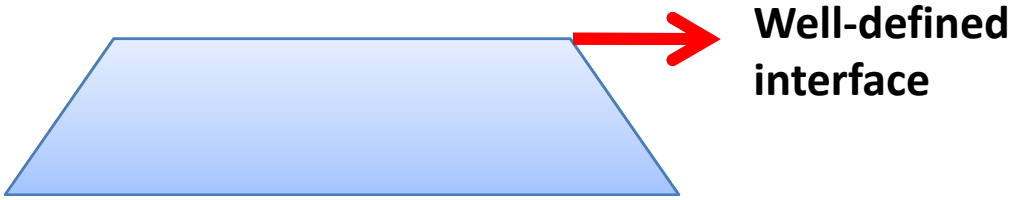
*CS632/SEP564: Embedded Operating Systems (Fall 2008)*

# Introduction to Operating Systems

KAIST

# What is an OS? (1)

- **Application view**
  - Provides an execution environment for running programs
  - Provides an abstract view of the underlying computer system
  - Abstraction: a key concept in computer science to tame the complexity




  - Abstraction is applied recursively → levels of abstraction

# What is an OS? (2)

- **Abstraction by operating systems:**
  - Processors → Processes, Threads
  - Memory → Address spaces (virtual memory)
  - Storage → Volumes, Directories, Files
  - I/O Devices → Files (ioctls)
  - Networks → Files (sockets, pipes, …)
  - …
- **Abstraction vs. virtualization?**

# What is an OS? (3)

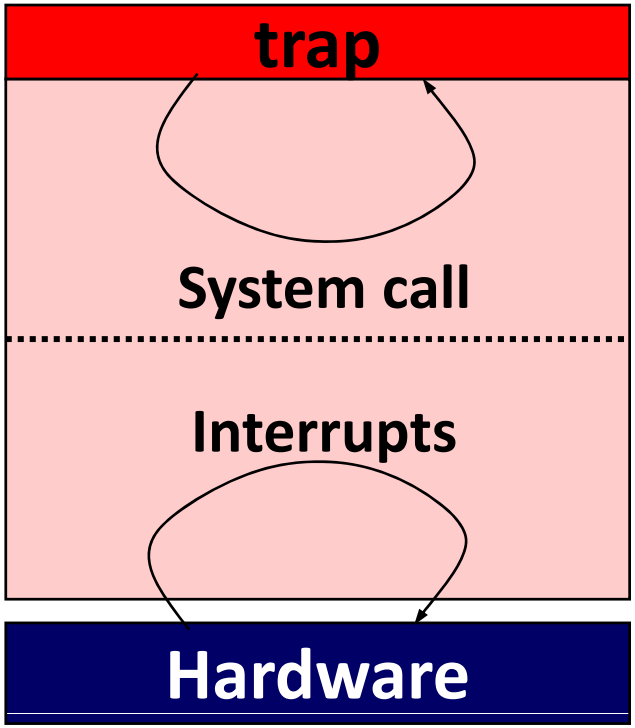
- **System view**
  - Manages various resources of a computer system
  - Sharing
  - Protection
  - Fairness
  - Efficiency
  - …

**Resources**

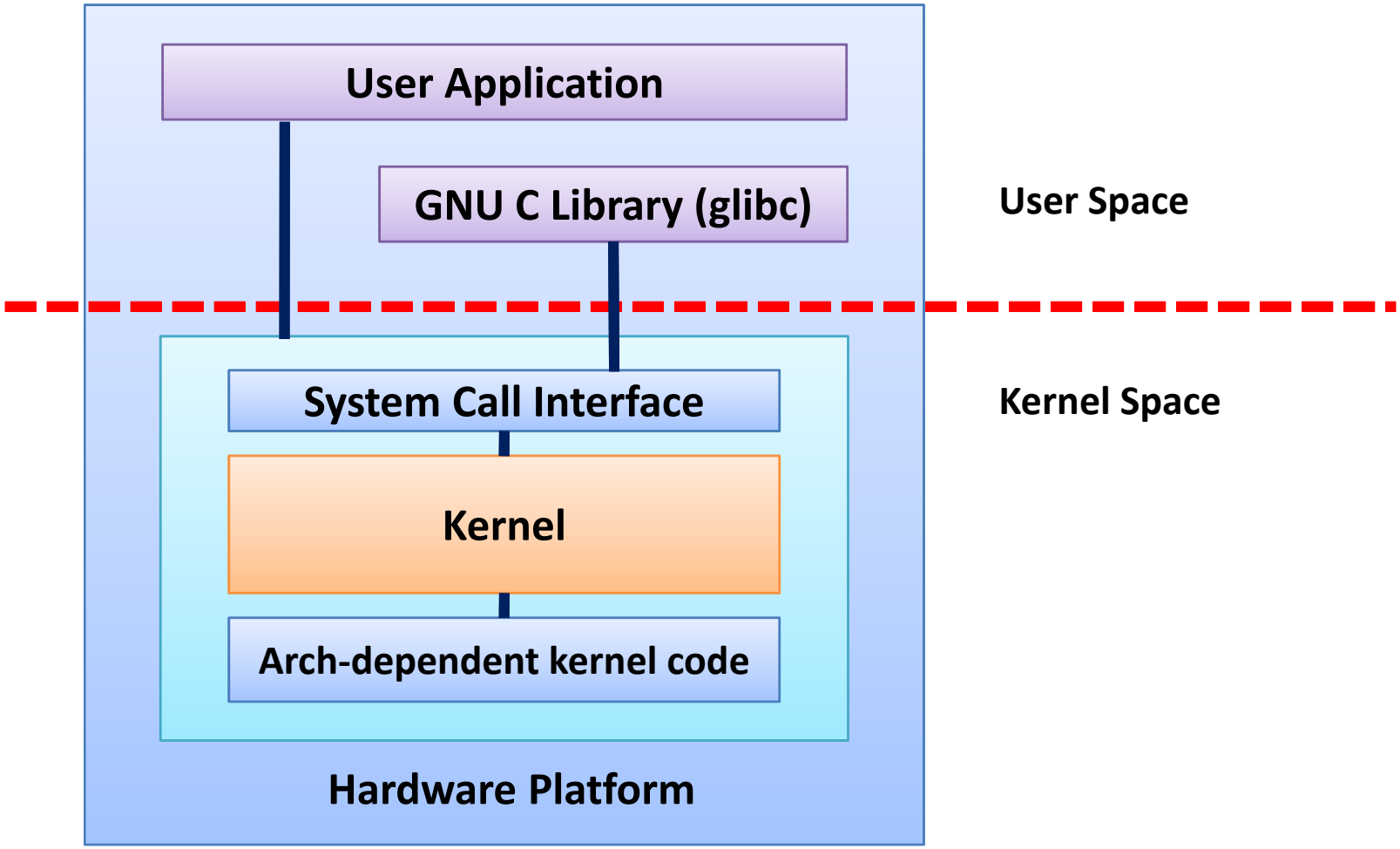
- **CPU**
- **Memory**
- **I/O devices**
- **Queues**
- **Energy**
- **…**

