast:
10-100x faster than kernel-level threads
 - Portable
 - Tunable to meet application needs



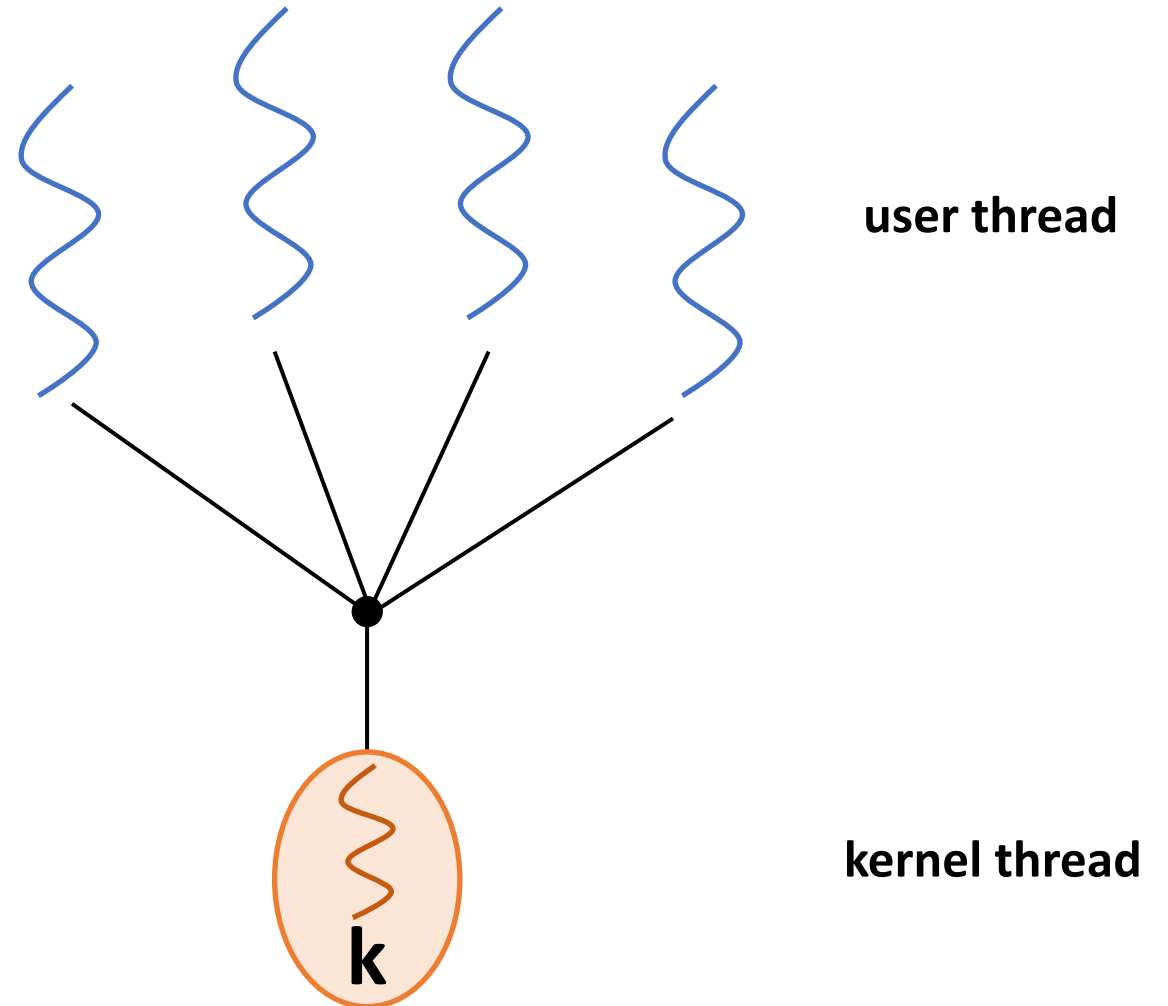
Threading Model: One-to-One (1:1)

- Each user-level thread maps to a kernel thread
- Most popular
- Windows XP/7/10, OS/2, Linux, Solaris 9+



