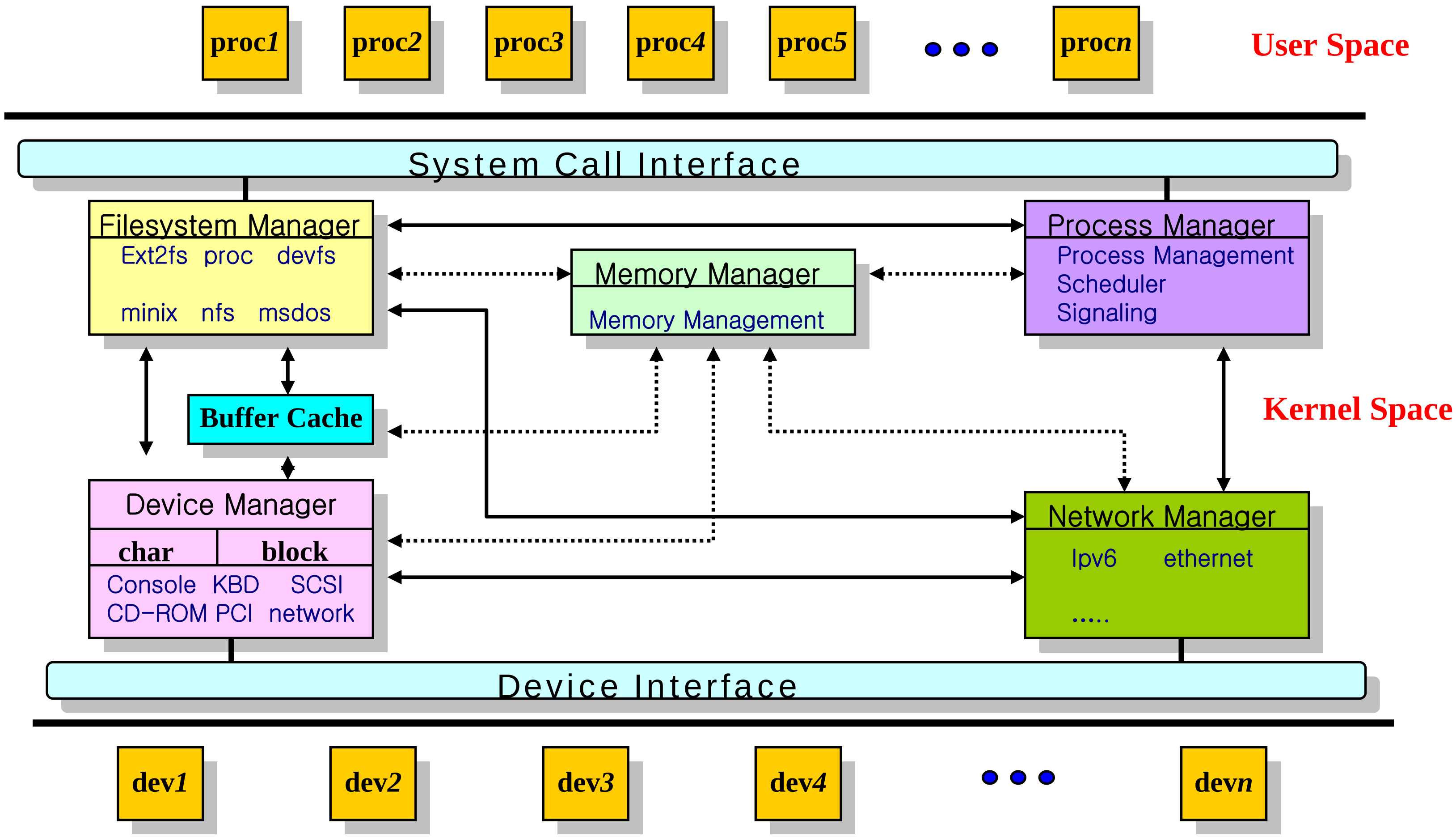
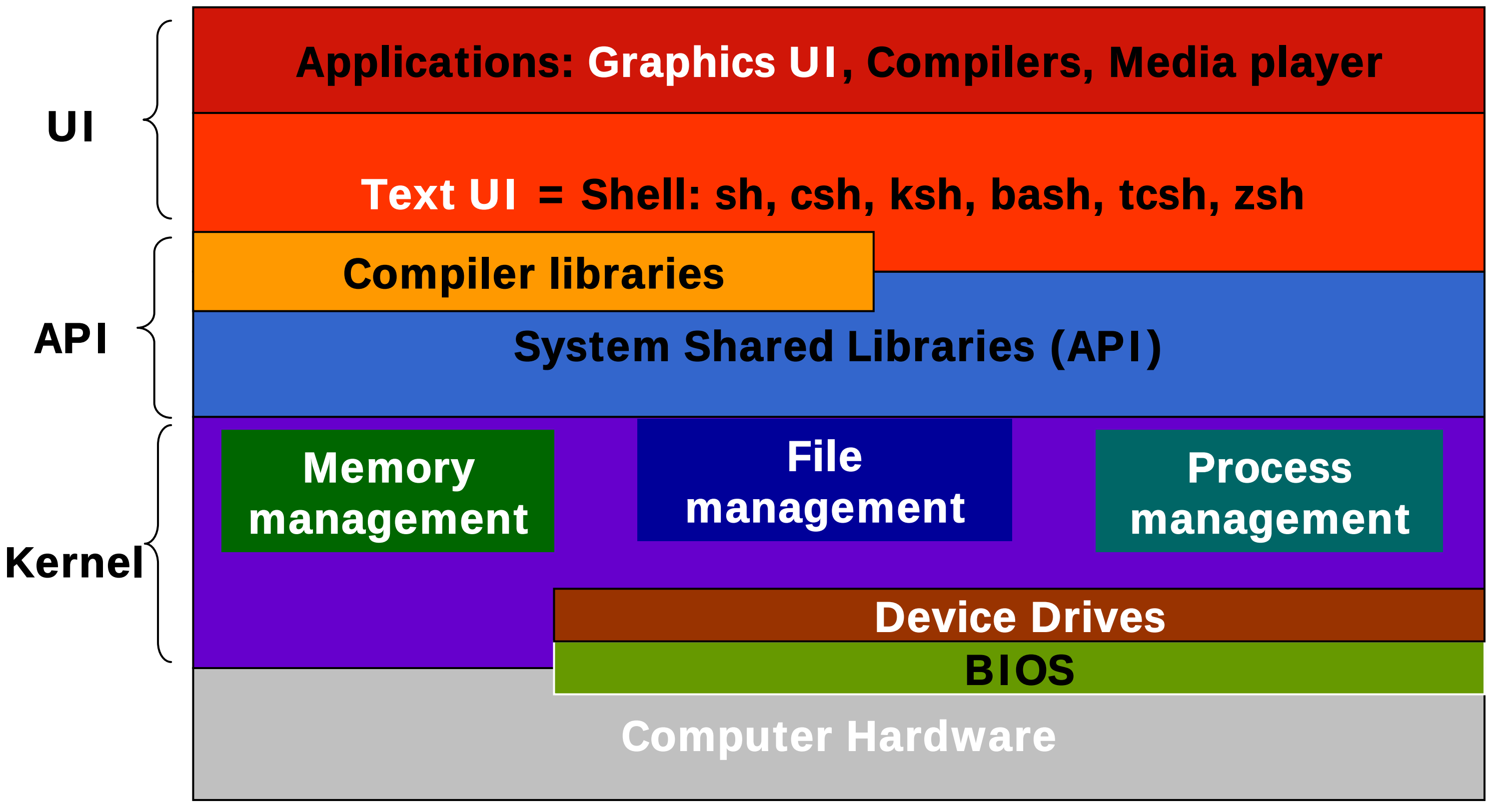
Embedded Systems