# What is an OS? (4)

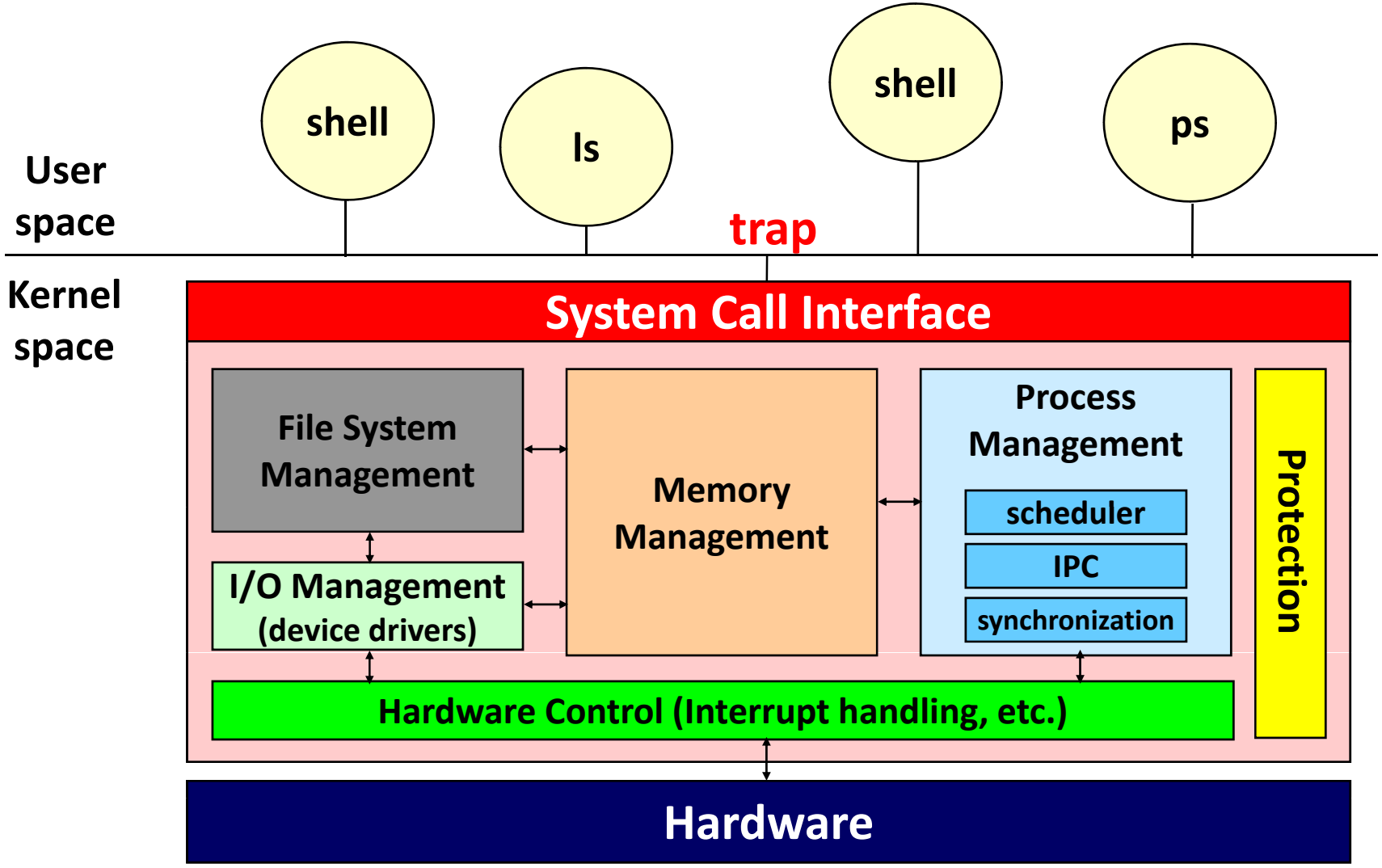
- **Implementation view**
  - The OS is a highly-concurrent, event-driven software



# OS Structure (1)



# OS Structure (2)



*CS632/SEP564: Embedded Operating Systems (Fall 2008)*

# Threads, Processes, and Tasks

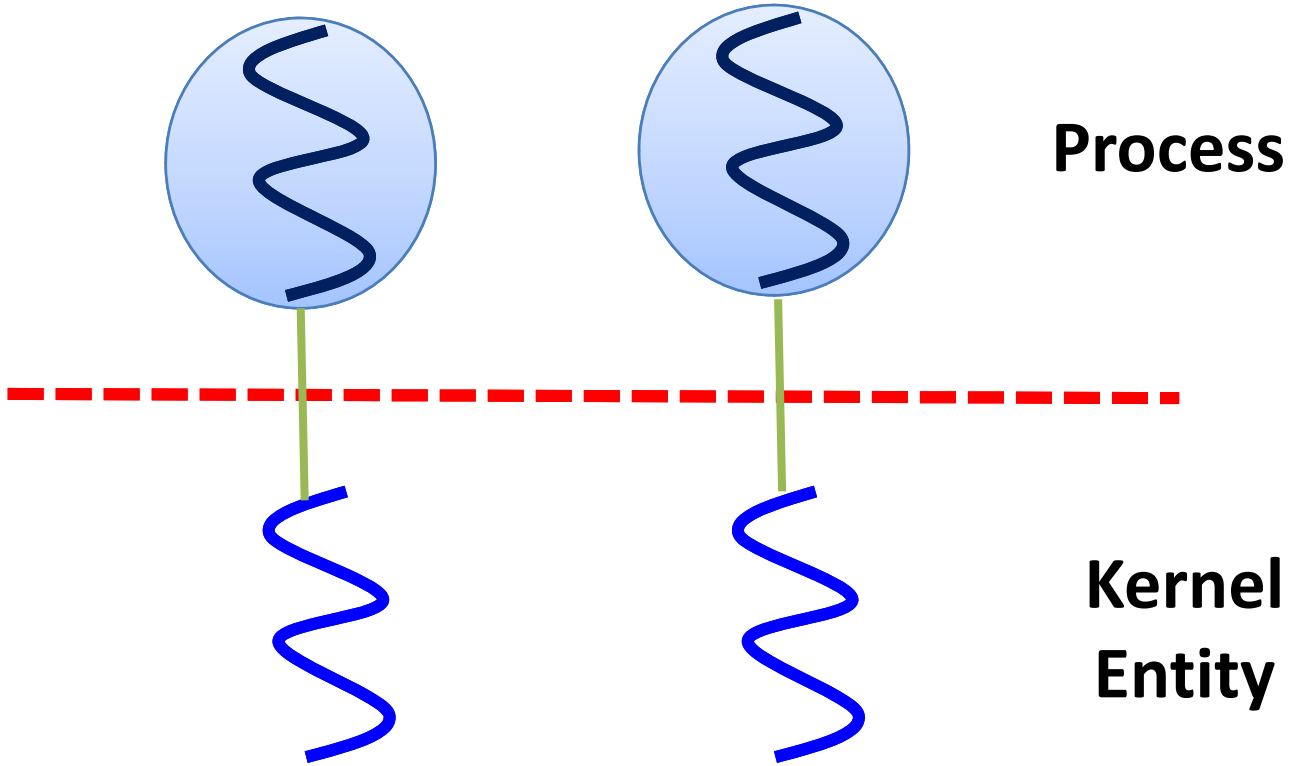
KAIST

# Abstracting CPUs

- **Process**
  - An instance of a program in execution
  - Resource allocation unit
  - fork(), exec(), wait()
- **Thread**
  - A sequential flow of control (for cheap concurrency)
  - Scheduling unit
  - Pthreads (POSIX Threads)
  - To Linux, a thread is a special kind of process
- **Task**
  - Process or thread

