Sistemas Embebidos de Internet de las Cosas

2/4

Diego Méndez Chaves, Ph.D

Programación de Formación en Internet de las Cosas Intel-Javeriana

Noviembre 2015

MINTIC

COLCIENCIAS
Ciencia, Tecnología e Innovación

Pontificia Universidad JAVERIANA Bogotá

Agenda

- **Processes**
- **Threads**
- **Synchronization**
- **Communication through Sockets**
- **Linux Drivers**

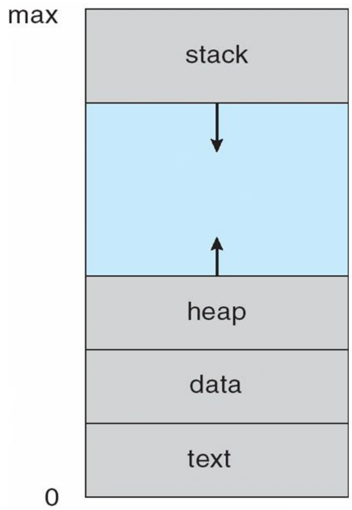
A program in execution

PROCESSES

Process Model

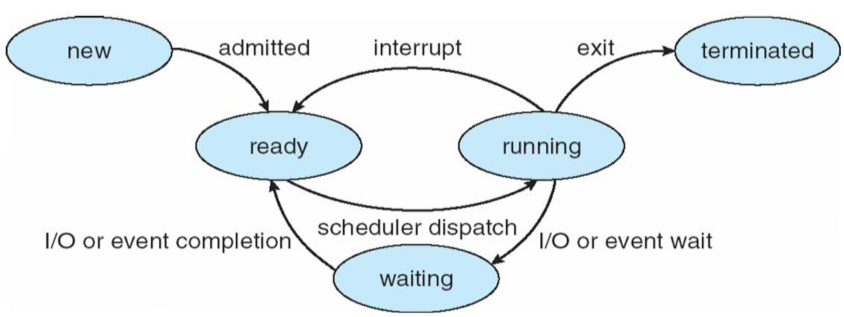
- **Why?**
  - To introduce the notion of a process – a program in execution, which forms the basis of all computation
- To describe the **various features of processes**, including
  - **scheduling**,
  - **creation,**
  - **termination**, and
  - **communication**
- To describe communication in client-server systems

Process in Memory

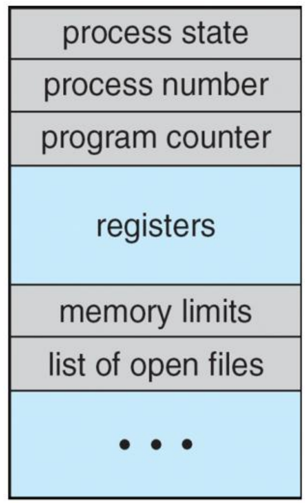


Stack = temp data
Data = global variables
Heap = dynamically allocated memory
Text = program code

Diagram of Process State

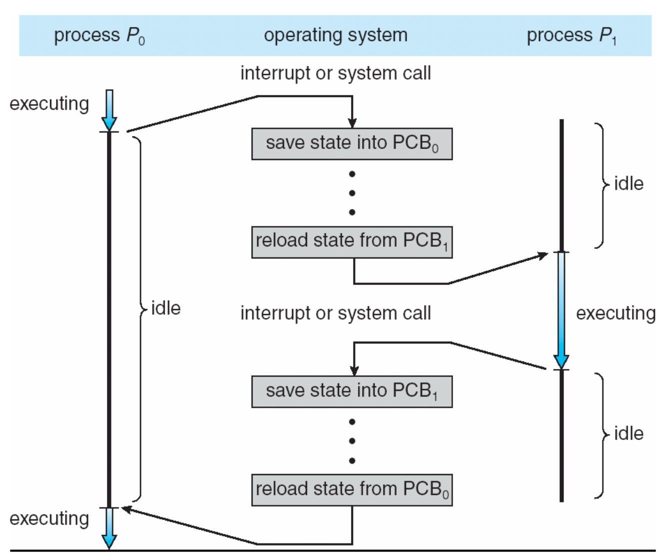


Process Control Block (PCB)



PCB is a data structure in the operating system kernel containing the information needed to manage a particular process.

CPU Switch From Process to Process



Process Creation

Four common events that lead to a process creation are:

1) When **a new batch-job** is presented for execution.
2) When **an interactive user logs in** / system initialization.
3) When **OS needs to perform an operation (usually IO) on behalf of a user process**, concurrently with that process.
4) **To exploit parallelism**, a user process can spawn a number of processes.

Reasons for the Termination of a Process

- **Normal completion, time limit exceeded, memory unavailable**
- **Bounds violation**, protection error, arithmetic error, invalid instruction
- **IO failure**, Operator intervention, **parent termination**, parent request, **killed by another process**

A number of other conditions are possible

Reasons for the Termination of a Process

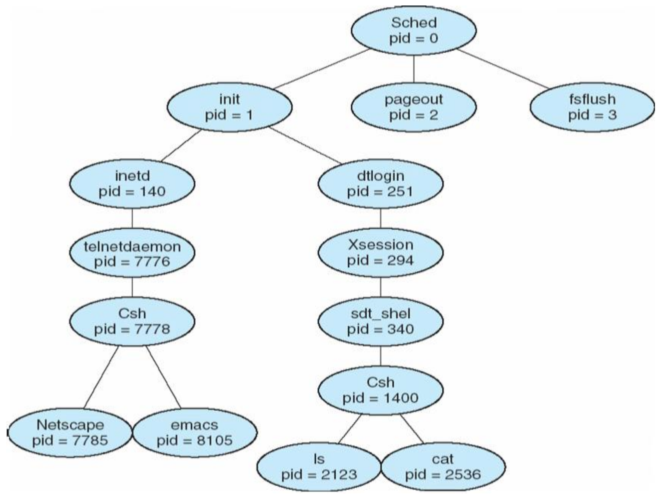
Common errors:

- **Segmentation fault**:
  - Usually happens when you **try write/read** into/from **a non-existent array/structure/object** component. Or access a pointer to a dynamic data before creating it. (new etc.)
- **Bus error:**
  - Related to function call and return. You have **messed up the stack where the return address** or parameters are stored.

