Introduction to Operating Systems

CPSC/ECE 3220 Summer 2018

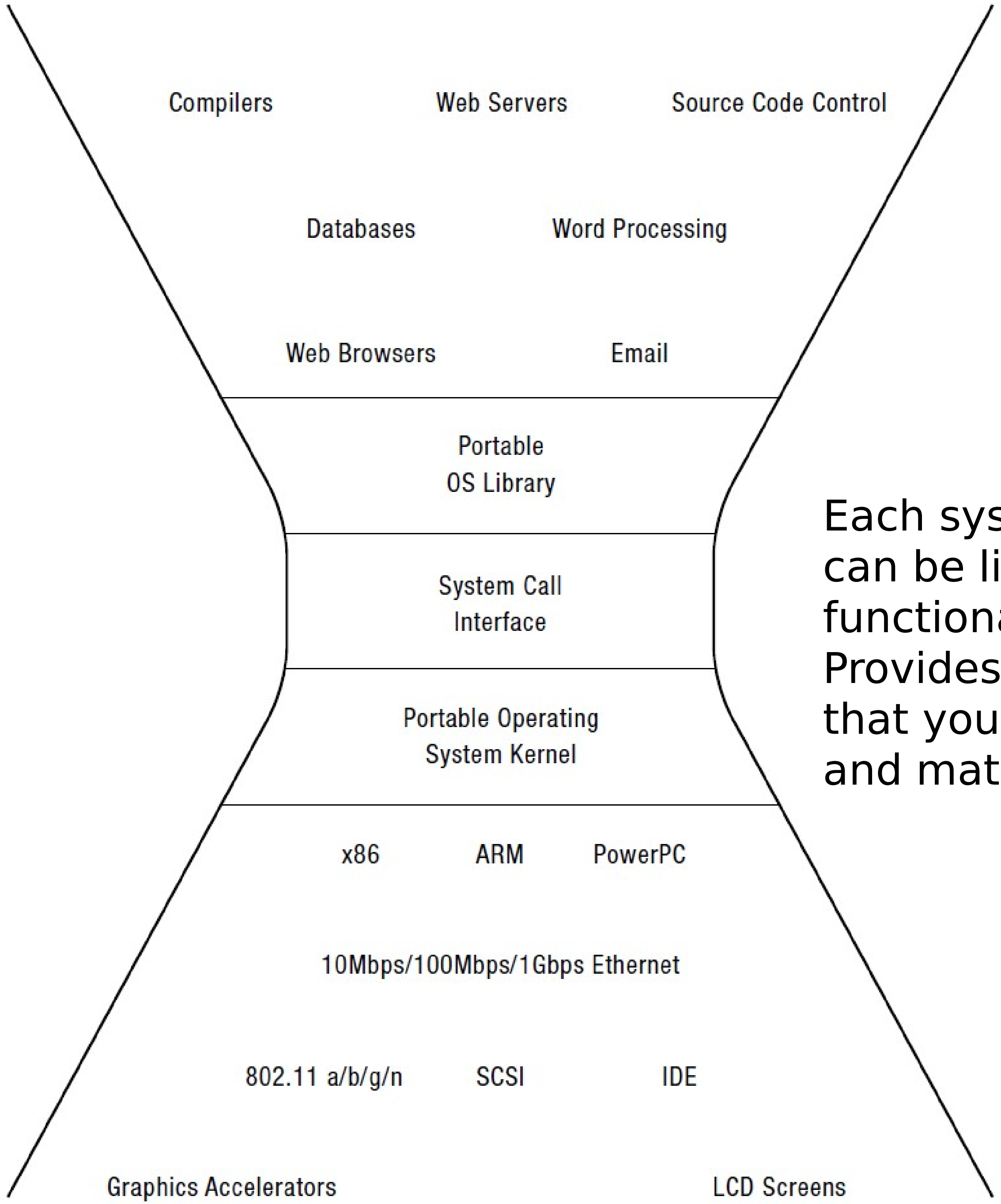
Lecture Notes
OSPP Chapter 3

(adapted by Mark Smotherman from Tom Anderson's slides on OSPP web site)

OS Design Questions

- What functions should OS provide?
- Where should this functionality be located?
  - In library routines that run in user mode?
  - In the kernel?
  - In OS server processes that run in kernel mode?
  - In OS server processes that run in user mode?
  - (Note that clients and servers will need to make kernel calls to communicate)

“thin waist”



Each system call can be limited in functionality. Provides primitives that you can “mix and match”.