Prof. Myung-Eui Lee (F-102)
melee@kut.ac.kr
한국기술교육대학교
korea University of Technology and Education

- **Linux Structure**



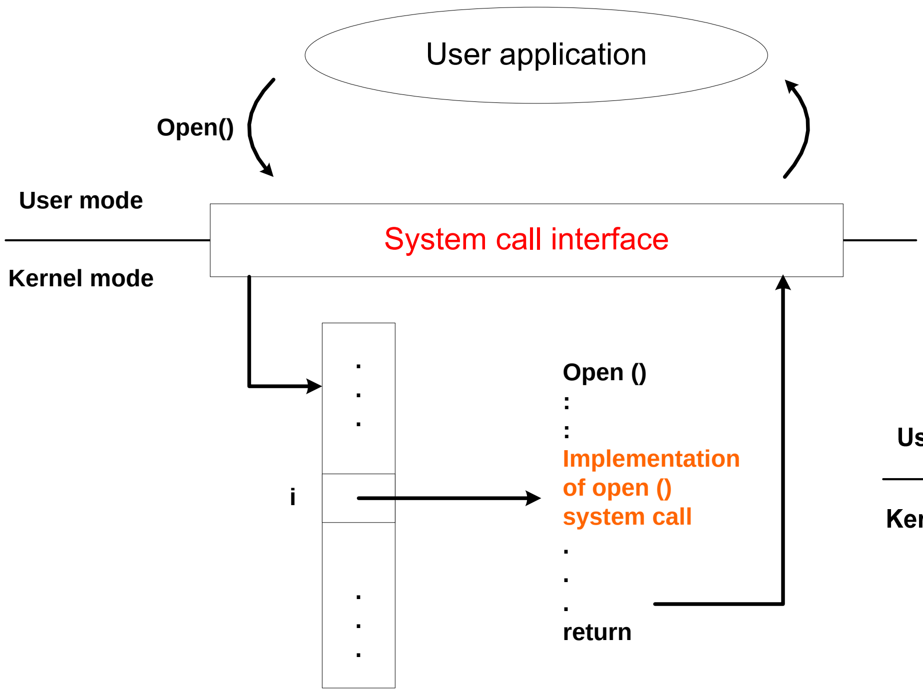
- **Linux Structure**



System Call Interface

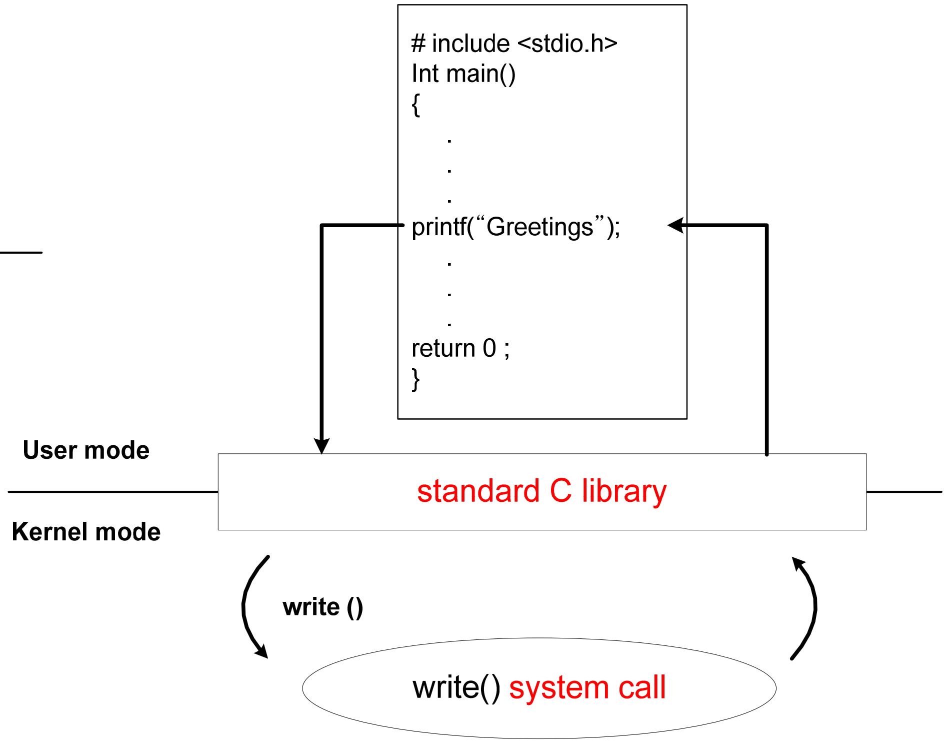
System Call

API – System call – OS Relationship



Standard C Lib.