Process Creation

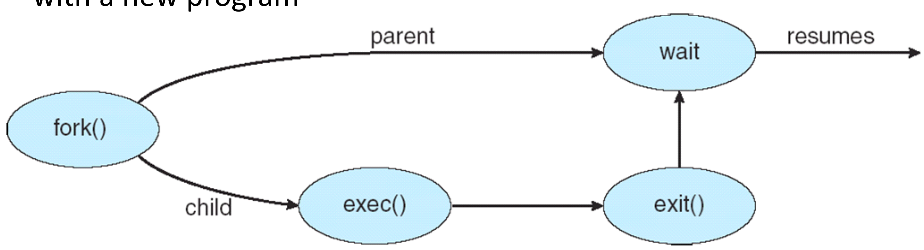
- **Parent** process creates **children** processes, which, in turn can create other processes, forming a tree of processes.
- Generally, process identified and managed via a **process identifier (pid)**
- **Resource sharing**
  - Parent and children share **all resources**
  - Children share **subset** of parent's resources
  - Parent and child share **no resources**
- **Execution**
  - Parent and children **execute concurrently**
  - Parent **waits until children terminate**

A tree of processes on a typical Solaris



Process Creation

- **Address space**
  - Child duplicate of parent
  - Child has a program loaded into it
- **UNIX examples**
  - **fork** system call creates new process
  - **exec** system call used after a **fork** to replace the process' memory space with a new program



Inter-Process Communication (IPC) and Synchronization

- An important and fundamental feature in modern operating systems is ***concurrent execution of processes/threads***.
  - This feature is essential for the realization of **multiprogramming, multiprocessing, distributed systems, and client-server model of computation.**
- **Concurrency encompasses many design issues** including
  - ***communication and synchronization*** among processes, ***sharing of and contention*** for resources.
- In this discussion we will look at the various design issues/problems and some solutions available.

Interactions among Processes

In a multi-process application these are the various degrees of interaction:

- **Competing processes:** Processes themselves do not share anything. But ***OS has to share among these processes "competing"*** some system resources, such as disk, file or printer.
- **Co-operating processes**: Results of one or more processes may be needed for another process.
  - **Co-operation by sharing**: Example: ***Sharing of an IO buffer***. Concept of critical section. (indirect)
  - **Co-operation by communication**: Example: typically no data sharing, but ***coordination through synchronization*** becomes essential in certain applications. (direct)

Interactions among Processes

- **Competing:** is at the System Level
  - Potential problems: **deadlock and starvation.**
- **Co-operation Sharing:** is at the Process Level
  - significant problem is in realizing **mutual exclusion**.
- **Co-operation Comm:** is more of a **synchronization** problem.

Race Condition (co-op processes)

- **Race condition**:
  - The situation where **several processes access – and manipulate shared data concurrently**.
  - The final value of the shared data depends upon which process finishes last.
- **To prevent race conditions**, concurrent processes must be **synchronized**.

Mutual Exclusion Problem

- **Successful** use of **concurrency** among processes requires the ***ability to define critical sections and enforce mutual exclusion.***
- **Critical section**: is that part of the process code that affects the shared resource.
- **Mutual exclusion**: in the use of a **shared resource**, mutual exclusion is provided by **making its access mutually exclusive** among the processes that share the resource.
- This is also known as the ***Critical Section (CS) problem***.

Mutual Exclusion

Any facility that provides mutual exclusion **should meet these requirements:**

- Only **one process allowed** in the CS.
- A process is in its CS for a **finite time only.**
- No assumption regarding the **relative speeds of the processes**.
- Process requesting access to CS should **not wait indefinitely.**
- A **process waiting to enter CS cannot be blocking a process** in CS or any other processes.

Message Passing

- **Both synchronization and communication requirements are taken care of by this mechanism.**
  - This mechanism yields to **synchronization** methods **among distributed processes.**
- **Basic primitives are:**

```
send (destination, message);
receive (source, message);
```

Issues in Message Passing

Send and receive: could be blocking or non-blocking:

- ***Blocking send:*** when a process sends a message it blocks until the message is received at the destination
- ***Non-blocking send:*** After sending a message the sender proceeds with its processing without waiting for it to reach the destination.
- ***Blocking receive:*** When a process executes a ***receive*** it waits blocked until the ***receive*** is completed and the required message is received.
- ***Non-blocking receive:*** The process executing the receive proceeds without waiting for the message.

Blocking Receive/Non-blocking Send is a common combination

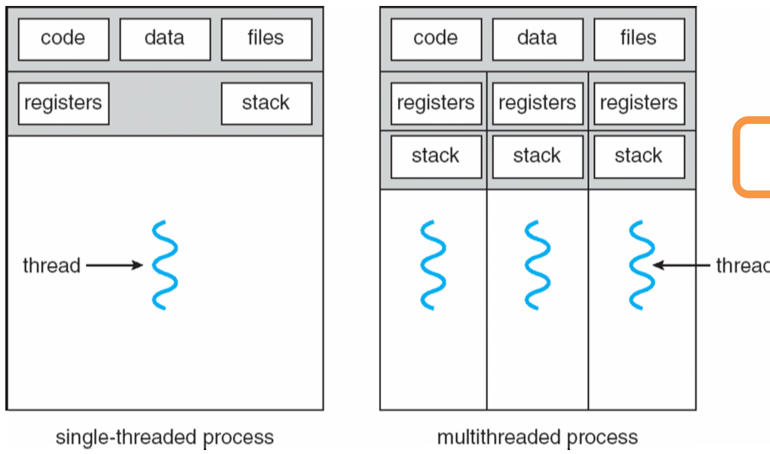
C Program Forking Separate Process

```
int main()
{
pid_t  pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
         fprintf(stderr, "Fork Failed");
         exit(-1);
    }
    else if (pid == 0) { /* child process */
         execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
         /* parent will wait for the child to complete */
         wait (NULL);
         printf ("Child Complete");
         exit(0);
    }
}
```

Divide and Conquer!