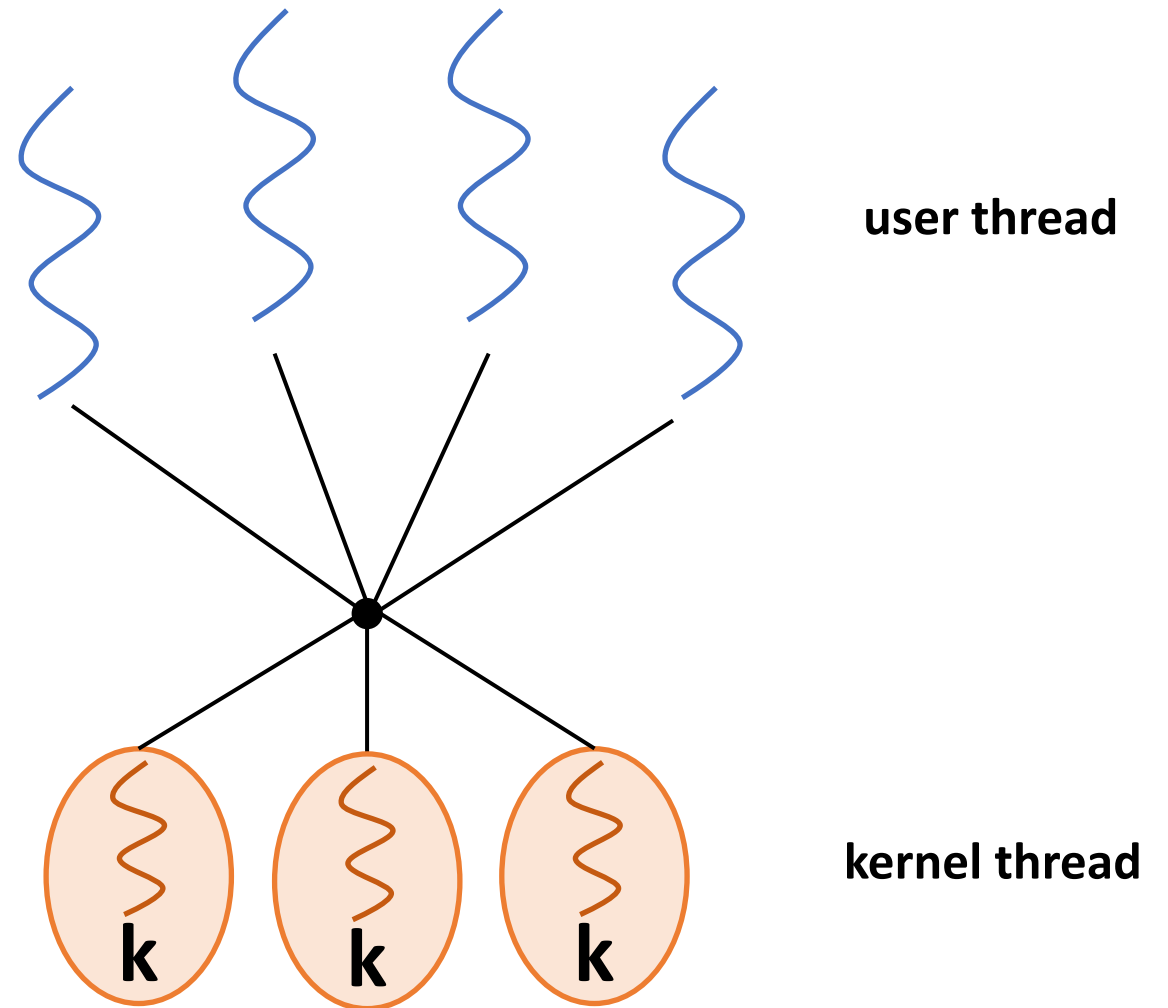
Threading Model: Many-to-One (N:1)

- Many user-level threads mapped to a single kernel thread
- Used on systems that do not support kernel-level threads
- Solaris Green Threads, GNU Portable Threads



Threading Model: Many-to-Many (M:N)

- Allows many user-level threads to be mapped to many kernel threads
- Allows the OS to create a sufficient number of kernel threads
- Solaris prior to v9, IRIX, HP-UX, Tru64



OS Classification

# threads per addr space:	# of addr spaces:	One	Many
One		Embedded Systems without OS MS/DOS Early Macintosh	Traditional UNIX
Many		Many Embedded OSes (VxWorks, QNX, μ Clinux, μ C/OS-II, ...)	Modern OSes (Mach, Windows, Linux, Mac OS X, HP-UX, Solaris, AIX, ...)