C program invoking printf() library call, which calls write() system call



System Call

- **System calls define the programmer interface to Linux system**
  - » **Interface between user-level processes and hardware devices.**
    - CPU, memory, disks etc.
  - » **Make programming easier:**
    - Let kernel take care of hardware-specific issues.
  - » **Increase system security:**
    - Let kernel check requested service via system call.
  - » **Provide portability:**
    - Maintain interface but change functional implementation
- **Roughly five categories of system calls in Linux**
  - » **Process control**
  - » **File management**
  - » **Device management**
  - » **Information maintenance**
  - » **Communications**

System Call

- **Invoked by executing int or swi instruction.**
  - » **CPU switches to kernel mode & executes a kernel function.**

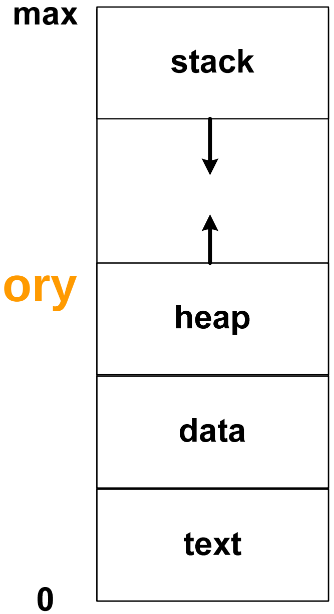
- **Linux files relating to system call:**
  - » **arch/i386/kernel/entry.S**
    - System call and low-level fault handling routines.
    - ENTRY(sys_call_table)
  - » **include/asm-i386/unistd.h**
    - System call numbers and macros.
  - » **kernel/sys.c**
    - System call service routines.

Process

- **Program**
  - » Structured set of commands stored in an executable file on a file system
  - » Executed to create a process
- **Process**
  - » Program running in memory and on the CPU (active program)
  - » Modern systems allow ‘multiprogramming’ (i.e., several processes exist concurrently)
  - » Each process requires an allocation of cpu time, in order to make forward progress
  - » The OS must do ‘scheduling’ of cpu time
  - » Each process also needs the use of other system facilities (memory, files, devices)
  - » the terms *job* (batch system) and *process* is almost used interchangeably

Process

- **A process includes:**
  - » text section : program
  - » stack
  - » data section
  - » heap
- **User process**
  - » Process begun by a user that runs on a terminal (tty)
- **Daemon process**
  - » System process
  - » Not associated with a terminal
- **Process ID (PID) : top**
  - » Unique identifier assigned to every process as it begins
  - » Processes are identified by their process identifier, an integer



process in memory

Process

- **Child processes**
  - » **Refers to a process that was started by another process (parent process)**
- **Parent processes**
  - » **Process that has started other processes (child processes)**
- **Parent Process ID (PPID)**
  - » **The PID of the parent process that created the current process**
- **Processes communicate via pipes; queues of bytes between two processes that are accessed by a file descriptor (IPC)**

Process

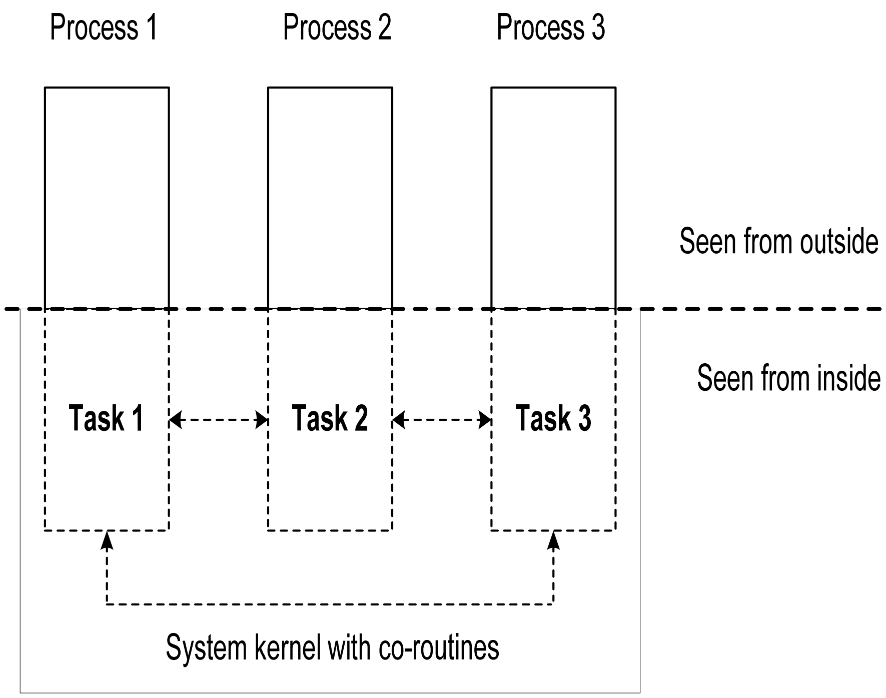
- **Process System Call**
  - » **Creation :** fork **system call creates new process**
    - the process that calls fork() is parent, the new process is child
    - fork() is actually implemented via the clone() system call
    - "cloning flags" : man clone or include/linux/sched.h
  - » **Execution :** exec **system call used after a** fork **to replace the process' memory space with a new program**
  - » **exit :** `exit` **terminates process and free all resources**
  - » **wait : A parent may** `wait` **for a child process to terminate**
- **Task**
  - » **Another name for a process is a task.**
  - » **Generally refer to a process from the kernel's point of view.**
  - » **Linux kernel internally refers to processes as tasks.**

Task

» **As seen by a process running under Linux**
- the kernel is a provider of services.
- Individual processes exist independently alongside each other and cannot affect each other directly.
- Each process's own area of memory is protected against modification by other processes

» **The internal viewpoint of a running Linux**
- Only one program (OS) is running on the computer, can access all the resource.
- The various tasks are carried out by co-routines
- an error in the kernel programming can block the entire system.