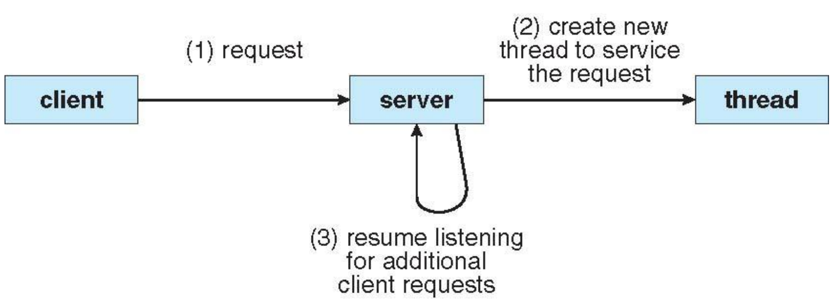
THREADS

Single and Multithreaded Processes



BENEFITS?

Multithreaded Server Architecture

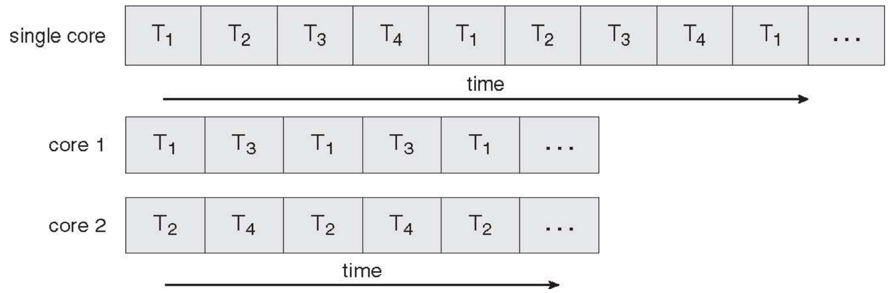


Benefits of multi-threaded programming:

- Responsiveness
- Resource sharing
- Economy (Solaris example: 30X > expensive to create a process, 5X > expensive to context-switch processes)
- Scalability

Multicore Programming

- **Each core seen by the OS as a separate processor**



- **Challenges in programming for multicore systems**
  - Identifying parallelism (dividing activities)
  - Load balance
  - Data splitting
  - Data dependency
  - Testing and debugging

We have more cores than what we can use

Thread Libraries

- **Thread library** provides programmer with API for creating and managing threads
- **Two primary ways of implementing**
  - Library entirely in user space
  - Kernel-level library supported by the OS

Thread Types (User vs Kernel Level)

User-level threads:

- **Thread management done by user-level threads library:**
  - create/destroy threads;
  - message passing or data sharing between threads;
  - scheduling threads;
  - saving/restoring threads contexts.
- **Advantages over kernel-level threads:**
  - Thread switching
  - Application-specific scheduling
  - No need of special support from OS
- **Disadvantages:**
  - One blocking system call in a user-level thread blocks all threads in the same process
  - No advantage from multiprocessing

Thread Types (User vs Kernel Level)

Kernel-Level threads:

- All work done in the kernel
- Most common OS have kernel-level threads
  - Windows XP/2000, Solaris, Linux, Tru64 UNIX, Mac OS X.

Multi-Threading Models

- Many-to-One
- One-to-One
- Many-to-Many (with the variation Two-level Model)

Two Popular Thread Libraries

Pthreads:

- May be provided either as user-level or kernel-level
- A POSIX (**P**ortable **O**perating **S**ystems **I**nterface) standard (IEEE 1003.1c) API for thread creation and synchronization
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

Java Threads:

- Java threads are managed by the JVM
- Typically implemented using the threads model provided by underlying OS
- Java threads may be created by extending Thread class and Implementing the Runnable interface

Threading Issues in OS Design

- Semantics of **fork()** and **exec()** system calls
  - Does **fork()** duplicate only the calling thread or all threads?
- Thread cancellation of target thread. Challenge: thread has resources; thread in the middle of updating data.
  - **Asynchronous cancellation** terminates the target thread immediately
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled
- Thread pools
- Thread-specific data
- Scheduler activations

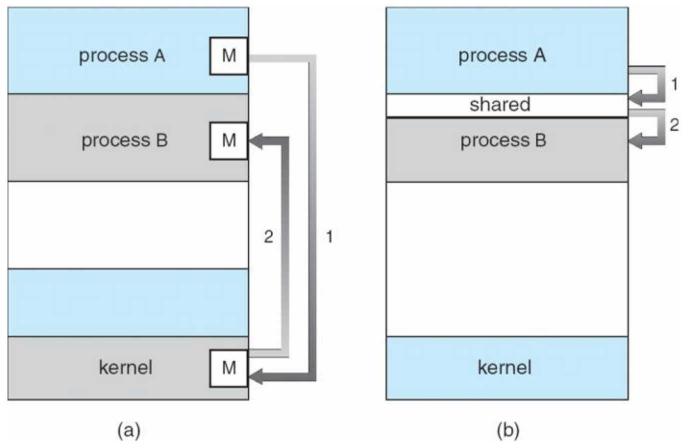
What's the Output of this Code? Why?