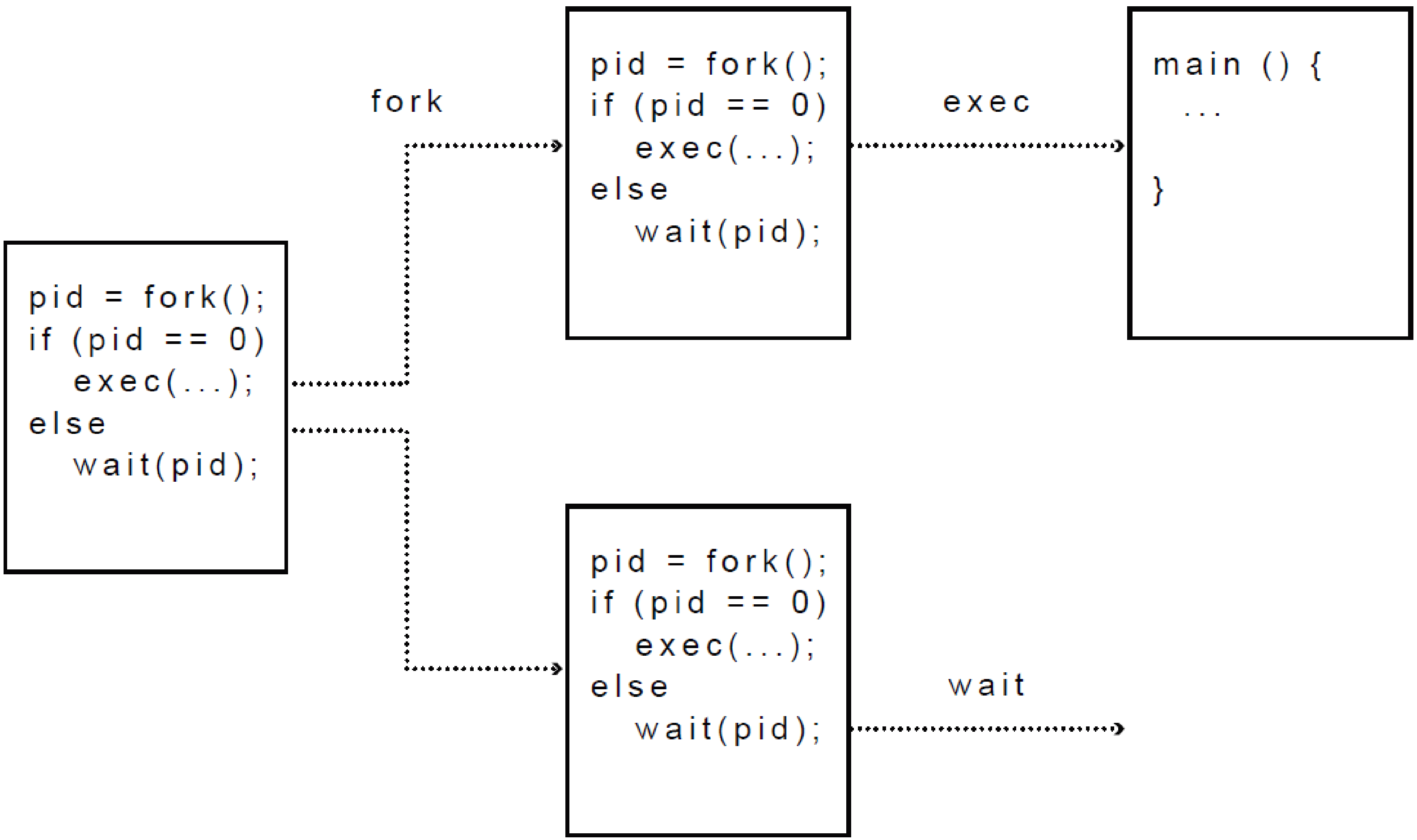
UNIX Case Study

- Creating and managing processes
  - fork(), exec(), wait()
- Performing I/O
  - open(), read(), write(), close()
- Communicating between processes
  - pipe(), dup2(), select(), connect()