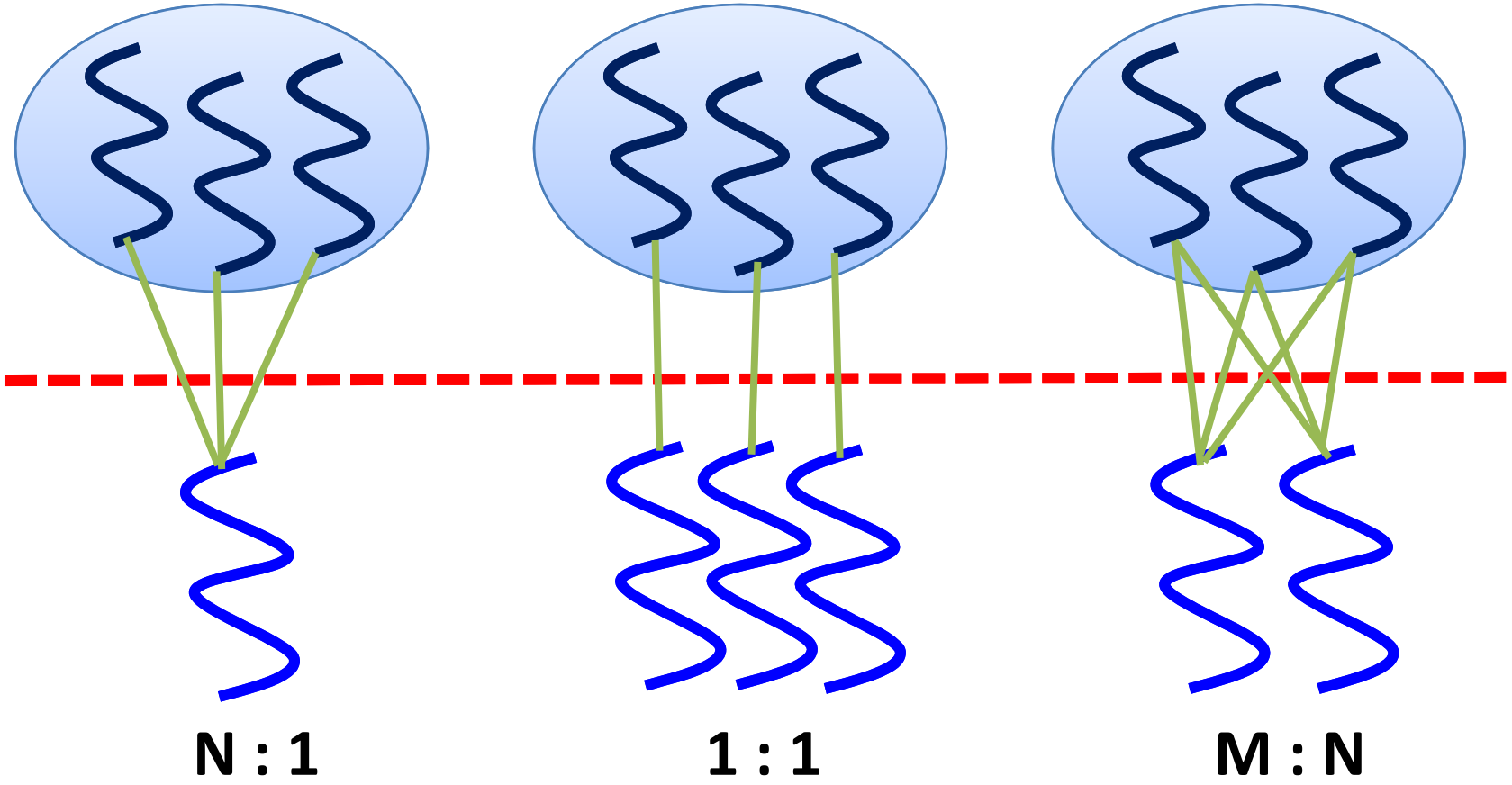
# Implementation (1)

- **Processes**



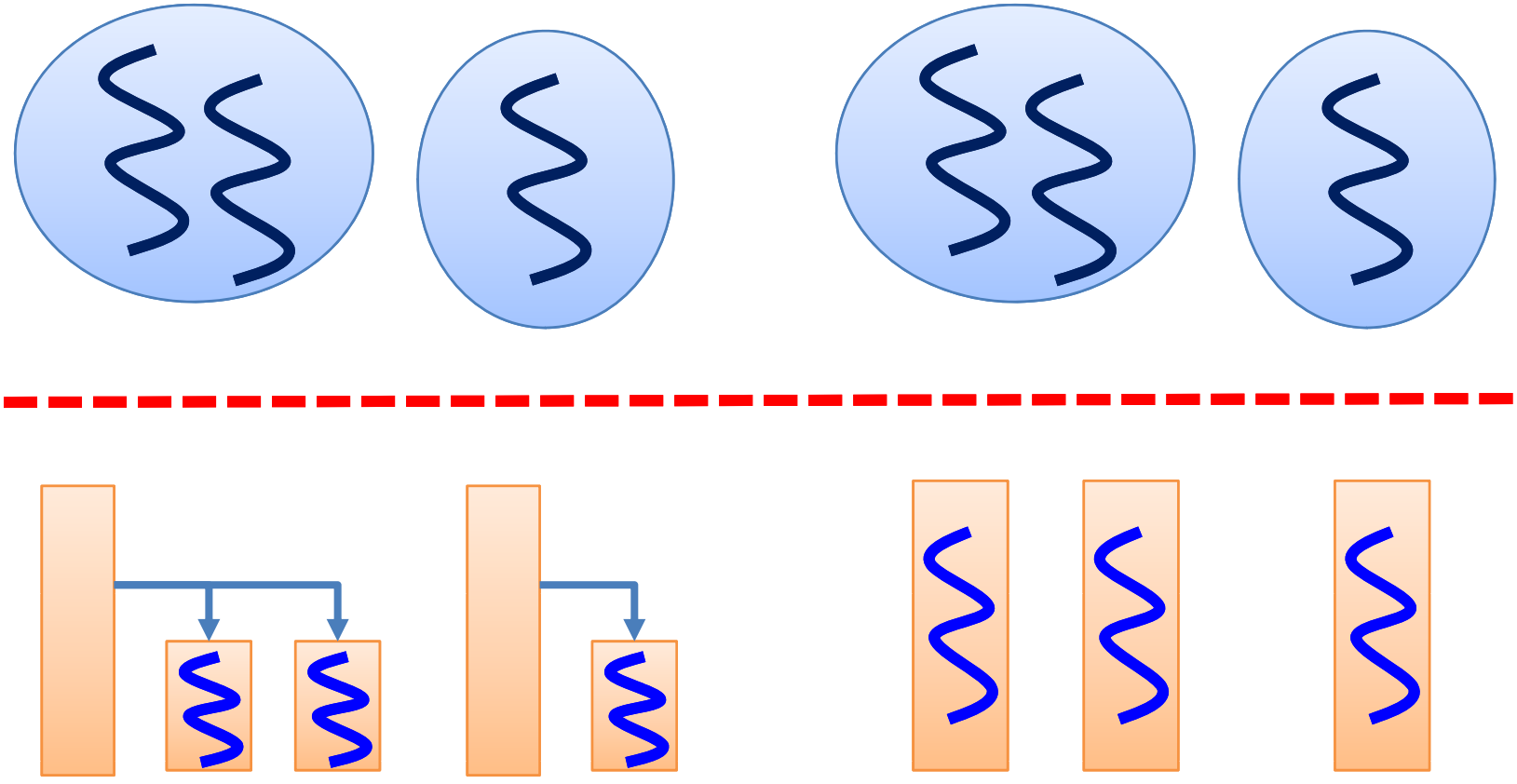
# Implementation (2)