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* the thread */

main(int argc, char *argv[])
{
    pthread_t tid1, tid2; /*the thread identifier*/
    pthread_attr_t attr; /*set of thread attributes*/

    if (argc != 2) {
      fprintf(stderr, "usage: a.out<integer
    value>\n");
      exit(0);
    }
    if (atoi(argv[1]) < 0) {
      fprintf(stderr, "%d must be >= 0\n",
      atoi(argv[1]));
      exit(0);
    }
    pthread_attr_init(&attr);
    /* create the threads*/
    pthread_create(&tid1,&attr,runner,argv[1]);
    pthread_create(&tid2,&attr,runner,argv[1]);

    pthread_join(tid1,NULL);
    pthread_join(tid2,NULL);
    printf("sum = %d\n",sum);
}

/*The thread will begin control in this
    function*/
void *runner(void *param)
{
    int upper = atoi(param);
    int i;
    sum = 0;
    if (upper > 0) {
          for (i = 1; i <= upper; i++){
                sum += i;
                printf("sum = %d, %d\n",sum,i);
          }
    }
    pthread_exit(0);
}
```

Juggling too many balls at once!

SYNCHRONIZATION

Inter-Process Communication: Two Models



Message passing   Shared memory

Message Passing

- Provides two operations:
  - **send**(*message*)
  - **receive**(*message*)
- If *P* and *Q* wish to communicate, they need to:
  - establish a *communication link* between them
    - **Implementation of communication link**
      - **physical** (e.g., shared memory, hardware bus)
      - **logical** (e.g., logical properties)
  - **exchange messages via send/receive**

Message Passing: 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

Shared Memory

- Communicating process must **establish a region of shared memory**
- Processes are responsible for **ensuring they are not writing at the same location in the shared memory simultaneously**

POSIX Shared Memory

- Process first creates shared memory segment

```
segment id = shmget(IPC_PRIVATE, size, S_IRUSR | S_IWUSR);
```

- Process wanting access to that shared memory must attach to it

```
shared_memory = (char *) shmat(id, NULL, 0);
```

- Now the process could write to the shared memory

```
sprintf(shared_memory, "Writing to shared memory");
```

- When done a process can detach the shared memory from its address space

```
shmdt(shared_memory);
```

Producer-Consumer Problem

- **Paradigm for cooperating processes**
  - ***producer*** process produces information that is consumed by a *consumer* process
  - ***unbounded-buffer*** places no practical limit on the size of the buffer
  - ***bounded-buffer*** assumes that there is a fixed buffer size

Producer/Consumer with BUFFER_SIZE items

PRODUCER:

```
while (true) {
/*  produce an item and put in
   nextProduced  */
   while (count == BUFFER_SIZE)
   ; // do nothing
   buffer [in] = nextProduced;
   in = (in + 1) % BUFFER_SIZE;
   count++;
}
```

CONSUMER:

```
while (true)  {
 while (count == 0)
  ; // do nothing
  nextConsumed =  buffer[out];
  out = (out + 1) % BUFFER_SIZE;
   count--;
/*consume the item in nextConsumed */
}
```

Problems?

How would you implement count++ or count--?

Race Condition

- `count++` could be implemented as
  ```
  register1 = count
  register1 = register1 + 1
  count = register1
  ```
- `count--` could be implemented as
  ```
  register2 = count
  register2 = register2 - 1
  count = register2
  ```
- Consider this execution interleaving with "`count = 5`" initially:

```
S0: producer execute register1 = count            {register1 = 5}
S1: producer execute register1 = register1 + 1     {register1 = 6}
S2: consumer execute register2 = count             {register2 = 5}
S3: consumer execute register2 = register2 - 1     {register2 = 4}
S4: producer execute count = register1             {count = 6 }
S5: consumer execute count = register2             {count = 4}
```

Definitions

- **Race Conditions:** two or more processes are reading and writing on shared data and the final result depends on who runs precisely when
- **Mutual exclusion:** making sure that if one process is accessing a shared memory, the other will be excluded from doing the same thing
- **Critical region:** the part of the program where shared variables are accessed

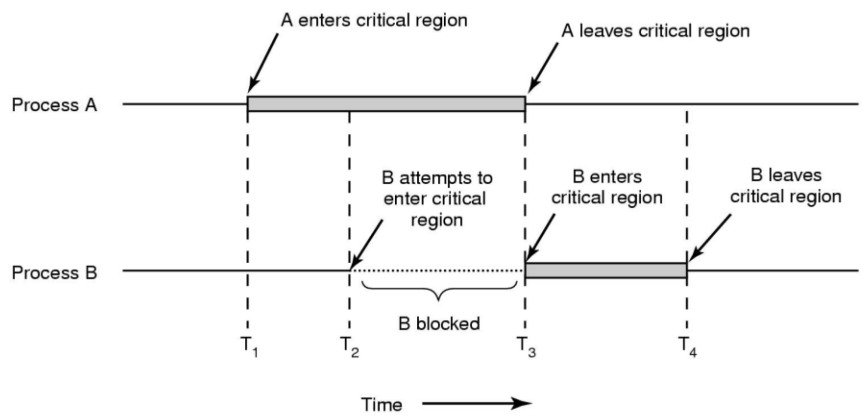
Solution to Critical-Section Problem

1. **Mutual Exclusion** - If a process is executing in its critical section, then ***no other processes can be executing*** in their critical sections
2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then ***the selection of the processes that will enter the critical section next cannot be postponed indefinitely***
3. **Bounded Waiting** - ***A bound must exist on the number of times that other processes are allowed to enter their critical sections*** after a process has made a request to enter its critical section and before that request is granted

Two assumptions:

- Assume that each process executes at a nonzero speed
- No assumption concerning relative speed of the N processes

Accessing the Critical-Section



Synchronization Hardware

- Many systems provide hardware support for critical section code
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
- Modern machines provide special atomic hardware instructions
    - Atomic = non-interruptable
  - Either test memory word and set value **(test-and-set)**
  - Or swap contents of two memory words **(swap)**

Solution to Critical-section Problem Using Locks