Processes and Threads in Linux

Linux Tasks

■ Tasks

- In Linux, tasks represent both processes and threads
- Each task is described using a task structure

■ struct task_struct

- @ include/linux/sched.h
- Everything the kernel has to know about a task
- About 3.5KB in size (Kernel 5.4.59 on x86_64)
- Allocated by the slab allocator (cf. [/proc/slabinfo](#))
- Task list (`t->tasks`): the list of task structures in a circular linked list

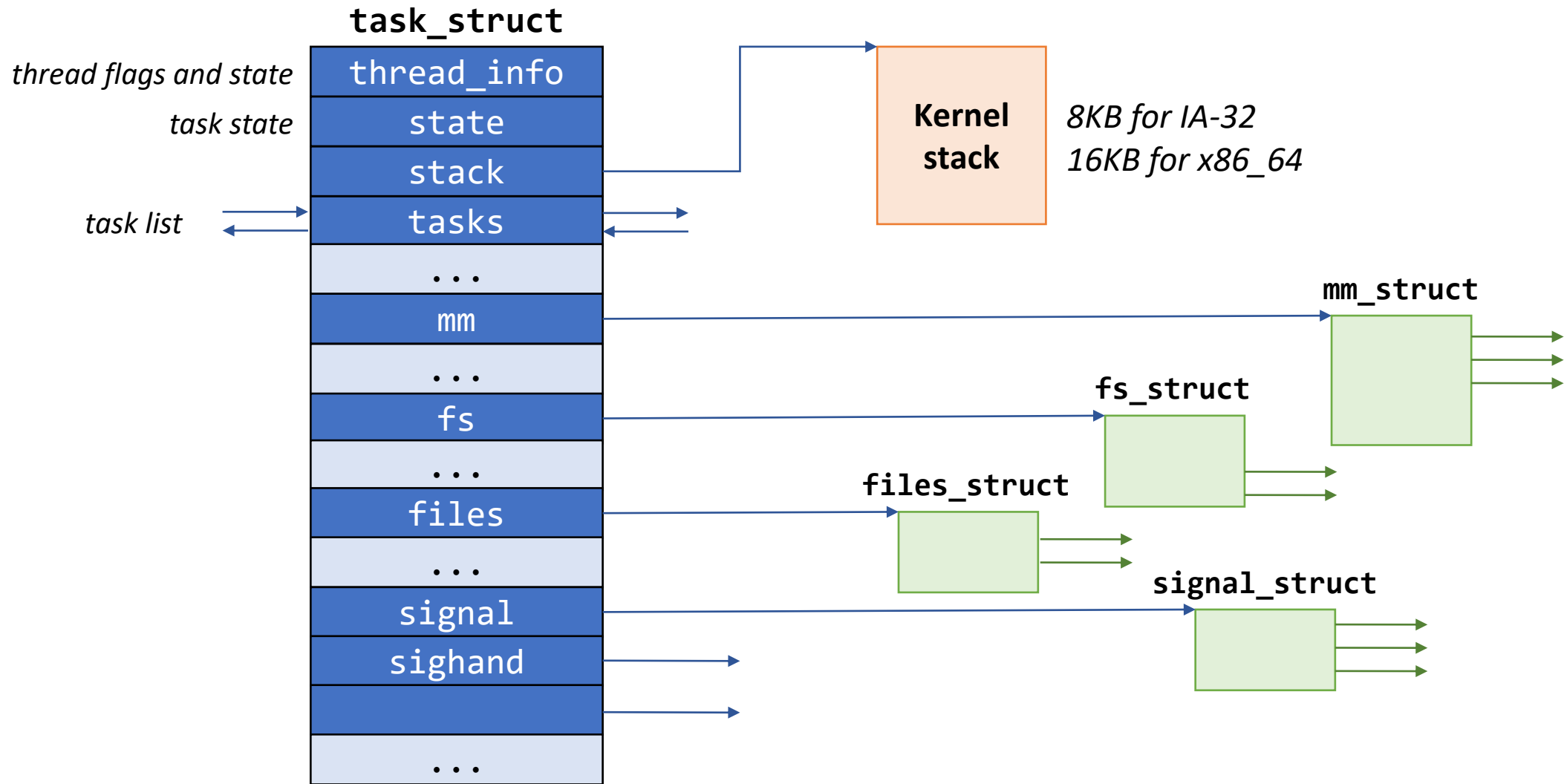
```
struct task_struct {
#ifdef CONFIG_THREAD_INFO_IN_TASK
/*
 * For reasons of header soup (see current_thread_info()), this
 * must be the first element of task_struct.