- **Threads**



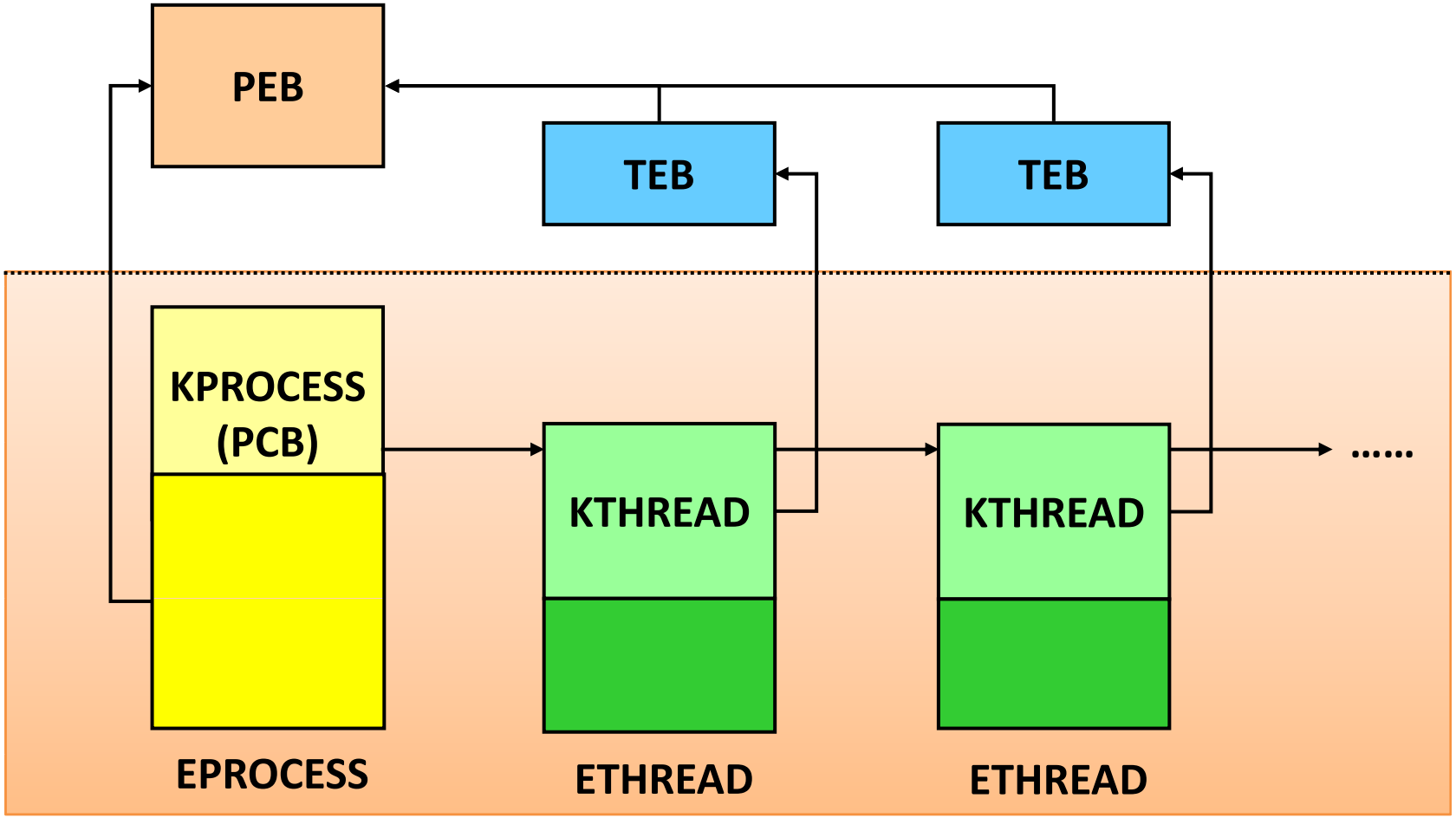
# Implementation (3)

- **Processes with threads (1:1 model)**



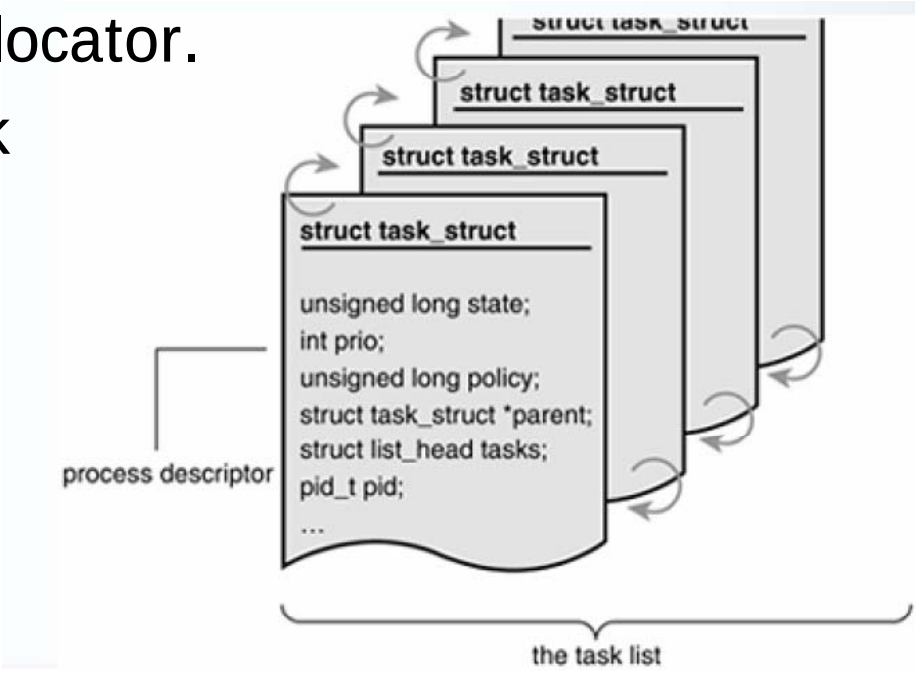
# Implementation (4)