UNIX Process Management

- UNIX fork() – system call to create a copy of the current process, and start it running
  - No arguments!
- UNIX exec() – system call to change the program being run by the current process
- UNIX wait() – system call to wait for a process to finish
- UNIX signal() – system call to send a notification to another process

UNIX Process Management



Question: What does this code print?

```
int child_pid = fork();
if (child_pid == 0) {           // I'm the child process
    printf("I am process #%d\n", getpid());
    return 0;
} else {                        // I'm the parent process
    printf("I am parent of process #%d\n", child_pid);
    return 0;
}
```

Questions

- Can UNIX fork() return an error? Why?
- Can UNIX exec() return an error? Why?
- Can UNIX wait() ever return immediately? Why?

Implementing UNIX fork()

- Create and initialize the process control block (PCB) in the kernel
- Create a new address space
- Initialize the address space with a copy of the entire contents of the address space of the parent
- Inherit the execution context of the parent (e.g., any open files)
- Inform the scheduler that the new process is ready to run

Implementing UNIX exec()

- Load the program into current address space
- Copy arguments into memory in address space
- Initialize the hardware context to start execution at ``start'‘

(note that exec() does not create a new process)

Windows CreateProcess()

- Create and initialize the process control block (PCB) in the kernel
- Create and initialize a new address space
- Load the program into address space
- Copy arguments into memory in address space
- Initialize the hardware context to start execution at ``start''
- Inform the scheduler that the new process is ready to run

Windows CreateProcess() API (simplified)

```
CreateProcess(
    NULL,          // No module name (use command line)
    argv[1],       // Command line
    NULL,          // Process handle not inheritable
    NULL,          // Thread handle not inheritable
    FALSE,         // Set handle inheritance to FALSE
    0,             // No creation flags
    NULL,          // Use parent's environment block
    NULL,          // Use parent's starting directory
    &si,           // Pointer to STARTUPINFO structure
    &pi            // Pointer to PROCESS_INFORMATION structure
)
```

UNIX I/O

- Uniformity
  - All operations on files, I/O devices, and pipes use the same set of system calls: open(), close(), read(), write()
- Open before use
  - Check permissions and set up internal housekeeping
  - open() returns a handle (file descriptor) for use in later calls on the file
- Byte-oriented interface
- Kernel-buffered read() and write()
- Explicit close()

UNIX File System Interface

- UNIX file open() is a Swiss Army knife:
  - Open the file, return file descriptor
  - Options:
    - if file doesn’t exist, return an error
    - If file doesn’t exist, create file and open it
    - If file does exist, return an error
    - If file does exist, open file
    - If file exists but isn’t empty, nix it then open
    - If file exists but isn’t empty, return an error
    - …

Interface Design Question

- Why not separate system calls for open()/create()/exists()?

```
if (!exists(name))
    create(name);   // can create() fail?
fd = open(name);   // does the file exist?
```

