Linux Kernel Compile and System Calls

Chung-Ang University

Monolithic Kernel vs. Microkernel

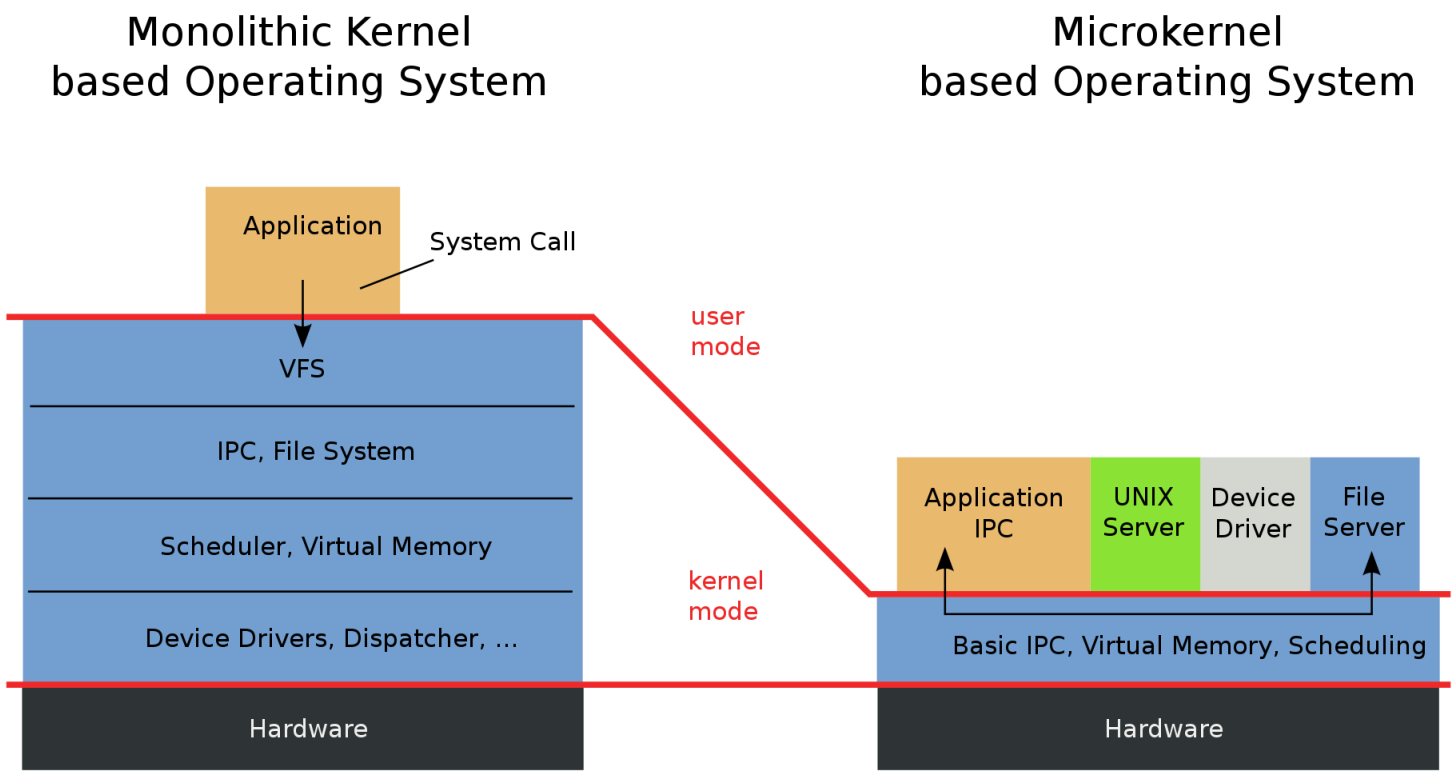
- **A kernel is the most important part of an operating system**
  - There are different design principles developing a kernel
- **Monolithic kernel**
  - A Monolithic kernel is an OS architecture
  - It includes most components (e.g., device drivers, file system, and IPC) in kernel space
  - Monolithic kernels are able to dynamically load (and unload) executable modules at runtime
  - Examples of operating systems that use a monolithic kernel are: Linux, BSDs, Solaris, Microsoft Windows, etc