```
do {
    acquire lock
        critical section
    release lock
        remainder section
} while (TRUE);
```

TestAndSet Instruction

```
boolean TestAndSet (boolean *target)
 {
      boolean rv = *target;
      *target = TRUE;
      return rv:
 }
```

Solution using TestAndSet

- Shared boolean variable lock, initialized to false.
- Solution:

```
do {
     while ( TestAndSet (&lock ))
     ;   // do nothing

     //    critical section

     lock = FALSE;

     //      remainder section

} while (TRUE);
```

Swap Instruction

```
void Swap (boolean *a, boolean *b)
{
      boolean temp = *a;
      *a = *b;
      *b = temp:
}
```

Solution using Swap

- Shared Boolean variable **lock** initialized to FALSE;
- Each process has a local Boolean variable **key**
- Solution:

```
do {
    key = TRUE;
    while ( key == TRUE)
        Swap (&lock, &key);
        //  critical section
    lock = FALSE;
        //  remainder section
} while (TRUE);
```

Problems?

What about bounded-waiting?

Bounded-waiting Mutual Exclusion with TestandSet()

```
do {
     waiting[i] = TRUE;
     key = TRUE;
     while (waiting[i] && key)
             key = TestAndSet(&lock);
     waiting[i] = FALSE;
             // critical section
     j = (i + 1) % n;
     while ((j != i) && !waiting[j])
             j = (j + 1) % n;
     if (j == i)
             lock = FALSE;
     else
             waiting[j] = FALSE;
             // remainder section
} while (TRUE);
```

Semaphores (by Dijkstra 1930 – 2002)

- Born in Rotterdam, The Netherlands
- 1972 recipient of the ACM Turing Award
- Responsible for
  - The idea of building operating systems as explicitly synchronized sequential processes
  - The formal development of computer programs
- Best known for
  - His efficient shortest path algorithm
  - Having designed and coded the first Algol 60 compiler.
  - Famous campaign for the abolition of the GOTO statement
- Also known for his hand-written communications with friends and colleagues. For example: http://www.cs.utexas.edu/users/EWD/ewd12xx/EWD1205.PDF

Semaphores

- Think about a **semaphore as a class**
  - ***Attributes:*** *semaphore value*
  - ***Functions:*** *init, wait, signal*
- **Support provided by OS**
- **Can easily implement mutual exclusion** among any number of processes.

Semaphore

- Semaphore *S* – integer variable
- Two standard operations modify S: wait() and signal()
  - Originally called P() and V()
- Less complicated
- Can only be accessed via two indivisible (atomic) operations
  - ```
    wait (S) {
         while S <= 0
            ; // no-op
         S--;
    }
    ```
  - ```
    signal (S) {
         S++;
    }
    ```

Semaphore as General Synchronization Tool

- Counting semaphore – integer value can range over an unrestricted domain
- Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement
  - Also known as mutex locks
- Can implement a counting semaphore S as a binary semaphore
- Provides mutual exclusion

```
Semaphore mutex;    //  initialized to 1
do {
   wait (mutex);
        // Critical Section
   signal (mutex);
        // remainder section
} while (TRUE);
```

Semaphores (cont)

A semaphore is like an integer, with three differences:

1. When you create the semaphore, you can initialize its value to any integer, but after that the only operations you are allowed to perform are **increment** (increase by one) and **decrement** (decrease by one). **You cannot read the current value of the semaphore**.
2. **When a thread decrements the semaphore, if the result is negative, the thread blocks itself** and cannot continue until another thread increments the semaphore.
3. When a thread increments the semaphore, if there are other threads waiting, **one of the waiting threads gets unblocked**.

Semaphore Implementation with no Busy waiting

- **With each semaphore there is an associated waiting queue.** Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list
- Two operations:
  - **block** – place the process invoking the operation on the appropriate waiting queue.
  - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue.

Semaphore Implementation with no Busy waiting (Cont.)

- Implementation of wait:

```
wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
                add this process to S->list;
                block();
        }
}
```

- Implementation of signal:

```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
                remove a process P from S->list;
                wakeup(P);
        }
}
```

Deadlock and Starvation

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let S and Q be two semaphores initialized to 1

P_0	P_1
`wait (S);`	`wait (Q);`
`wait (Q);`	`wait (S);`
.	.
.	.
.	.
`signal (S);`	`signal (Q);`
`signal (Q);`	`signal (S);`

- Starvation – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended

Consequences

- In general, **there is no way to know** before a thread decrements a semaphore **whether it will block or not.**
- **After a thread increments a semaphore and another thread gets woken up, both threads continue running concurrently**. There is no way to know which thread, if either, will continue immediately.
- When you signal a semaphore, you don't necessarily know whether another thread is waiting, so **the number of unblocked threads may be zero or one.**

Syntax (in pseudo-language)

sem = semaphore (value)

Multiple equivalent notations:

- sem.signal(); sem.wait()
- sem.up(); sem.down()
- sem.V(); sem.P()

Basic Synchronization Patterns

- Serialization
- Rendez-vous
- Mutual exclusion
- Multiplex

Serialization

S=0

Thread A	**Thread B**
	Wait (S)
statement a1	statement b1
Signal (S)	

Requirement: statement a1 runs before statement b1.

Rendez-vous

A=B=0

Thread A	**Thread B**
statement a1	statement b1
Signal (A)	**Signal (B)**
Wait (B)	**Wait (A)**
statement a2	statement b2

Requirement: Thread A has to wait for Thread B and vice-versa:

a1 happens before b2,

b1 happens before a2.

Mutual Exclusion

mutex=1

Thread A	**Thread B**
mutex.wait()	**mutex.wait()**
count++;	count--;
mutex.signal()	**mutex.signal()**

Requirement: only one thread accesses the shared variable at a time.

Multiplex

Multiplex = semaphore (N)

Thread i

Multiplex.wait()

critical section;

Multiplex.signal()

Requirement: allow multiple threads to run in the critical section at the same time, but not more than n threads. (think busy nightclubs/bouncer)

Communicating over an IP network

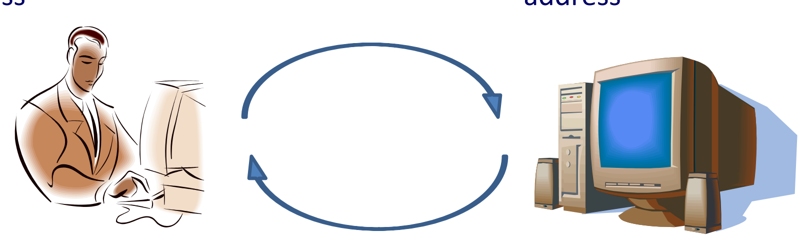
SOCKETS

The Client-Server Model

- These terms refer to the two processes which will be communicating with each other.
- One of the two processes, **the client, connects** to the other process, the server, to make a **request for information**.
- ***A person who makes a phone call to another person.***
- The **client needs to know** of the existence of and the **address** of the **server**, but
  - the **server does not need to know the address of** (or even the existence of) **the client** prior to the connection being established.

The Client-Server Model

- **Client "sometimes on"**
  - Initiates a request to the server when interested
  - E.g., Web browser on your laptop or cell phone
  - Doesn't communicate directly with other clients
  - Needs to know the server's address
- **Server is "always on"**
  - Services requests from many client hosts
  - E.g., Web server for the www.cnn.com Web site
  - Doesn't initiate contact with the clients
  - Needs a fixed, well-known address

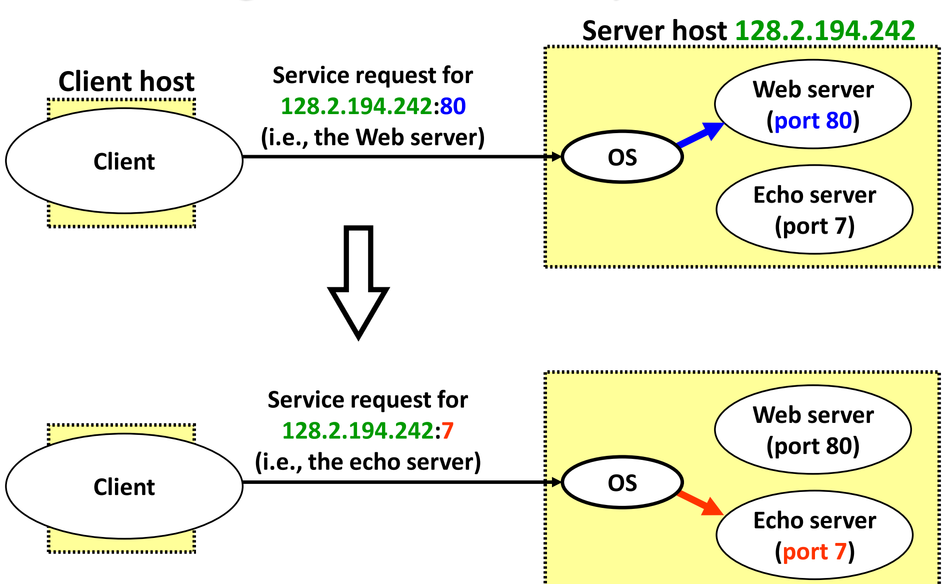


Socket Types

We need to specify the Address Domain and the Socket Type:

- **Address Domain:**
  - ***Unix Domain:*** the systems share a common file system. An address is a character string, basically an entry in the file system.
  - ***Internet Domain:*** communication between two hosts using the Internet. Address is a unique 32 bit value, plus a 16bit unsigned port number (ports above 2000 are normally available).