Process and Tasks



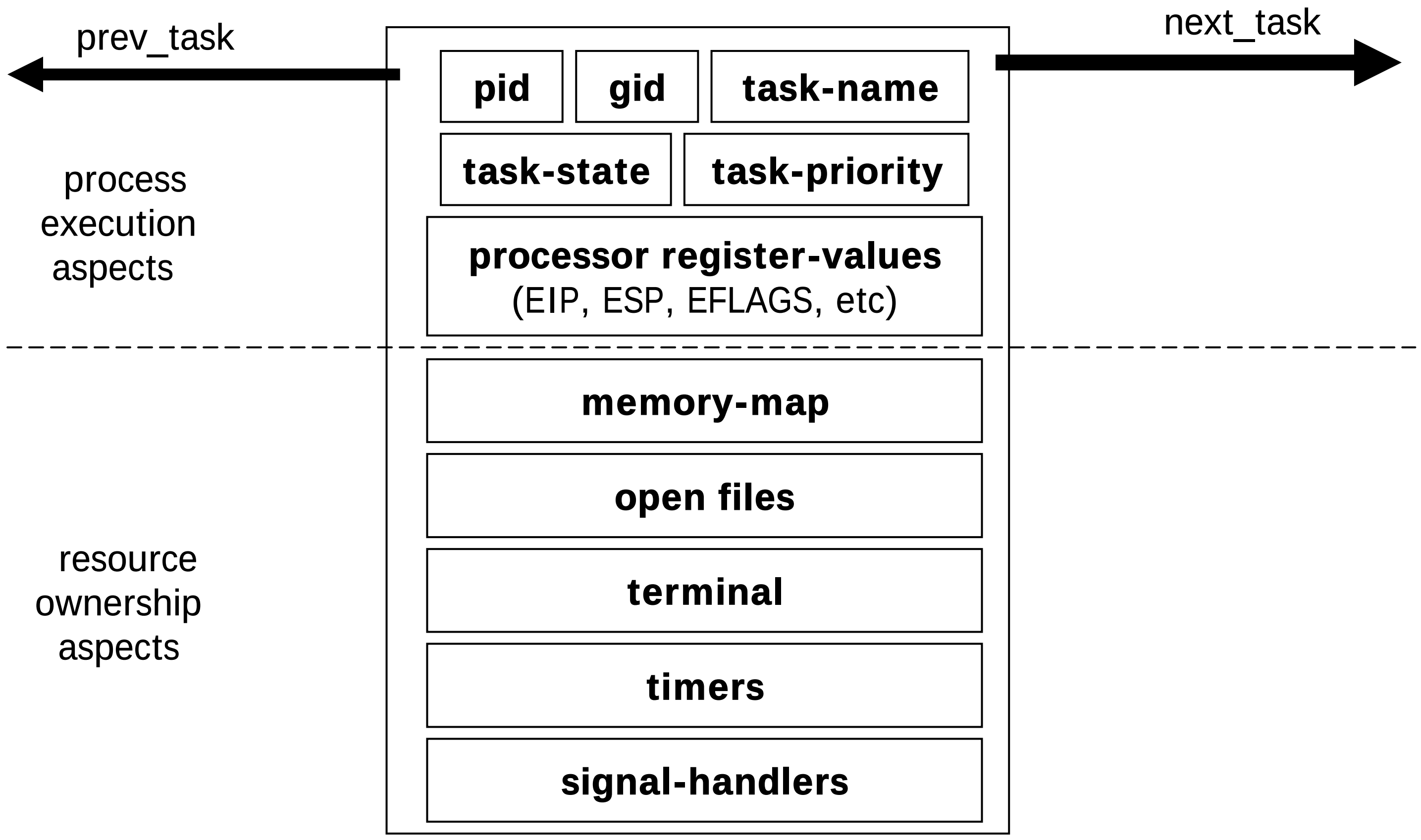
Process Control Block (PCB)

- **Information associated with each process**
- **type sturcture `task_struct`** `; line 382`
  - » **/usr/src/linux-2.4/include/linux/sched.h**

Process state
Unique process identifier
Memory-management information
File system information
Signal information
Program counter
CPU registers
CPU scheduling information (e.g., priority)
Accounting information
I/O status information
Pointers to other control blocks

Process Control Block (PCB)