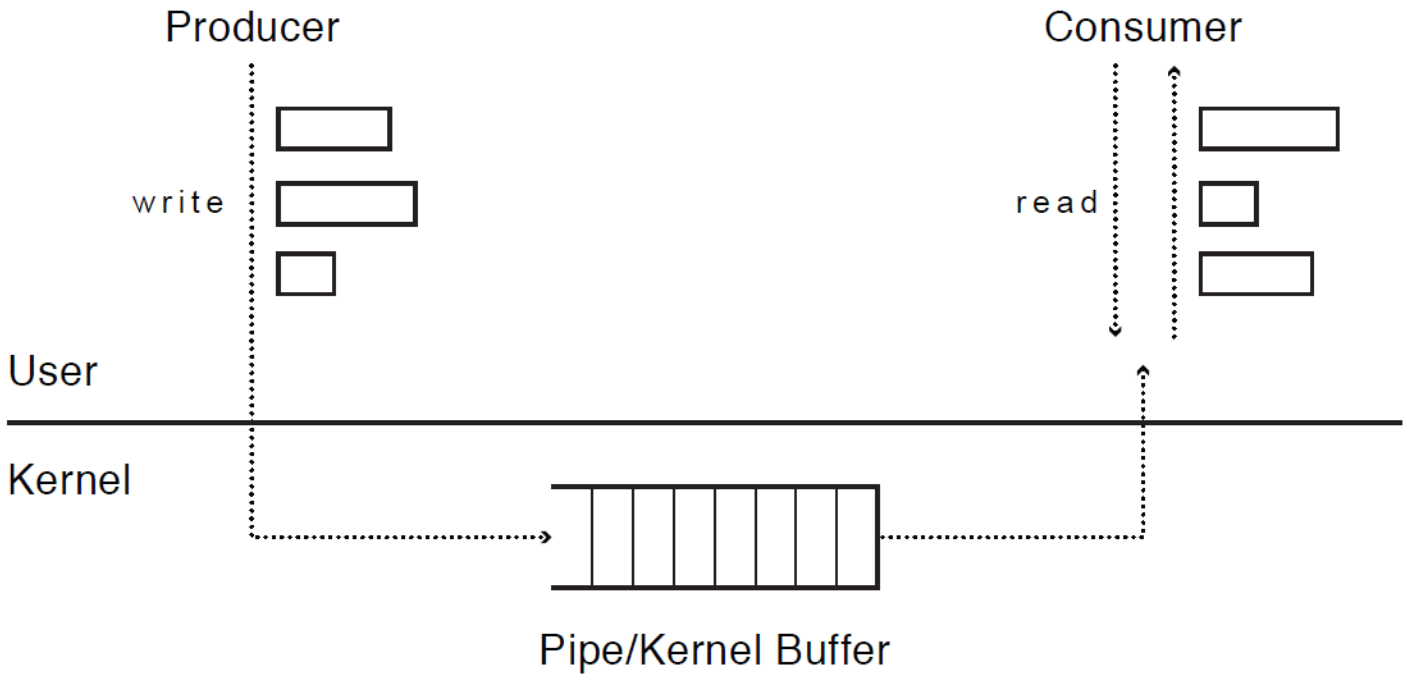
Command Interpreter ("Shell")

- Interactive interface to OS system calls
- Finds executable file associated with a command and creates a process (passing any parameters)
- Typically extended as a mini-language (e.g., control structures, macros)
- Shell scripts ("batch files" on Windows)
- Start-up files (e.g., .cshrc) and environment variables (predefined macros)
- Handles I/O redirection