Monolithic Kernel vs. Microkernel

- **Microkernel**
  - It is the near-minimum amount of software
  - It can provide the mechanisms to implement an operating system
  - There are mechanisms including address space management, thread management, and inter-process communication (IPC)

Monolithic Kernel vs. Microkernel

- **Structures**



Monolithic Kernel vs. Microkernel

- **Advantages of Monolithic Kernel**
  - One of the major advantage of having monolithic kernel is that it provides CPU scheduling, memory management, file management and other operating system functions through system calls.
- **Disadvantages of Monolithic Kernel**
  - One of the major disadvantage of monolithic kernel is that, if any service fails, it leads to entire system failure
  - If user has to add any new service, user needs to modify entire operating system

Monolithic Kernel vs. Microkernel

- **Key differences between Monolithic Kernel and Microkernel**

Basic for Comparision	Microkernel	Monolithic Kernel
Size	Microkernel is smaller than monolithic kernel	It is larger than microkernel
Execution	Slow Execution	Fast Execution
Extendible	It is easily extendible	It is hard to extend
Failure	If a service crashes, it never affects the working of microkernel	If a service crashes, the whole system crashes in monolithic kernel
Example	MkLinux, FreeRTOS	Linux, Solaris

How can we compile kernel?

- The **make** command in Linux is one of the most frequently used commands by the system administrators and the programmers.
  - Programmers use it to manage the compilation of their large and complicated projects.
  - Makefiles are a simple way to organize code compilation

Make command

- The make is a command or a software tool for managing and maintaining computer programs consisting many component files.
  - The make utility automatically determines which pieces of a large program need to be recompiled, and issues commands to recompile them.
  - Make reads its instruction from Makefile (called the descriptor file) by default.
  - Makefile is a way of automating software building procedure and other complex tasks with dependencies.

Make command

- **Example**

```
/* main.cpp */
#include <iostream>
#include "functions.h"

using namespace std;
Int main()
{
    print_hello();
    cout << endl;
    cout << "The factorial of 5 is " << factorial(5) << endl;
    return 0;
}
```

```
/* factorial.cpp */
#include "functions.h"

int factorial(int n)
{
    int i, fac = 1;
    if(n != 1){
        for(i = 1; i <= n; i++)
            fac *= i;
    }
    else return 1;
}
```

```
/* hello.cpp */
#include <iostream>
#include "functions.h"

using namespace std;
Int print_hello()
{
    cout << " Hello World! ";
}
```

```
/* functions.h */
#if !defined(_FUNC_H_)
#define _FUNC_H_

void print_hello();
int factorial(int n);

#endif    /* if !define(_FUNC_H_) */
```

Command Line Approach to Compile

- **g++ -c hello.cpp main.cpp factorial.cpp**
  - ls *.o

    → factorial.o hello.o main.o
- **g++ -o prog factorial.o hello.o main.o**
  - ./prog

  → Hello World!

  → The factorial of 5 is 120

- **Suppose we later modified hello.cpp, we need to:**
  - g++ -c hello.cpp
  - g++ -o prog factorial.o hello.o main.o

Example Makefile

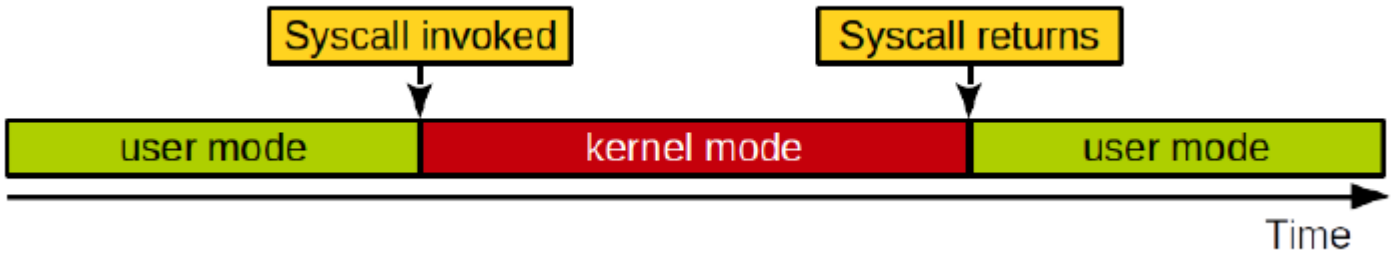
```
# This is a comment line
CC=g++
# CFLAGS will be the options passed to the compiler.
CFLAGS= -c -Wall
all: prog

prog: main.o factorial.o hello.o
	$(CC) main.o factorial.o hello.o -o prog

main.o: main.cpp
	$(CC) $(CFLAGS) main.cpp

factorial.o: factorial.cpp
	$(CC) $(CFLAGS) factorial.cpp

hello.o: hello.cpp
	$(CC) $(CFLAGS) hello.cpp

clean:
rm -rf *.o
```