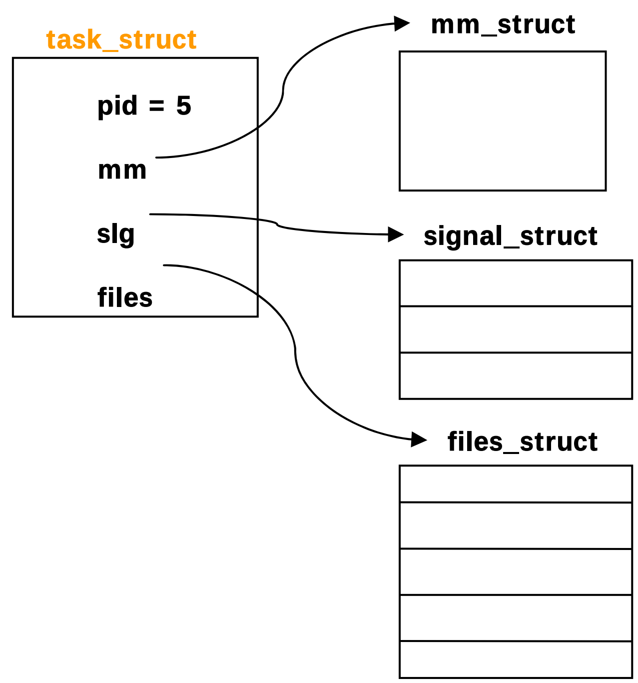
- **task_struct**



Process Control Block

- ## task_struct

```
struct task_struct {
  volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
  unsigned long flags; /* per process flags */
  mm_segment_t addr_limit; /* thread address space:
                        0-0xBFFFFFFF for user-thead
                        0-0xFFFFFFFF for kernel-thread */
  struct exec_domain *exec_domain;
  long need_resched;
  long counter;
  long priority;
  /* SMP and runqueue state */
     struct task_struct *next_task, *prev_task;
     struct task_struct *next_run,  *prev_run;
     ...
  /* task state */
  /* limits */
  /* file system info */
  /* ipc stuff */
  /* tss for this task */
    ...
  /* open file information */
  /* memory management info */
  /* signal handlers */
     ...
};
```

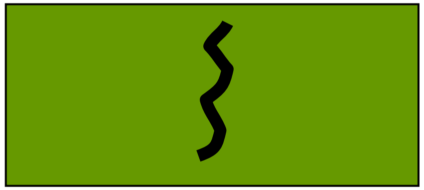


Thread

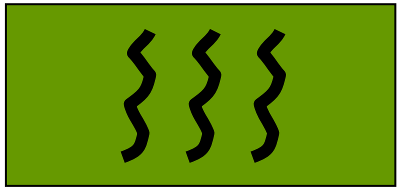
- Thread of execution = Thread
- The object of activity within the process.
- Each thread includes a unique program counter, process stack, and set of processor registers.
- The kernel schedules individual threads, not process.
- Linux has a unique implementation of threads, and does not differentiate between threads and processes : a thread is just a special kind of process.
- A process has two distinct aspects
  - » Its 'execution' (forward progression of states)
  - » Its 'ownership' (share of system's resources)
  - » The word 'thread' refers to the execution aspect of a process (i.e., the entity which gets 'scheduled' by an Operating System for a share of the available cpu time)

Thread

- **multi-threading**
  - » It is possible for an operating system to support a concept of 'process' in which more than one execution-thread exists (with process-resources being shared)



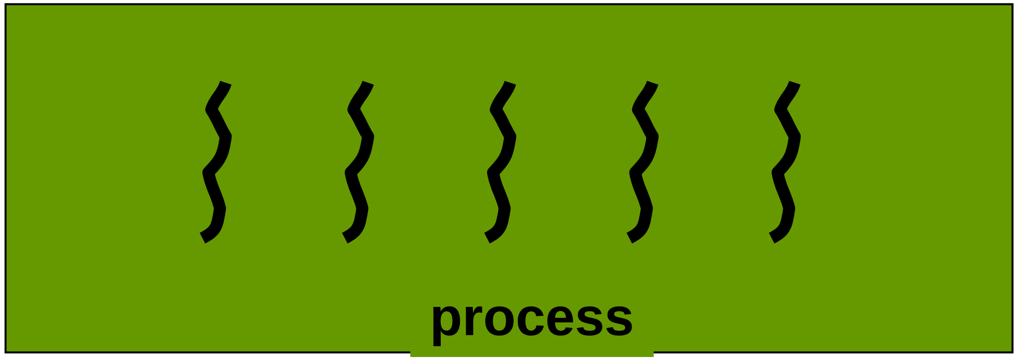
one process with
one thread



one process with
multiple threads

- **Advantages of 'threads'**
  - » All the 'threads' that belong to a process can access the same memory, the same files, the same terminal, the same timers and the same signal-handlers
  - » Thus the system 'overhead' involved in managing a multithreaded task is more efficient (less time-consuming to set up and keep track of) than if each of those threads had individualized ownerships

- **Thread communication**
  - » **Since the threads in a process share the same memory, no special mechanism is needed for them to communicate data**