- **Windows XP**

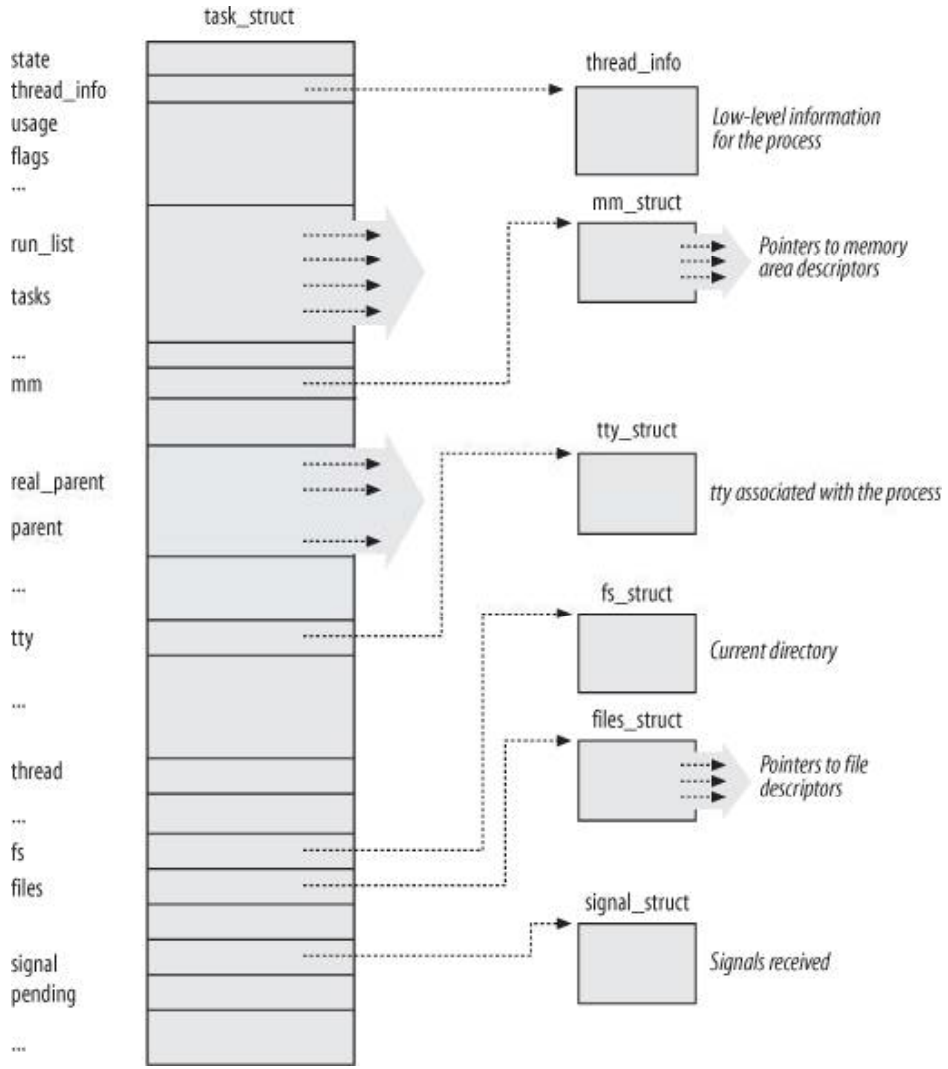


# Process Descriptor (1)

- **struct task_struct** <linux/sched.h>
  - Everything the kernel has to know about a process or a thread.
  - About 1.7KB on a 32-bit machine.
  - Allocated by the slab allocator.
  - Task list: the list of task structures in a circular doubly linked list.