How can we compile kernel?

- See the slides of the practical class

System call

- A **system call** is the programmatic way in which a computer program requests a service to the kernel of the operating system it is executed on.
  - A system call is a way for programs to interact with the operating system.
  - A computer program makes a system call when it makes a request to the operating system's kernel.
  - System call provides the services of the operating system to the user programs via Application Program Interface (API).
  - It provides an interface between a process and operating system to allow user-level processes to request services of the operating system.



System call

- Popular system calls are open, read, write, close, wait, exec, fork, exit, and kill.
- Many modern operating systems have hundreds of system calls.
  - For example, Linux and OpenBSD each have over 300 different calls, NetBSD has close to 500, FreeBSD has over 500, Windows 7 has close to 700

System call

- **System calls can be grouped roughly into six major categories**
  - Process control
    - create process
    - terminate process
    - load, execute
    - get/set process attributes
    - wait for time, wait event, signal event
    - allocate and free memory

System call

- **System calls can be grouped roughly into six major categories**
  - File management
    - create file, delete file
    - open, close
    - read, write, reposition
    - get/set file attributes
  - Device management
    - request device, release device
    - read, write, reposition
    - get/set device attributes
    - logically attach or detach devices

System call

- **System calls can be grouped roughly into six major categories**
  - Information maintenance
    - get/set time or date
    - get/set system data
    - get/set process, file, or device attributes
  - Communication
    - create, delete communication connection
    - send, receive messages
    - transfer status information
    - attach or detach remote devices
  - Protection
    - get/set file permissions

Example of Windows and Linux System calls

Process Control	fork() exit() wait()
File management	open() read() write() close()
Device management	ioctl() read() write()
Information Maintenance	getpid() alarm() sleep()
Communication	pipe() shmget() mmap()
Protection	chmod() umask() chown()