 */
struct thread_info  thread_info;
#endif
/* -1 unrunnable, 0 runnable, >0 stopped: */
volatile long      state;

/*
 * This begins the randomizable portion of task_struct. Only
 * scheduling-critical items should be added above here.
 */
randomized_struct_fields_start

void                *stack;
refcount_t          usage;
/* Per task flags (PF_*), defined further below: */
unsigned int        flags;
unsigned int        ptrace;

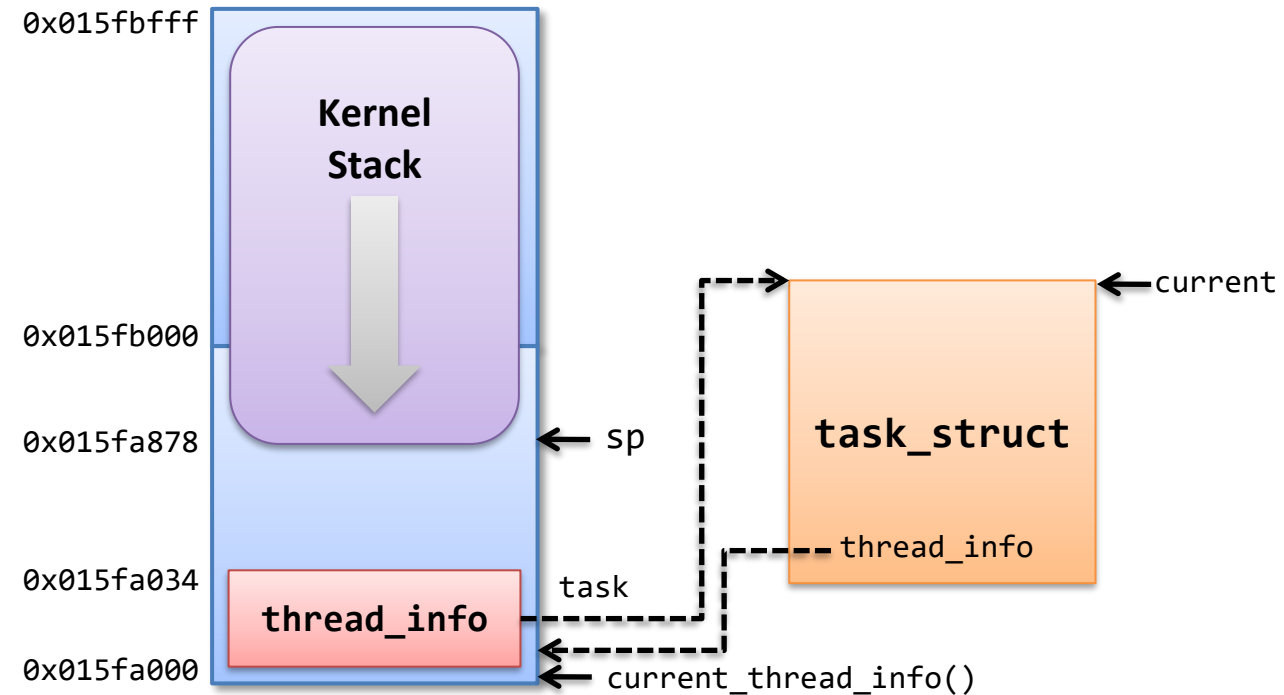
#ifdef CONFIG_SMP
struct llist_node   wake_entry;
int                 on_cpu;
#endif
#ifdef CONFIG_THREAD_INFO_IN_TASK
/* Current CPU: */
unsigned int        cpu;
#endif
}
```