- **Socket Type:**
  - ***Stream Sockets:*** communication as a continuous stream of characters (TCP – Transmission Control Protocol)
  - ***Datagram Sockets:*** read entire message at once (UDP – Unix Datagram Protocol)

Using Ports to Identify Services



Knowing What Port Number to Use

- **Popular applications have well-known ports**
  - E.g., port 80 for Web (http) and port 25 for e-mail
  - 20,21: FTP; 23: Telnet
  - see RFC 1700 (about 2000 ports are reserved)
  - Well-known ports listed at http://www.iana.org
- **Well-known vs. ephemeral ports**
  - Server has a well-known port (e.g., port 80)
    - Between 0 and 1023
  - Client picks an unused ephemeral (i.e., temporary) port
    - Between 1024 and 65535
- **Uniquely identifying the traffic between the hosts**
  - Two IP addresses and two port numbers
  - Underlying transport protocol (e.g., TCP or UDP)

Delivering the Data: Division of Labor

- **Network**
  - Deliver data packet to the destination host
  - Based on the destination IP address
- **Operating System**
  - Deliver data to the destination socket
  - Based on the protocol and destination port #
- **Application**
  - Read data from the socket
  - Interpret the data (e.g., render a Web page)

Creating a Connection

On the client side:

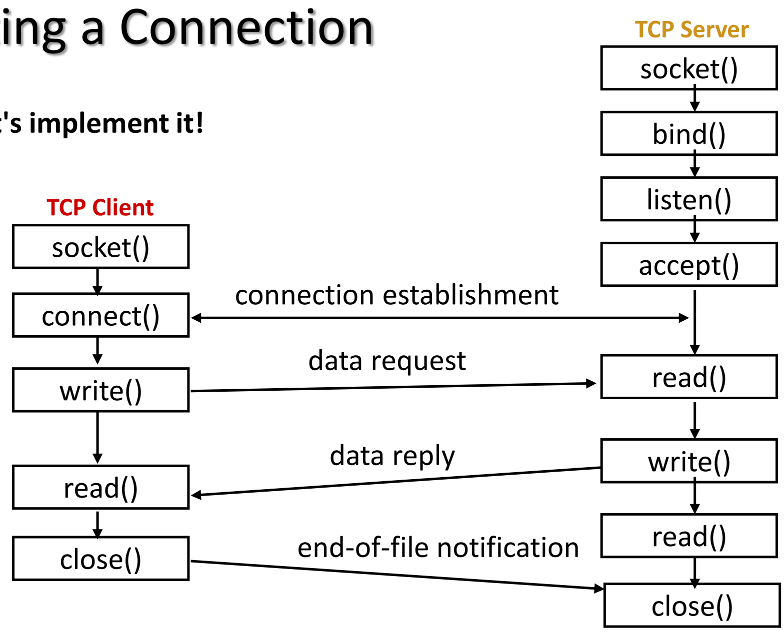
- Create a socket with the **socket()** system call
- Connect the socket to the address of the server using the **connect()** system call
- Send and receive data. There are a number of ways to do this, but the simplest is to use the **read()** and **write()** system calls.

On the server side:

- Create a socket with the **socket()** system call
- Bind the socket to an address using the **bind()** system call. For a server socket on the Internet, an address consists of a port number on the host machine.
- Listen for connections with the **listen()** system call
- Accept a connection with the **accept()** system call. This call typically blocks until a client connects with the server.
- **Send and receive data**.

Creating a Connection

Let's implement it!



From: UNIX Network Programming Volume 1, figure 4.1

Server Side Program

Basic code for the Server side and Client side can be found in:

- **Sockets Tutorial www.LinuxHowtos.org**
  http://www.linuxhowtos.org/C_C++/socket.htm

What if I want to include unsupported new hardware?

DRIVERS

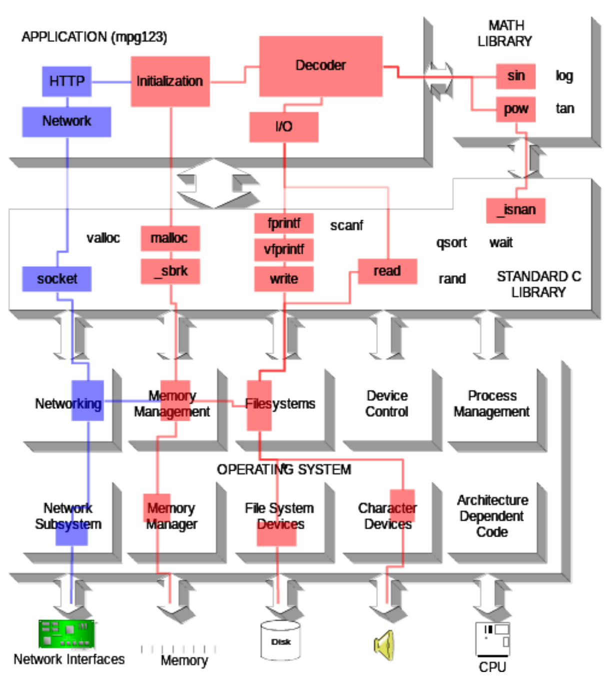
Concept

- **The role of device driver**
  - To allow interaction with hardware devices.
  - Providing *mechanism*, not *policy*.
    - **Mechanism:** What capabilities are to be provided?
    - **Policy:** How those capabilities can be used?
- **Writing a Linux device driver**
  - Pre-requisites
    - C programming
    - Microprocessor programming
  - Important concepts
    - User space vs. kernel space

Concept

Execution paths:

From user to kernel



Concept

Classes of devices:

- **Characters devices**
  - Can be accessed as a stream of bytes.
  - Such a driver usually implements at least the *open*, *close*, *read*, and *write* system calls.
  - Example: RTC driver.
- **Block devices**
  - A device (e.g., a disk) that can host a filesystem.
  - Example: Ramdisk driver.
- **Network interfaces**
  - In charge of sending and receiving data packets, driven by the network subsystem of the kernel.
  - Example: Network card driver.

Concept

However,

- **A device driver is a kernel module.**
- **A kernel module is a device driver?**
  - Not necessarily so.
  - Example: ext3 file system module.

Kernel Module

The first kernel module
“Hello, world”