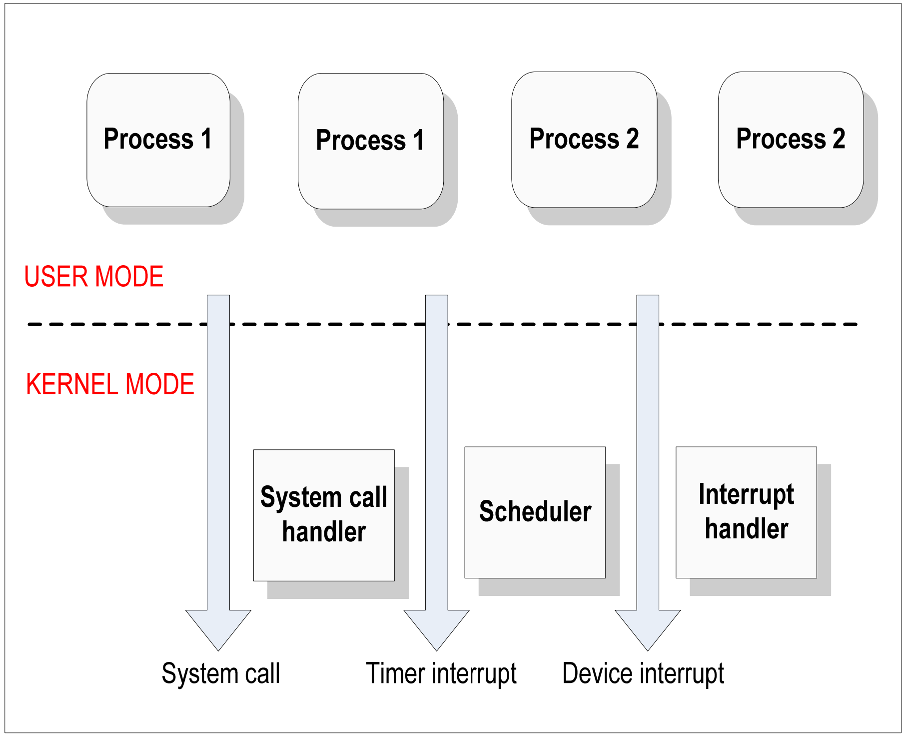


All these threads can read or write the same memory-locations

Process

- » 1. P1 in User Mode issues a system call
- » 2. the process switches to Kernel Mode, and the system call is serviced.
- » 3. P1 then resumes execution in User Mode until a timer interrupt occurs.
- » 4. the scheduler is activated in Kernel Mode.
- » 5. P2 starts its execution in User Mode until a hardware device raises an interrupt.
- » 6. As a consequence of the interrupt, P2 switches to Kernel Mode and services the interrupt.

Transitions between User and Kernel Mode



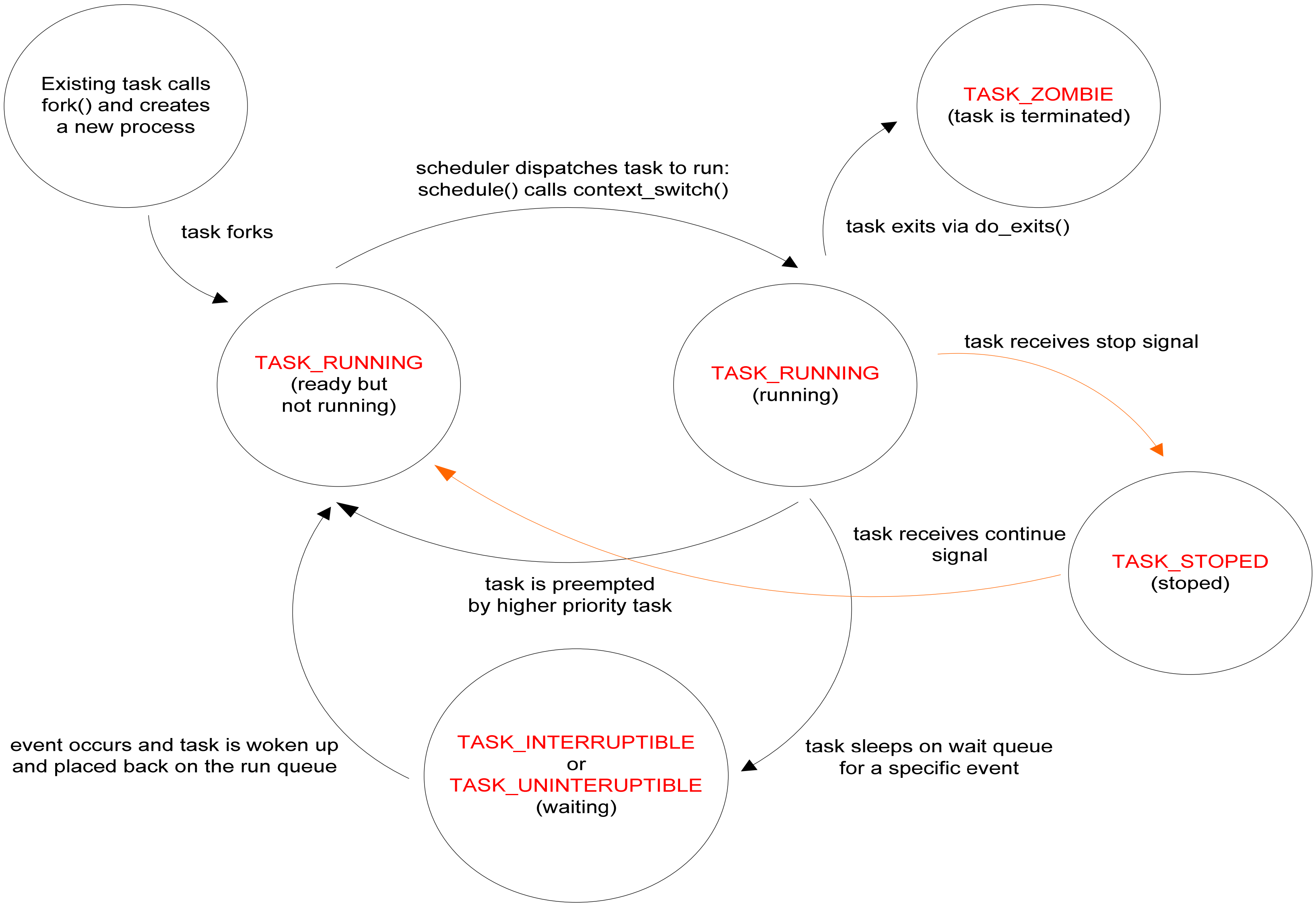
Process State

- TASK_RUNNING : 1. running / 2. ready
  - » The process is runnable
    - 1. currently running - running
    - 2. on a runqueue waiting to run - ready
- TASK_INTERRUPTIBLE : waiting/sleeping/blocked
  - » waiting on a condition: interrupts, or signals
  - » becomes runnable if it receives a signal
- TASK_UNINTERRUPTIBLE
  - » Waiting process cannot be woken by a signal
  - » cannot become runnable if it receives a signal
  - » this is used in situations where the process must wait without interruption.
- TASK_STOPPED : terminated (exit)
  - » stopped process - e.g., by a debugger
  - » this occurs if the process receives SIGSTOP signal

Process State

- TASK_ZOMBIE : defunct
  - » the process has terminated, but its parent has not yet issued a wait system call
  - » When process terminates, still consumes system resources.
  - » Process finished, but parent has not released PID.
  - » Defunct process.
  - » A parent may `wait` for a child process to terminate
  - » `wait` provides the process id of a terminated child so that the parent can tell which child terminated
  - » `wait3` allows the parent to collect performance statistics about the child
  - » Eliminated by killing the parent process

Diagram of Process State



Process

- **Context Switch**
  - » When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process
  - » Context-switch time is overhead; the system does no useful work while switching
  - » Time dependent on hardware support

- **Remote Procedure Call**
  - » Remote procedure call (RPC) abstracts procedure calls between processes on networked systems.