Implementing a Shell

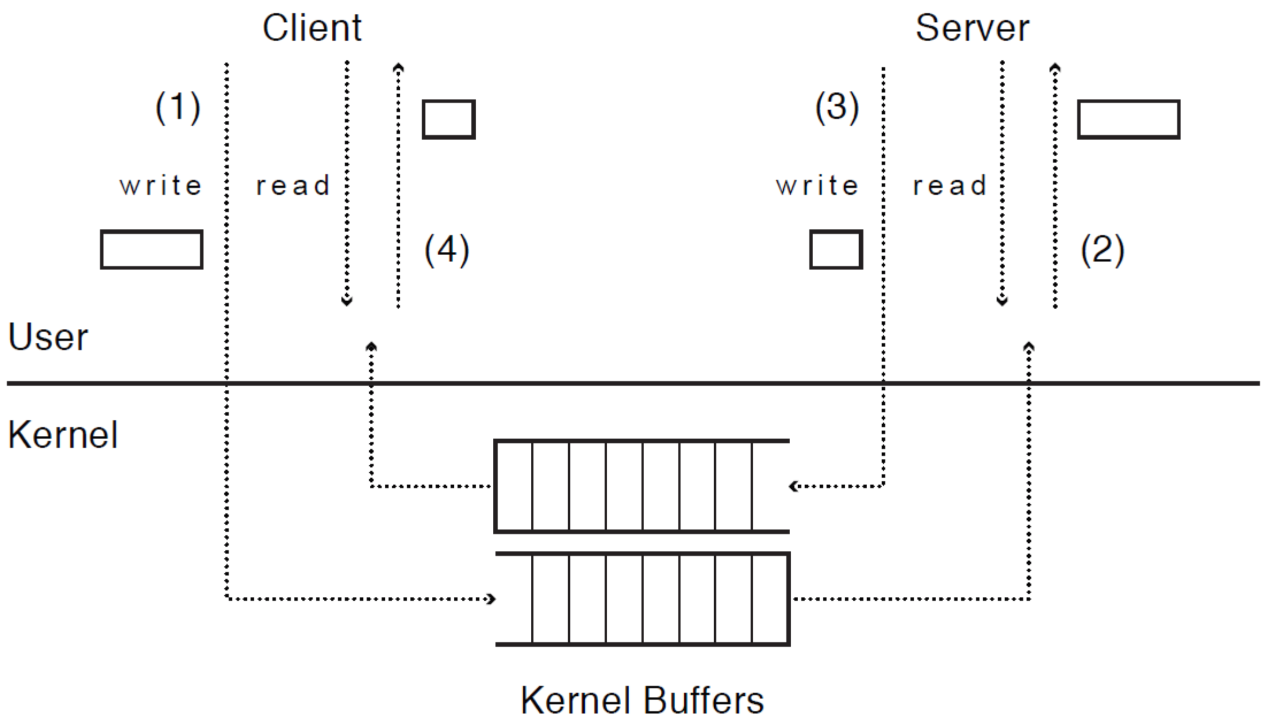
```
char *prog, **args;
int child_pid;

// Read and parse the input a line at a time
while (readAndParseCmdLine(&prog, &args)) {
   child_pid = fork();      // create a child process
    if (child_pid == 0) {
       exec(prog, args);       // I'm the child process.  Run
    program
      // NOT REACHED
    } else {
       wait(child_pid);       // I'm the parent, wait for child
       return 0;
    }
}
```

Interprocess Communication (IPC) Using Pipes



- Connect the I/O of two programs on the command line using a pipe ("|")
- Combine applications into complex structures

Client-Server IPC Using Pipes



- Server process can wait on input from multiple client processes using select()

OS Structure Design Choice

- Large kernel provides all or most system services
  - Sometimes called a "bloated kernel"
- Microkernel provides minimal services
  - Other services provided by user-mode servers