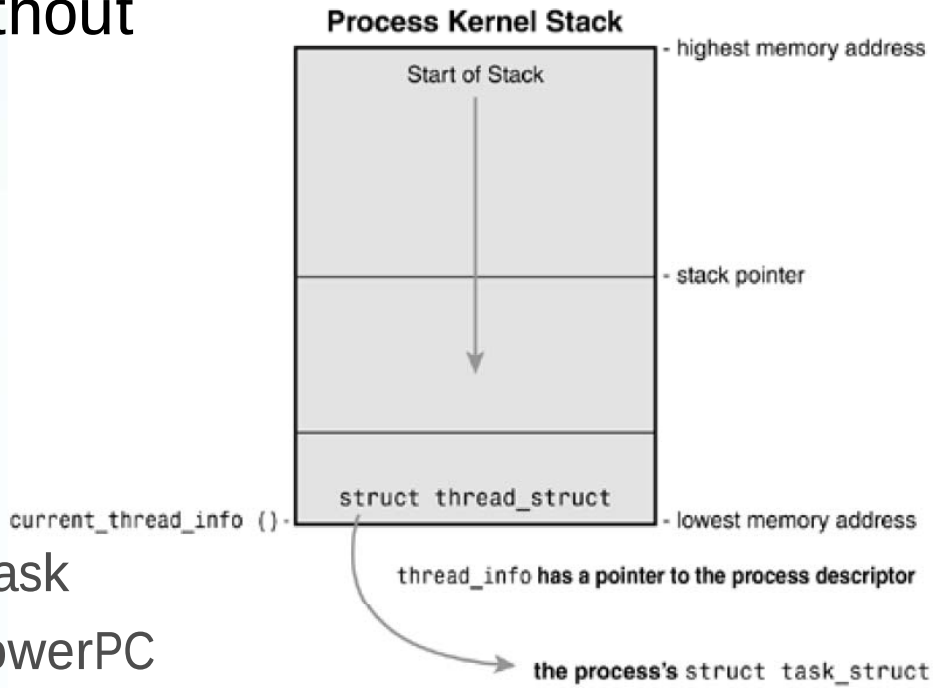


# Process Descriptor (2)

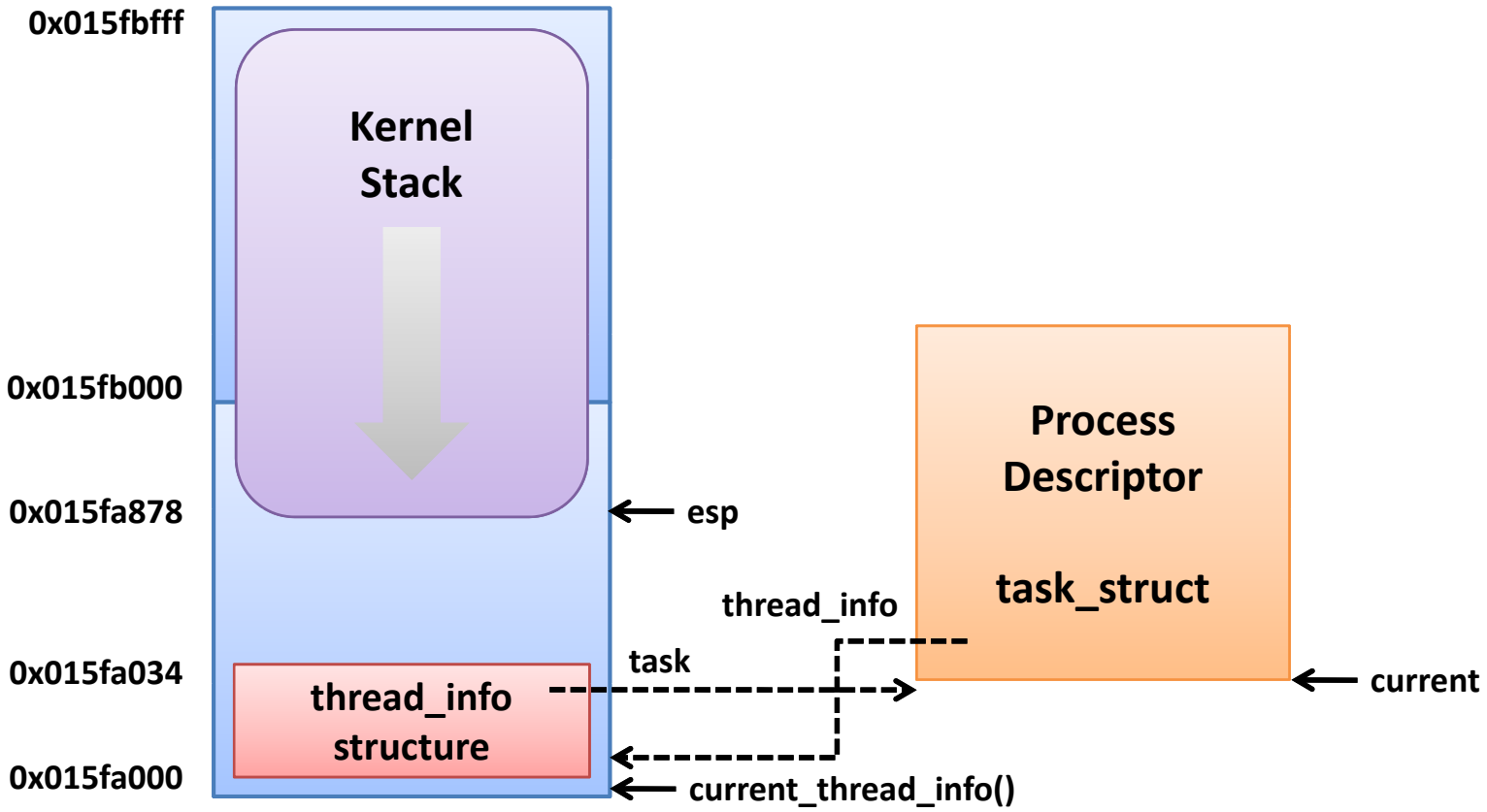


# Process Descriptor (3)

- **struct thread_info** <asm/thread_info.h>
  - Stored at the end of the kernel stack for each process.
    - Usually the kernel stack is 8KB.
  - Easy to calculate the location of the process descriptor via the stack pointer without using an extra register.
  - current_thread_info():
    - %ESP & 0xffffe000
    - Useful especially for multithreaded system
  - current:
    - current_thread_info()→task
    - Register r2 is used for PowerPC



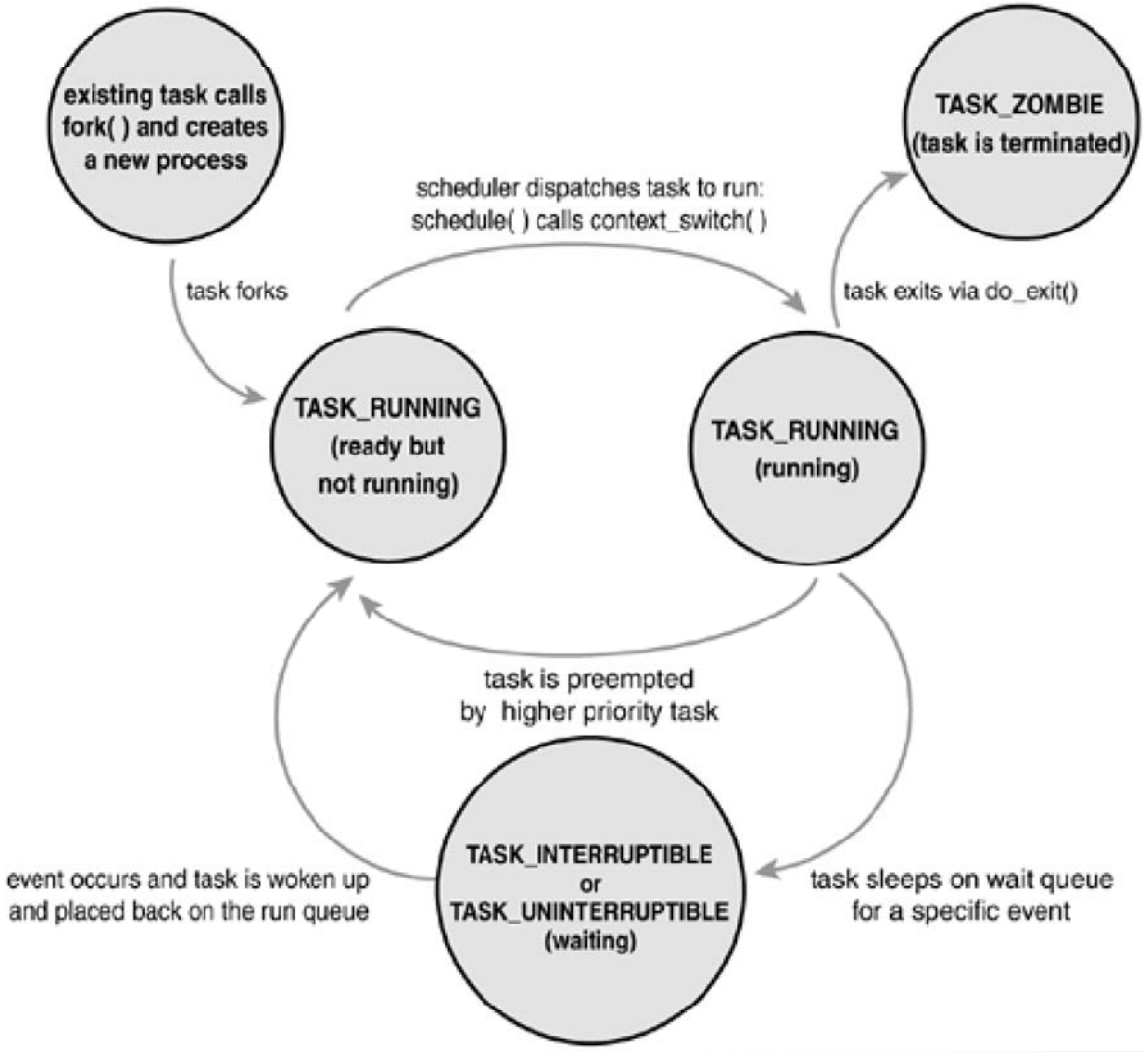
# Process Descriptor (4)