Process

- **Viewing Processes**
  - » ***ps –f* command : user process**
    - **User identifier (UID), PID, PPID, start time, CPU utilization**
  - » ***ps –l* command : user process**
    - **More complete information**
    - **Process state can be seen**
    - **R** = running, **S** = sleeping, **D** = uninterruptible sleep,
      T = stopped or being traced, **Z** = zombie
  - » ***top* or *ps –efl* command : system / user process**
    - Process priority (PRI) : priority
      - → Higher value means lower priority
      - → 0 (high priority) to 139 (low priority)
    - Nice value (NI) : time slice
      - → Indirectly represents priority
      - → The priority value may be in the range -20 (800ms) to 19(5ms).
      - → Higher value means lower priority
    - NICE_TO_PRIO(nice), PRIO_TO_NICE(prio) : kernel/sched.c

Interprocess Communication (IPC)

- **Mechanism for processes to communicate and to synchronize their actions**
- **Message system – processes communicate with each other without resorting to shared variables**
- **IPC facility provides two operations:**
  - » send(***message***) **– message size fixed or variable**
  - » receive(***message***)
- **If *P* and *Q* wish to communicate, they need to:**
  - » **establish a *communication link* between them**
  - » **exchange messages via send/receive**
- **Implementation of communication link**
  - » **physical (e.g., shared memory, hardware bus)**
  - » **logical (e.g., logical properties)**

Communications Model

- **Direct Communication**
  - **Processes must name each other explicitly:**
  - send **(*P, message*) – send a message to process P**
  - receive**(*Q, message*) – receive a message from process Q**
- **Indirect Communication**
  - **Messages are directed and received from mailboxes (also referred to as ports)**
  - **Each mailbox has a unique id**
  - **Processes can communicate only if they share a mailbox**
  - **Operations**
    - create a new mailbox
    - send and receive messages through mailbox
    - destroy a mailbox
  - send**(*A, message*) – send a message to mailbox A**
  - receive**(*A, message*) – receive a message from mailbox A**

Synchronization

- **Message passing may be either blocking or non-blocking**
- Blocking **is considered** synchronous
  - » Blocking send **has the sender block until the message is received**
  - » Blocking receive **has the receiver block until a message is available**
- Non-blocking **is considered** asynchronous
  - » Non-blocking **send has the sender send the message and continue**
  - » Non-blocking **receive has the receiver receive a valid message or null**

Signal

- Introduced in UNIX systems to simplify IPC.
- Used by the kernel to notify processes of system events.
- A signal is a short message sent to a process, or group of processes, containing the number identifying the signal.
- Linux supports 31 non-RT signals.
  - » /include/asm-i386/signal.h
- POSIX standard defines a range of values for RT signals

Signal

- **A signal is sent due to occurrence of corresponding event** (kernel/signal.c – line 1110)
  - » **`send_sig_info(int sig, struct siginfo *info, struct task_struct *t);`**
  - » **`sig` is signal number.**
  - » **`info` is either:**
    - 0, if user mode process is signal sender.
    - 1, if kernel is signal sender.
  - » **`kill_proc_info(int sig, struct siginfo *info, pid_t pid);`**

#	Signal Name	Default Action	Comment
1	SIGHUP	Abort	Hangup terminal or process
2	SIGINT	Abort	Keyboard interrupt (usually Ctrl-C)
…			
9	SIGKILL	Abort	Forced process termination
10	SIGUSR1	Abort	Process specific
11	SIGSEGV	Dump	Invalid memory reference
…			

Signal

- **Sending a signal**
  - » **Kernel sends (delivers) a signal to a destination process by updating some state in the context of the destination process.**
- **Kernel sends a signal for one of the following reasons:**
  - » **Generated internally:**
    - Kernel has detected a system event such as divide-by-zero (SIGFPE) or the termination of a child process (SIGCHLD).
  - » **Generated externally:**
    - Another process has invoked the kill system call to explicitly request the kernel to send a signal to the destination process.

Signal

- **Receiving a signal**
  - » **A destination process receives a signal when it is forced by the kernel to react in some way to the delivery of the signal.**
- **Three possible ways to react:**
  - » **Ignore the signal (do nothing)**
  - » **Terminate the process**
  - » **Catch the signal by executing a user-level function called a signal handler.**
    - Similar to a hardware exception handler being called in response to an asynchronous interrupt.

Scheduler

- **Scheduling**
  - » **Non-preemptive scheduling: the process keeps the CPU until the process terminates or it switches to waiting state (simple to implement).**
  - » **Preemptive scheduling: the process can be interrupted and must release the CPU**
- **Scheduling Policies** : /linux/sched.h
  - » **SCHED_NORMAL : non-RT / classic Unix**
    - Interactive : make the system appear fast to user.
    - RT have higher priorities than any non-RT tasks.
  - » **SCHED_FIFO : soft-RT**
    - if no other higher-priority RT process is runnable, the process will continue to use the CPU as long as it wishes.
  - » **SCHED_RR : soft-RT**
    - assign CPU time to all SCHED_RR processes that have the same priority (fixed time slice).
    - interrupted when its time slice has expired.