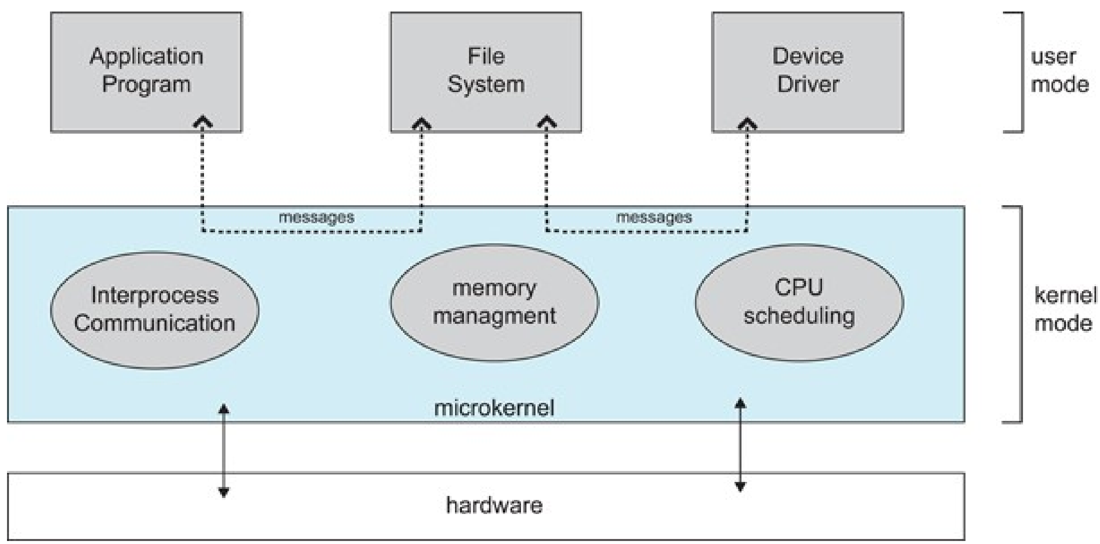


Diagram 2.14 from Silberschatz, Galvin, and Gagne, OS concepts, 9th ed.