# Process State (1)

| State | Description |
|---|---|
| TASK_RUNNING | Currently running or on a runqueue waiting to run. |
| TASK_INTERRUPTIBLE | Sleeping, waiting for some condition to exist. |
| TASK_UNINTERRUPTIBLE | Identical to TASK_INTERRUPTIBLE except that it doesn't wake up and become runnable if it receives a signal. |
| TASK_ZOMBIE | Terminated, but its parent has not yet issued a wait(). |
| TASK_STOPPED | Process execution has stopped. The task isn't running nor is it eligible to run. |

# Process State (2)



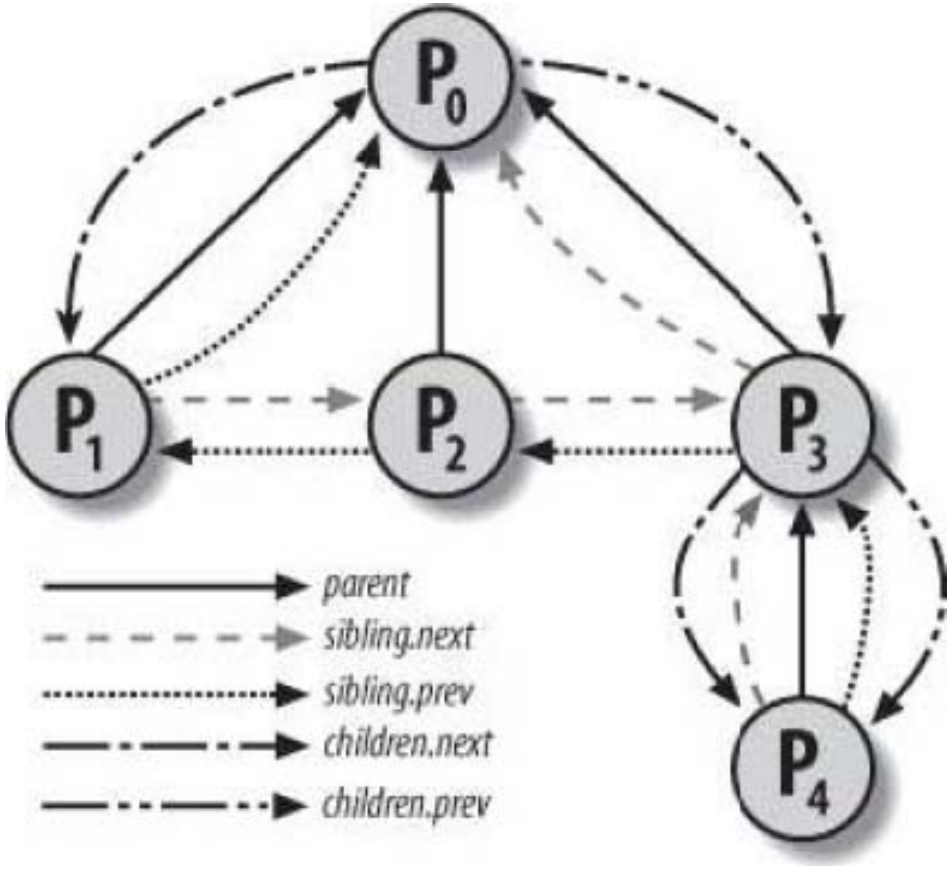
# Process Execution

- **Process context**
  - Process enters kernel space by a system call or an exception.
  - The kernel is “executing on behalf of the process”
  - The **current** macro is valid.

- **Interrupt context**
  - The system is executing an interrupt handler.
  - There is no process tied to interrupt handlers.
  - The **current** macro should not be used.

# Managing Processes (1)

- **Process family tree**



# Managing Processes (2)

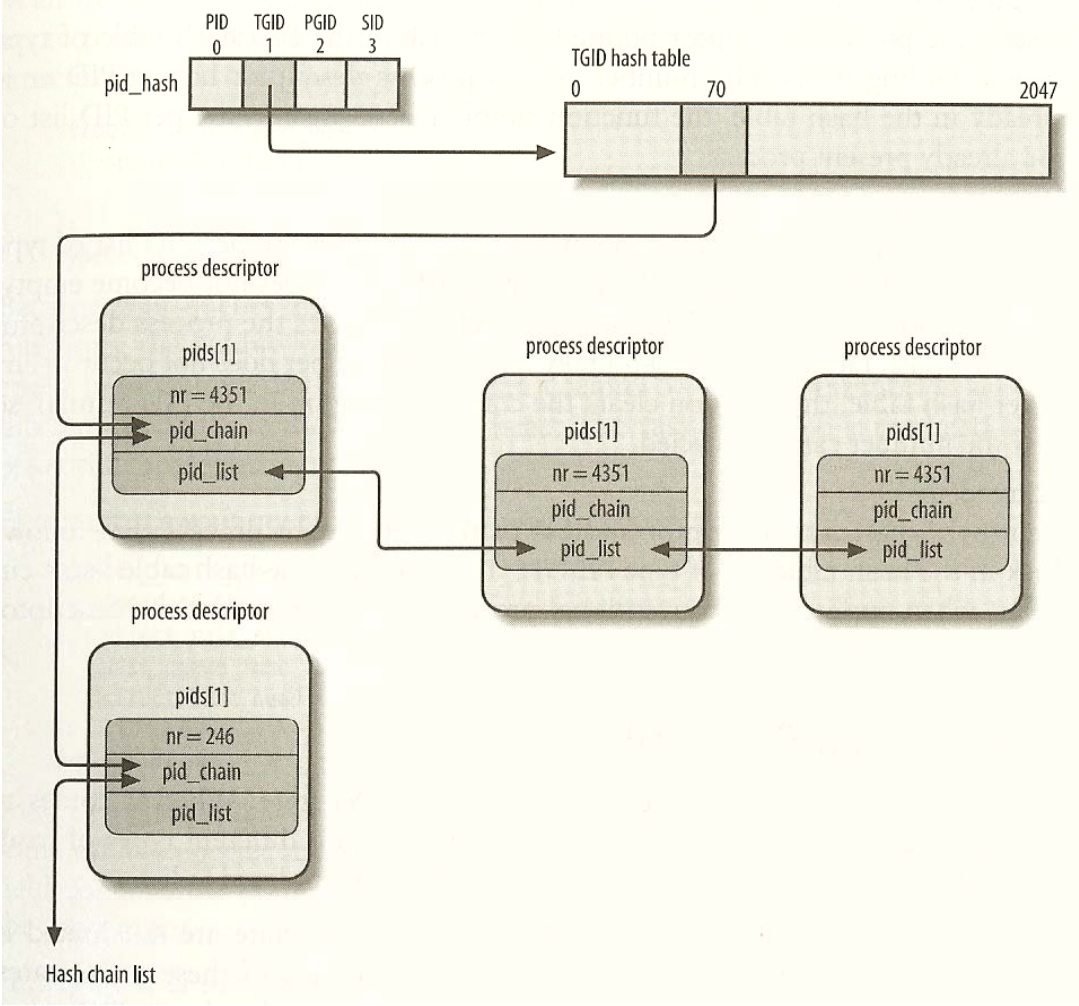
- **PID (Process ID)**
  - PIDs are numbered sequentially.
  - When the kernel reaches an upper limit, it starts recycling the lower, unused PIDs.
    - Maximum PID number: 32,767 (PID_MAX_DEFAULT – 1)
    - Adjustable using /proc/sys/kernel/pid_max
  - pidmap_array
    - A bitmap that denotes which are the PIDs currently assigned and which are the free ones.
    - In 32-bit machines, the pidmap_array bitmap is stored in a single page.