* Scheduler *schedule()* function : /kernel/sched.c - line 986

Scheduler

- Long-term scheduler **(or job scheduler)**
  - » **selects which processes should be brought into the ready queue**
  - » **invoked very infrequently (seconds, minutes) ⇒ (may be slow)**
- Short-term scheduler **(or CPU scheduler)**
  - » **selects which process should be executed next and allocates CPU**
  - » **invoked very frequently (milliseconds) ⇒ (must be fast)**
- **Processes can be described as either:**
  - » I/O-bound process **– spends more time doing I/O than computations, many short CPU bursts**
  - » CPU-bound process **– spends more time doing computations, few very long CPU bursts**

Linux implicitly favors I/O-bound processes over CPU-bound ones

Module

- **Linux is a monolithic kernel**
  - » trivial modifications require kernel to be recompiled
  - » kernel is increasing in size by adding new features
  - » many modules occupy permanent space in memory though they are used rarely
- **Module: steps toward micro-kernelized Linux**
  - » small and compact kernel
  - » clean kernel
  - » rapid kernel
  - » components-based Linux
- **Easier way to extend Linux kernel functions.**
- **It is then linked into the kernel when the module is installed.**

Module

- **Most modern operating systems implement kernel modules**
  - » **Uses object-oriented approach**
  - » **Each core component is separate**
  - » **Each talks to the others over known interfaces**
  - » **Each is loadable as needed within the kernel**
- **Overall, similar to layers but with more flexible**
- **Modules can be compiled and dynamically linked into kernel address space.**
  - » **Useful for device drivers that need not always be resident until needed.**
    - ◆ Keeps core kernel “footprint” small.

Module versus Application

- **Modules**
  - » **1. runs in kernel space (no main function).**
  - » **2. just registers itself in order to serve future requests, and its initialization function terminates immediately (event-driven program)**
  - » **3. no libraries to link to (linked only to the kernel): the only functions it can call are the ones exported by the kernel.**
    - printk() : kernel library
- **Applications**
  - » **1. runs in user space (main function).**
  - » **2. performs a single task from beginning to end (not all applications are event-driven)**
  - » **3. can call functions it doesn't define: the linking stage resolves external references using the appropriate library of functions (libc).**
    - printf()

Module

- **Every module consist of two basic functions (minimum)**

```
int init_module(void) /*used for all initialization*/
{
...
}
void cleanup_module(void) /*used for exit */
{
...
}
```

- **Loading a module - by issuing the following command:**
  - » ***insmod module.o***

Module

- **Instruction**
  - **insmod : /proc/modules**
    - Insert an module into the kernel.
  - **rmmod**
    - Remove an module from the kernel.
  - **depmod : /lib/modules/2.4.20-8/modules.dep**
    - Determine interdependencies between modules.
  - **ksyms : /proc/ksyms**
    - Display symbols that are exported by the kernel
  - **lsmod**
    - List currently loaded modules.
  - **modinfo**
    - Display module information (.modinfo section in module object file).
  - **modprobe : /etc/modules.conf**
    - Insert module and resolve module dependency (as described by modules.dep). For example, if you must load A before loading B, modprobe will automatically load A when you tell it to load B.

Module

- **modinfo hello.o**
  - » **MODULE_AUTHOR(“MELEE”);**
  - » **MODULE_DESCRIPTION(“Hello Module”);**
  - » **MODULE_LICENCE(“GPL”);**
- **/proc/ksyms : Kernel Symbol Table**
  - » lists all the symbols the kernel exports
  - » **/usr/src/linux/System.map** (core symbols)
  - » **/proc/ksyms** (all symbols)
- **/proc/modules**
  - » holds information about the loaded modules
  - » column : module name / size / use count / dependency
- **/etc/modules.conf**
  - » modprobe command
  - » A module name : eth0, usb-controller
  - » A more generic identifier : e100, ehci-hcd

Module

- **kernel 2.4 : host**

```
/* HELLO MODULE example -Tested on Linux Kernel 2.4 */

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("MELEE");
MODULE_DESCRIPTION("HELLO MODULE");

int init_module(void)
{
        printk("HELLO MODULE is loaded. \n");

        return 0;
}

void cleanup_module(void)
{
        printk("HELLO MODULE is unloaded. \n");
}
```

```
##          Makefile for Linux Kernel Module for 2.4

CC = gcc

KERNEL_PATH = /usr/src/linux-2.4.20-8
CFLAGS = -DMODULE -D__KERNEL__ -I$(KERNEL_PATH)/include

MOD_OBJ = hello

all: $(MOD_OBJ).o

$(MOD_OBJ): $(MOD_OBJ).c
        $(CC) $(CFLAGS) -c $(MOD_OBJ).c

clean:
        rm -f *.o
```

-D__KERNEL__ = #define __KERNEL__
-DMODULE = #define MODULE

Module

kernel 2.6 : target

```c
/* HELLO MODULE example - Tested on Linux Kernel 2.6.x */

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>

MODULE_LICENSE("GPL");


int hello_init(void)
{
		printk(" HELLO MODULE is loaded. \n");

		return 0;
}

void hello_exit(void)
{
		printk(" HELLO MODULE is unloaded. \n");
}

module_init(hello_init);
module_exit(hello_exit);
```

```makefile
##		Makefile for Linux Kernel Module for 2.6

obj-m		:= hello.o

CC	:= /opt/iwmmxt-1.0.0/bin/arm-linux-gcc

KDIR		:= /pxa270/kernel/linux-2.6.11-h270-tku_v1.1

PWD		:= $(shell pwd)

default:
		$(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules

clean:
		rm -rf *.ko
		rm -rf *.mod.*
		rm -rf .*.cmd
		rm -rf .tmp*
		rm -rf *.o
```

Module

- Make module : sample modules (microcom homepage)
- Load the module
  - » ***insmod hello.o*** or ***hello.ko***
  - » ***dmesg*** : at host - /var/log/messages
- Check that the module is loaded
  - » ***cat /proc/modules***
- Remove the module
  - » ***rmmod hello***