Task Structure



Finding the Current Task

- `get_current()`
 - Per-cpu variable called `current_task` is maintained
- Old way
 - When `CONFIG_THREAD_INFO_IN_TASK=n`
 - Put the `thread_info` at the top of the kernel stack
 - Get current `thread_info` from the stack pointer
 - `thread_info` has a pointer to the `task_struct`



Execution Contexts

■ Process context

- Process enters kernel space by a system call or an exception
- The kernel is executing on behalf of the process
- The `current` variable is valid

■ Interrupt context

- The system is executing an interrupt handler
- There is no task tied to interrupt handlers
- The `current` variable should not be used (except for the scheduler)

Creating a New Process

- `sys_fork()` → `_do_fork()` (@ kernel/fork.c)
- `copy_process()`
 - Check parameters
 - Invoke `dup_task_struct()` to create a new kernel stack and `task_struct` for the new process
 - Make sure the child will not exceed the resource limit
 - Invoke `sched_fork()` to initialize the scheduler-related data structure
 - Invoke `copy_files()`, `copy_fs()`, `copy_sighand()`, `copy_signal()`, `copy_mm()`, etc. to copy those data structures
 - Invoke `copy_thread_tls()` to initialize user registers of the child
 - Allocate a new PID by calling `alloc_pid()`

Creating a New Process (cont'd)

- `copy_process()` (cont'd)
 - Initialize the fields for parenthood relationship and thread group
 - Invoke `attach_pid()` to insert the child PID to the PID hash table
- `wake_up_new_task()`
 - Invoke `activate_task()` to insert the child into the runqueue
- Returns the PID of the child

Linux Threads

- Linux implements all threads using standard tasks
 - There is no concept of a thread
 - A thread is merely a task that shares certain resources with other tasks
- One-to-one model
 - Linux creates a task for each application thread using `clone()` system call
- Sharing resources
 - Resources to be shared can be specified in the `flags` argument in `clone()`
 - `CLONE_VM`: parent and child share address space
 - `CLONE_FILES`: parent and child share open files
 - `CLONE_FS`: parent and child share filesystem information
 - `CLONE_SIGHAND, ...` (cf.) `$ man 2 clone`

POSIX Compatibility

- Basic difference in multithreading model
 - **POSIX**: a single process that contains one or more threads
 - **Linux**: separate tasks that may share one or more resources
- Resources
 - **POSIX**: the following resources are specific to a thread, all other resources are global to a process
 - CPU registers, user stack, blocked signal mask
 - **Linux**: the following resources may be shared between tasks via `clone()`, while all other resources are local to each task
 - Address space, signal handlers, open files, working directory, ...
- `getpid()`, `fork()`, `exec()`, `exit()`, signals, suspend/resume, ...?

Thread Group

- A set of threads that act as a whole with regards to some system calls
- The first thread (task) in a process becomes the thread group leader
 - A new thread created with `CLONE_THREAD` is placed in the same thread group as the calling thread
- Handling process-based system calls:
 - `getpid()` returns the PID of the thread group leader (`t->tgid`)
 - On `exec()`, all threads other than the thread group leader are terminated, and the new program is executed in the thread group leader
 - After all of the threads in a thread group terminate, a `SIGCHLD` signal is sent to the parent process
 - Signals may be sent to a thread group as a whole

Kernel Threads

- Standard tasks that exist solely in the kernel space
 - Kernel threads share the kernel's address space
 - They operate only in the kernel space and do not context switch into the user space
 - Kernel threads are, however, schedulable and preemptable as normal tasks
 - Used to perform certain tasks in background (e.g., kswapd)
- Creating a kernel thread
 - `pid_t kernel_thread(int (*fn)(void *), void *arg, unsigned long flags)`
 - Also created by `clone()` with `CLONE_VM`