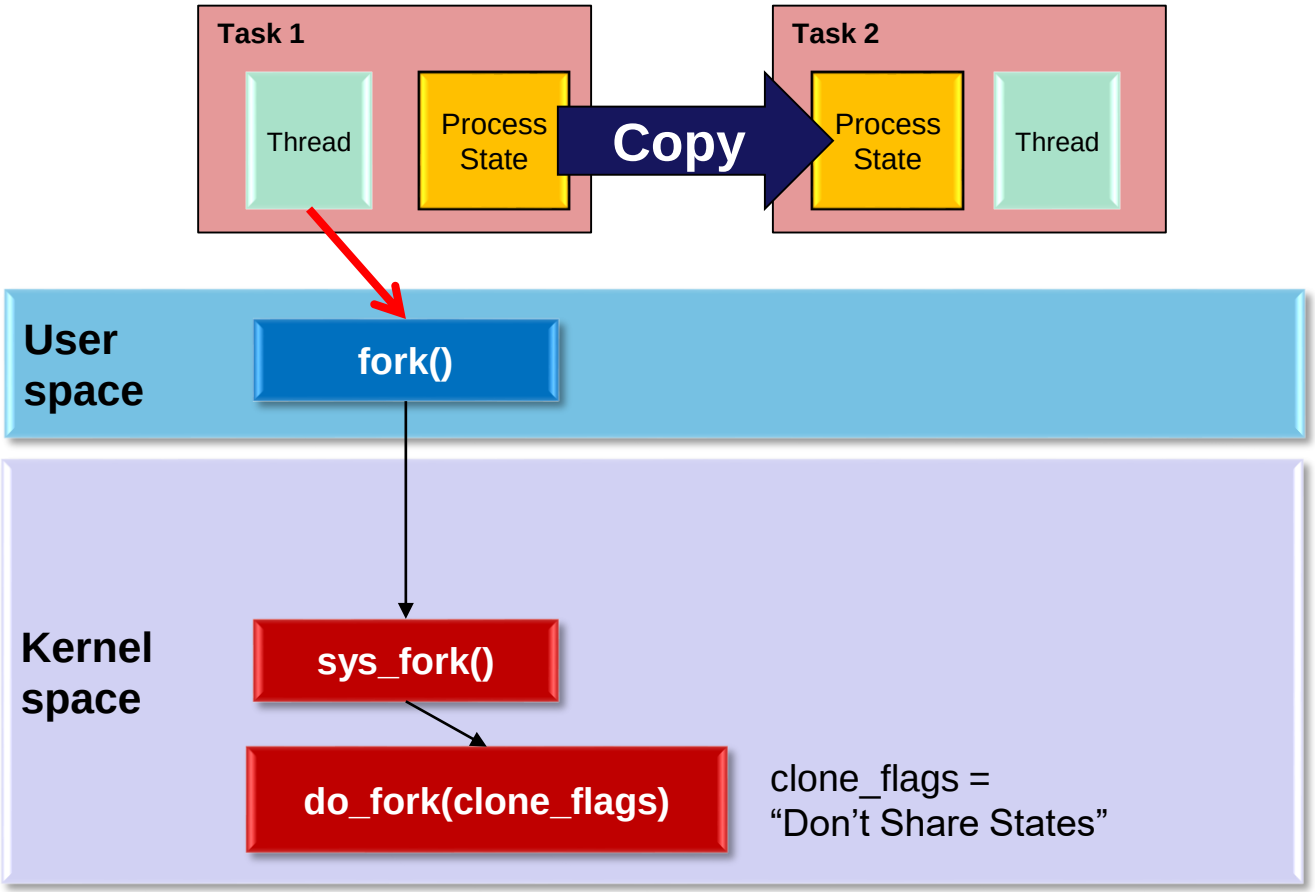
Syscall table and syscall identifier

- **Where are system call implementations in Linux kernel?**
  - The syscall table for x84_64 architecture
    - linux/arch/x86/entry/syscalls/syscall_64.tbl
  - Syscall ID: unique integer ← sequentially assigned

```
# 64-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point>
#
# The abi is "common", "64" or "x32" for this file.
0   common  read            sys_read
1   common  write           sys_write
2   common  open            sys_open
...
332 common  statx           sys_statx
```