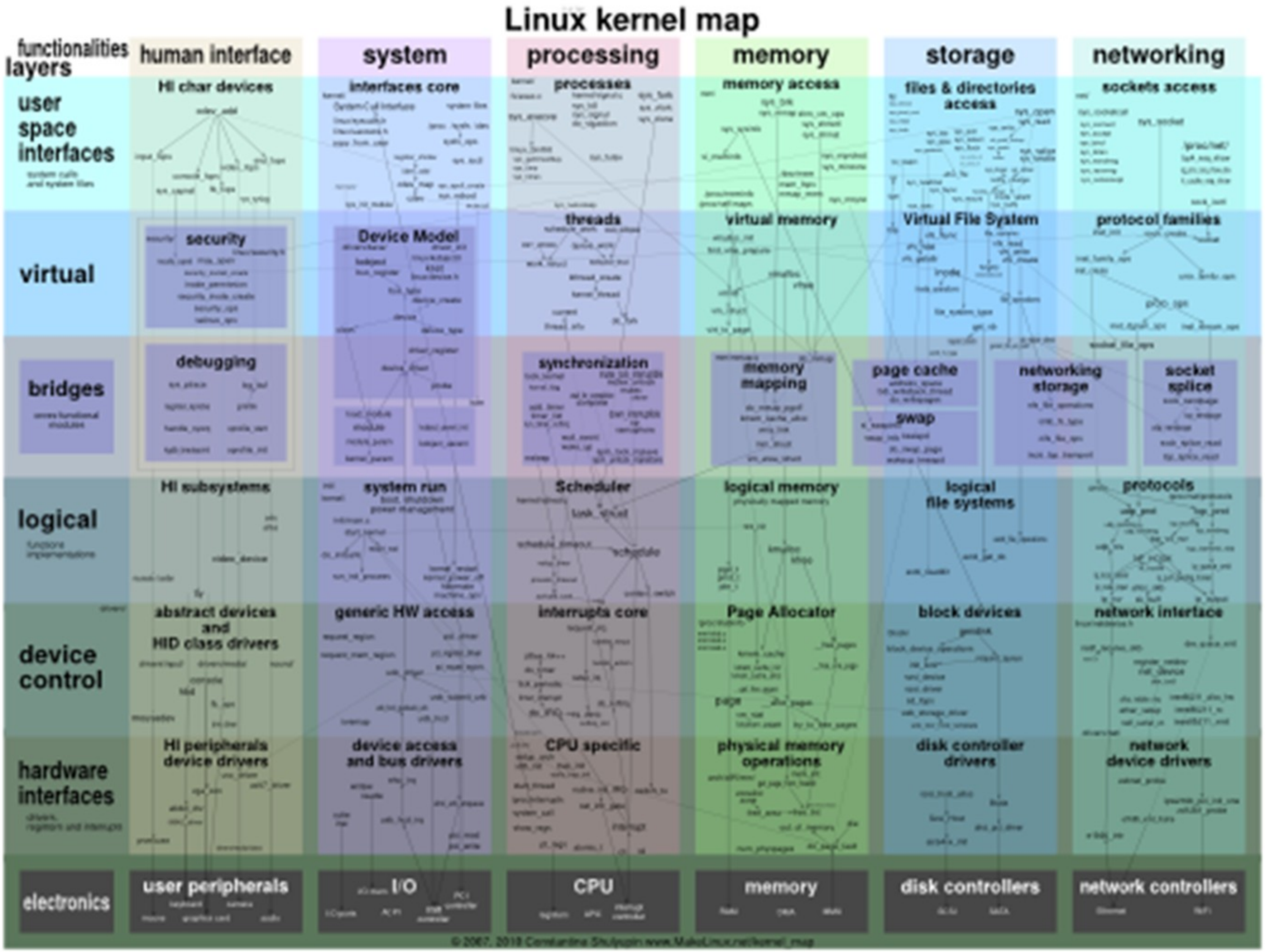


Diagram from http://www.makelinux.net/kernel_map/

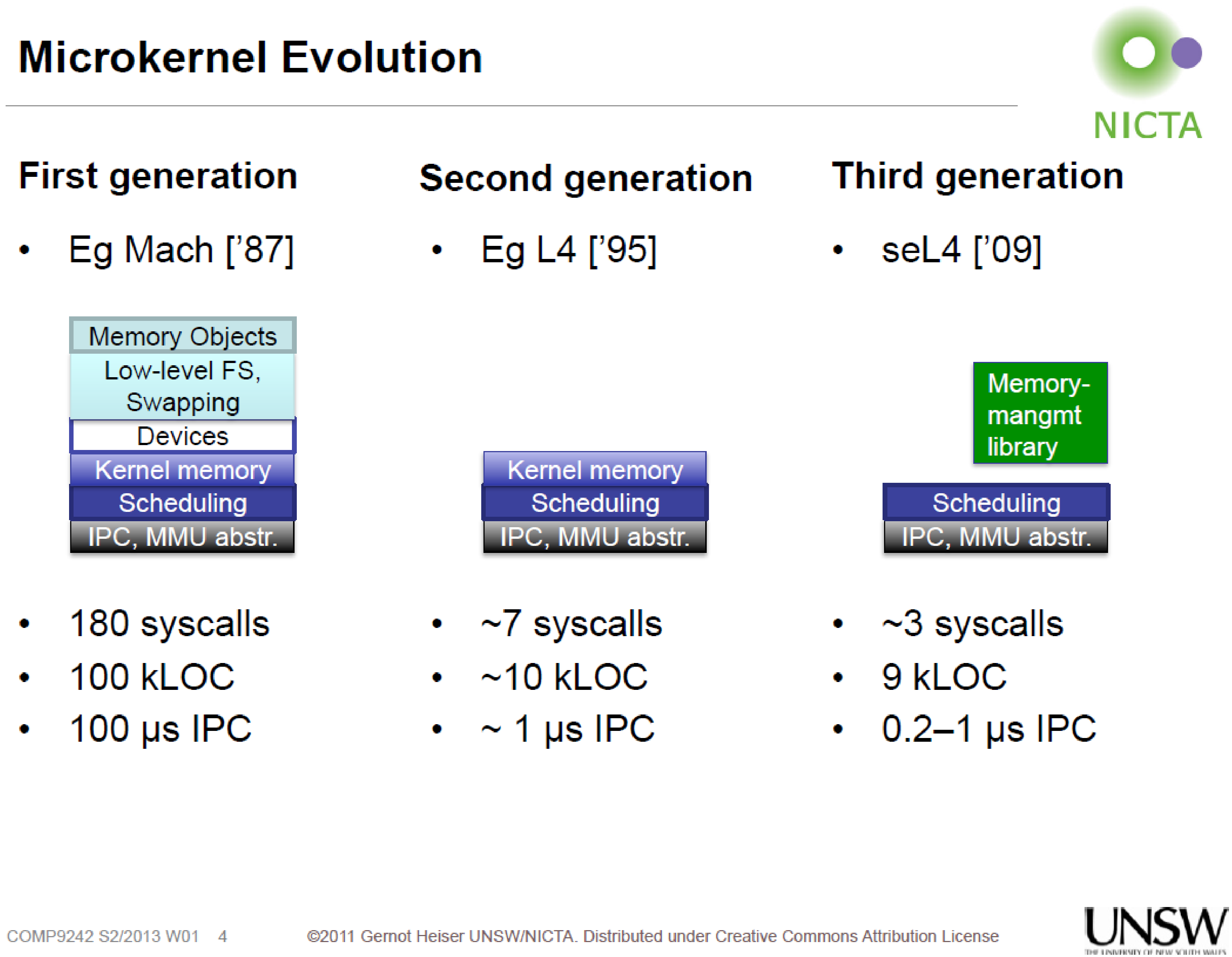


Diagram courtesy of Gernot Heiser, UNSW

Message Passing in Microkernels

- Typically define a message object for IPC
- Use send() and receive() operations
  - instead of file-like read() and write()
- Messages may be typed
  - For checking at compile time or run time
- Messages may be buffered in a queue
  - Also called a channel, endpoint, mailbox, port
- Zero-buffer approach is called rendezvous

```
// seL4 microkernel example from GitHub -  seL4/libsel4/sel4_arch_include/aarch32/sel4/sel4_arch/syscalls.h

LIBSEL4_INLINE_FUNC void
seL4_Send(seL4_CPtr dest, seL4_MessageInfo_t msgInfo)
{
    arm_sys_send(seL4_SysSend, dest, msgInfo.words[0], seL4_GetMR(0), seL4_GetMR(1), seL4_GetMR(2),
seL4_GetMR(3));
}

...

static inline void
arm_sys_send(seL4_Word sys, seL4_Word dest, seL4_Word info_arg, seL4_Word mr0,
   seL4_Word mr1, seL4_Word mr2, seL4_Word mr3)
{
    register seL4_Word destptr asm("r0") = dest;
    register seL4_Word info asm("r1") = info_arg;

    /* Load beginning of the message into registers. */
    register seL4_Word msg0 asm("r2") = mr0;
    register seL4_Word msg1 asm("r3") = mr1;
    register seL4_Word msg2 asm("r4") = mr2;
    register seL4_Word msg3 asm("r5") = mr3;

    /* Perform the system call. */
    register seL4_Word scno asm("r7") = sys;
    asm volatile (
        "swi $0"