# Managing Processes (3)

- **Finding the process descriptor**
  - PID hash tables
  - 4 different hash tables according to different types
    - PIDTYPE_PID: pid of the process
    - PIDTYPE_TGID: pid of thread group leader process
    - PIDTYPE_PGID: pid of the group leader process
    - PIDTYPE_SID: pid of the session leader process
  - Hash table size
    - 2048 entries for 512MB RAM
  - Hash collision: use chaining
    - Each table entry is the head of a doubly linked list.

# Managing Processes (4)

- **PID hash tables**



# Process Creation (1)

- **Implementing fork()**
  - Implemented via clone() system call
  - do_fork() <kernel/fork.c>
    - Allocate a new PID by looking in the pidmap_array bitmap.
    - Invoke copy_process()
      - » Check parameters
      - » Invoke dup_task_struct() to create a new kernel stack, thread_info structure, and task_struct for the new process.
      - » Make sure the child will not exceed the resource limit.
      - » Invoke copy_semundo(), copy_files(), copy_fs(), copy_sighand(), copy_signal(), copy_mm(), and copy_namespace() to initialize those data structures
      - » Invoke copy_thread() to initialize the kernel stack of the child

# Process Creation (2)

- **Implementing fork() (cont'd)**
  - » Invoke sched_fork() to complete the initialization of the scheduler data structure of the new process
    - » Determine the CPU on which the child will be running
    - » Set the state of the new process to TASK_RUNNING
    - » Share the remaining timeslice of the parent between the parent and the child.
  - » Initialize the fields that specify the parenthood relationships.
  - » Initialize thread group-related fields.
  - » Invoke attach_pid() to insert the child PID to the PID hash table.
  - – Invoke wake_up_new_task()
    - » Adjust the scheduling parameters
    - » Adjust to run the child first, if the child will run on the same CPU as the parent.
  - – Terminate by returning the PID of the child

# Threads (1)

- **Linux threads implementation**
  - There is no concept of a thread.
  - Linux implements all threads as standard processes.
  - A thread is merely a process that shares certain resources with other processes.

    (cf.) Microsoft Windows or Sun Solaris: explicit kernel support for threads
  - Thread is also created by clone() system call.
    - clone (CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND, 0);
    - fork(): clone (SIGCHLD, 0);
    - vfork(): clone (CLONE_VFORK | CLONE_VM | SIGCHLD, 0);

# Threads (2)

- **clone() flags**

| Flag | Meaning |
|---|---|
| CLONE_FILES | Parent and child share open files. |
| CLONE_FS | Parent and child share filesystem information. |
| CLONE_IDLETASK | Set PID to zero (used only by the idle tasks). |
| CLONE_NEWNS | Create a new namespace for the child. |
| CLONE_PARENT | Child is to have same parent as its parent. |
| CLONE_PTRACE | Continue tracing child. |
| CLONE_SETTID | Write the TID back to user-space. |
| CLONE_SETTLS | Create a new TLS for the child. |
| CLONE_SIGHAND | Parent and child share signal handlers and blocked signals. |
| CLONE_SYSVSEM | Parent and child share System V SEM_UNDO semantics. |
| CLONE_THREAD | Parent and child are in the same thread group. |
| CLONE_VFORK | vfork() was used and the parent will sleep until the child wakes it. |
| CLONE_UNTRACED | Do not let the tracing process force CLONE_PTRACE on the child. |
| CLONE_STOP | Start process in the TASK_STOPPED state. |
| CLONE_SETTLS | Create a new TLS (thread-local storage) for the child. |
| CLONE_CHILD_CLEARTID | Clear the TID in the child. |
| CLONE_CHILD_SETTID | Set the TID in the child. |
| CLONE_PARENT_SETTID | Set the TID in the parent. |
| CLONE_VM | Parent and child share address space. |

# Threads (3)

- **Kernel threads**
  - Standard processes that exist solely in kernel space.
  - Kernel threads do not have an address space.
  - They operate only in kernel-space and do not context switch into user-space.
  - Kernel threads are, however, schedulable and preemptable as normal processes.
  - Creating a kernel thread
    - int kernel_thread(int (*fn)(void*), void *arg, unsigned long flags);
    - Also created by clone()
    - CLONE_KERNEL = (CLONE_FS | CLONE_FILES | CLONE_SIGHAND)

# Threads (4)

- **POSIX compatibility problems**
  - 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, while 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
    - PID, PPID, UID, GID, pending signal mask, ...???

# Threads (5)

- **POSIX threads semantics**
  - Fork
    - Duplicate only a calling thread
  - Signals
    - All signals sent to a process will be collected into a process wide set of pending signals, then delivered to any thread that is not blocking that signal.
  - Exit
    - All threads are killed and the process exits with a status indicating the cause of the death.
  - Exec
    - Terminate all threads, throw away the address space, and instantiate a new address space with a single thread.
  - Suspend/Resume
    - Take effect on all the threads in a process.

# Threads (6)

- **POSIX compatibility**
  - A single process contains one or more threads.
  - The following resources are specific to a thread, while all other resources are global to a process.
    - CPU registers, user stack, blocked signal mask
- **Approaches to POSIX compliance**
  - NGPT (Next Generation POSIX Threading): IBM
    - M:N model
    - Extends GNU Pth library (M:1) by using multiple Linux tasks
  - NPTL (Native POSIX Threading Library): RedHat
    - 1:1 model
    - Adopted for Linux kernel 2.6

# Threads (7)

- **Thread group**
  - A set of threads that act as a whole with regards to some system calls such as getpid(), kill(), and _exit().
  - The POSIX 1003.1c standard states that all threads of a multithreaded application must have the same PID.
  - task_struct *t→tgid:
    - The PID of the thread group leader
      = the PID of the first lightweight process in the group
  - getpid() returns the value of tgid.
    - For thread group leaders, tgid = pid.