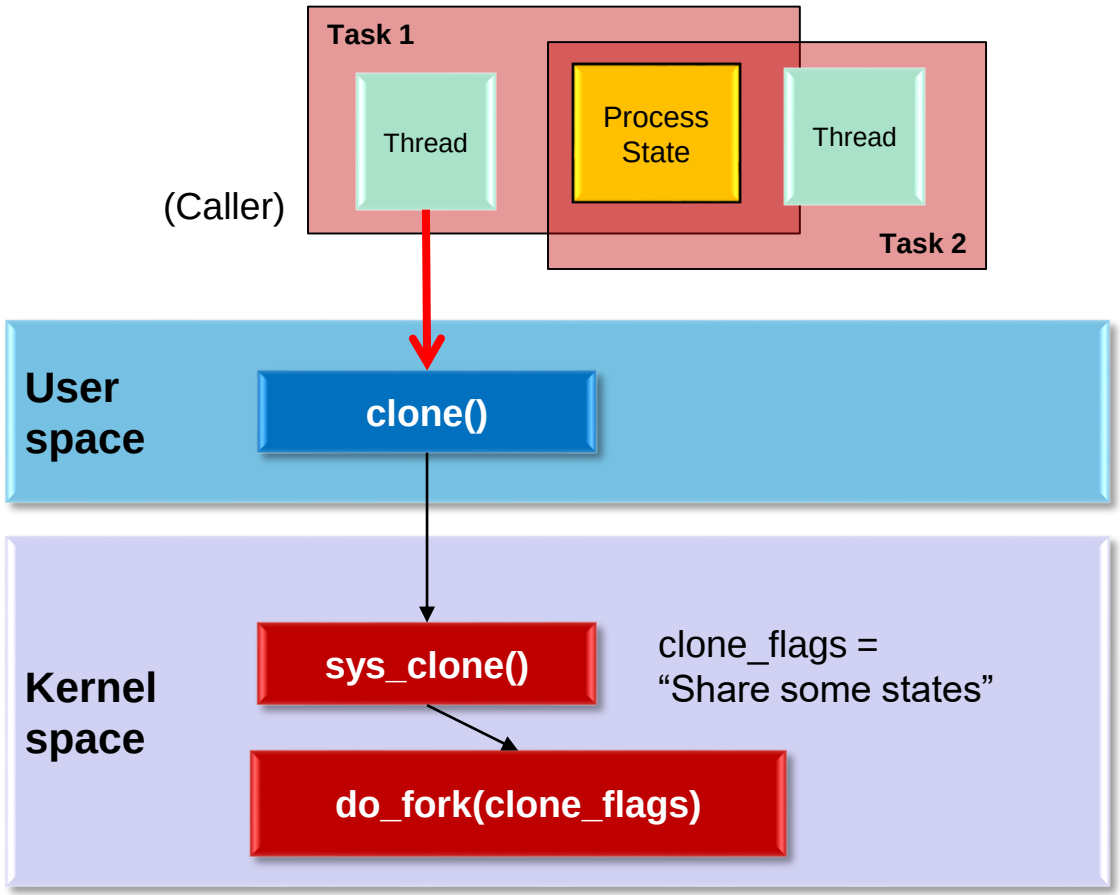
System calls for Task Creation

- **Fork()**
  - Creates a task by copying all the states of the caller task



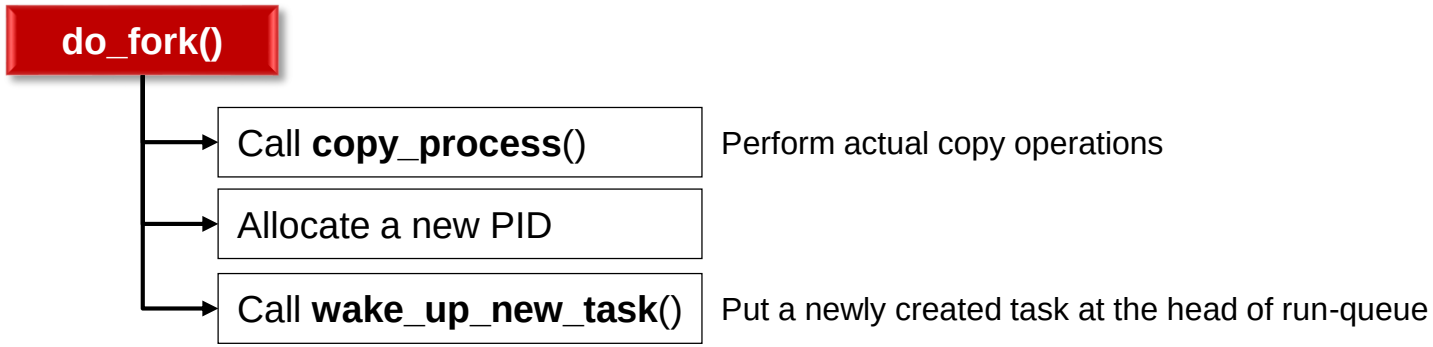
System calls for Task Creation

- **clone()**
  - Creates a task that shares a subset of states of the caller task



Implementation of do_fork()

- **Execution flow**



kernel/fork.c

```
long do_fork(unsigned long clone_flags, unsigned long stack_start,
             struct pt_regs *regs, unsigned long stack_size,
             int __user *parent_tidptr, int __user *child_tidptr)
{
   …
```

How can we make system call?

- **See the slide of practical class**