        : "+r" (destptr), "+r" (msg0), "+r" (msg1), "+r" (msg2),
        "+r" (msg3), "+r" (info)
        : "r"(scno)
    );
}
```

swi is ARM “software interrupt” instruction to trap into the kernel

Design Tradeoffs

- Performance concerns
  - Procedure call to library routine is fastest
  - Kernel call is slower
  - Passing messages to a server process is slowest
- Reliability concerns
  - Linux 4.11 has over 18M lines of code
    - half of which is reported to be device driver code
  - seL4 microkernel is 9,700 lines of C and 500 lines of assembler
    - but only implements interrupt handling, message passing, and scheduling; furthermore, it only runs on a limited number of platforms

Linus Shares His Opinion

I think microkernels are stupid. They push the problem space into *communication*, which is actually a much bigger and fundamental problem than the small problem they are purporting to fix. They also lead to horrible extra complexity as you then have to fight the microkernel model, and make up new ways to avoid the extra communication latencies etc.

from http://meta.slashdot.org/story/12/10/11/0030249/linus-torvalds-answers-your-questions

Typical OS Design Approaches

- Hardware abstraction layer (HAL)
  - Portability across processors
  - Allows rest of OS to be written in a machine-independent manner
- Loadable device drivers
  - “Plug and Play”
  - The kernel does not need to be recompiled to work with new I/O devices
  - Trend to make device drivers run in user mode