```
#include <linux/module.h>       /* Needed by all modules */
#include <linux/kernel.h>       /* Needed for KERN_INFO */
#include <linux/init.h>         /* Needed for the macros */

static int __init hello_start(void)
{
    printk(KERN_INFO "Loading hello.ko module...\n");
    printk(KERN_INFO "Hello world\n");
    return 0;
}

static void __exit hello_end(void)
{
    printk(KERN_INFO "Goodbye cruel world: hello.ko exiting.\n");
}

module_init(hello_start);
module_exit(hello_end);

MODULE_DESCRIPTION("Example kernel module.");
MODULE_AUTHOR("Diego Mendez");
MODULE_LICENSE("GPL");
```

Kernel Module

- We require to first install the development package and build all the scripts:

```
sudo apt-get install kernel-dev
shutdown -r now
ln -s /usr/src/kernel /lib/modules/$(uname -r)/build
make -C /usr/src/kernel scripts
```

- How to compile (makefile)

```
obj-m+=hello.o
all:
        make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
clean:
        make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

- How to insert into kernel

```
insmod hello.ko
```

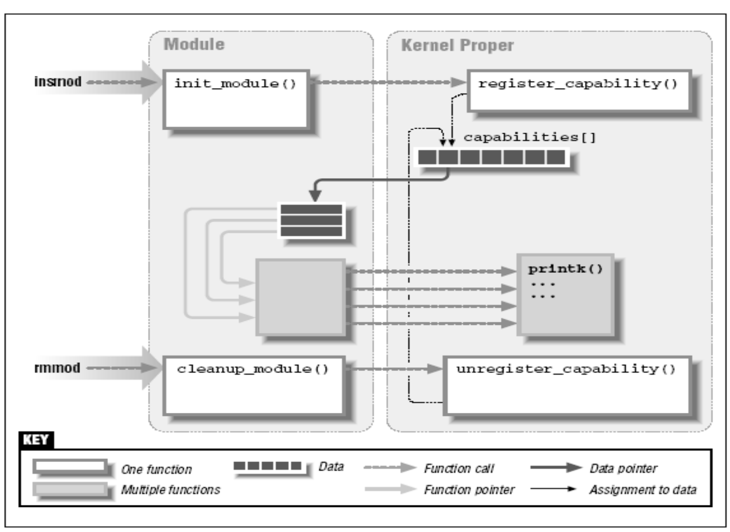
- How to remove from kernel

```
rmmod hello
```

Kernel Module

- **Kernel module vs Application**
  - **Application**
    - Runs in user space.
    - Performs a task from beginning to end.
    - Linked to the appropriate library such as *libc*.
  - **Kernel Module**
    - Runs in kernel space.
    - Registers itself in order to serve future requests.
    - Linked only to the kernel, and the only functions it can call are the ones exported by the kernel.

Kernel Module



Kernel Module

- Just about all module code has the following:

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

- ***module.h*** contains many definitions of symbols and functions needed by loadable modules.
- ***init.h*** is needed to specify your initialization and cleanup functions.

Kernel Module

- You should specify which license applies to your code. Doing so is just a matter of including one line:
- *MODULE_LICENSE("GPL");*
  - ***General Public License***

  the GPL grants the recipients of a computer program the rights of the Free Software Definition and uses copyleft to ensure the freedoms are preserved whenever the work is distributed, even when the work is changed or added to. The GPL is a copyleft license, which means that derived works can only be distributed under the same license terms.
- Other descriptive definitions that can be contained within a module include MODULE_AUTHOR, MODULE_DESCRIPTION, MODULE_VERSION … etc.

Kernel Module

- **Module initialization**

  –The actual definition of the initialization function always looks like

```
static int __init initialization_function(void)
{
    /* Initialization code here */
}
module_init(initialization_function);
```

module_init adds a special section to the module's object code stating where the module's initialization function is to be found.

Kernel Module

- **Module initialization**
  - In initialization function, you can register many different type of facilities, including different kind of devices, file systems, and more.
  - Most registration functions are prefixed with *register_* , such as:
    - `register_chrdev()`
    - `register_blkdev()`
    - `register_netdev()`

Kernel Module

- **Module cleanup**
  - The cleanup function is defined as

```
static void __exit cleanup_function(void)
{
    /* Cleanup code here */
}
module_exit(cleanup_function);
```

  - In cleanup function, you're supposed to unregister interfaces and return all resources to the system.
  - If your module is built directly into kernel or the kernel is configured to disallow the unloading of modules, functions marked __*exit* are simply discarded.

Kernel Module

`printk()`

- Kernel version of printf().
- Priority of kernel messages can be specified with the following symbols defined in <linux/kernel.h>.

High

Priority

Low

  - KERN_EMERG: Emergency message
  - KERN_ALERT: Alert message
  - KERN_CRIT: Critical situation
  - KERN_ERR: Error report
  - KERN_WARNING: Warning message
  - KERN_NOTICE: Noticeable message
  - KERN_INFO: Information
  - KERN_DEBUG: Debug message

- Example:

```
printk(KERN_DEBUG "line %s:%i\n", __FILE__, __LINE__);
```

Kernel Module

- **Error handling**

```
int __init my_init_function(void)
{
    int err;
    /* registration takes a pointer and a name */
    err = register_this(ptr1, "skull");
    if (err) goto fail_this;
    err = register_that(ptr2, "skull");
    if (err) goto fail_that;
    err = register_those(ptr3, "skull");
    if (err) goto fail_those;
    return 0; /* success */
fail_those: unregister_that(ptr2, "skull");
fail_that: unregister_this(ptr1, "skull");
fail_this: return err; /* propagate the error */
}
```

  - Error recovery is sometimes best handled with the *goto* statement.

Kernel Module

- **Module parameters**
  - Parameters are declared with the *module_param* macro, which is defined in *moduleparam.h*.

```
module_param(var_name, type, perm_mask);
```

```
#include <linux/moduleparam.h>
#include <linux/stat.h>
…
static char *word = "world";
static int times = 1;
module_param(times, int, S_IRUGO);
module_param(word, charp, S_IRUGO);
```

  - Numerous types are supported
    - *bool, invbool, charp, int, long, short, uint, ulong, ushort*

Kernel Module

- **Module parameter**
  - Permission field
    - ***S_IRUGO*** means the parameter can be read but cannot be changed.
    - ***S_IRUGO | S_IWUSR*** allows root to change the parameter.
    - Other permission definitions can be found in <linux/stat.h>.
  - The parameter value can be assigned at load time by *insmod* or *modprobe*

```
# insmod hello.o times=10 word="Mom"
```

Char Device Driver

- **Major and minor numbers**
  - Char devices are accessed through device files in the filesystem.
  - Device files are conventionally located in the ***/dev*** directory.
  - Device files for **char drivers** are identified by a **"c"** in the first column of the output of *ls -l*.
  - **Block devices** appear in */dev* as well, but they are identified by a **"b".**

Char Device Driver

- **Linux Naming Conventions**

```
fb: frame buffer
fd: (platform) floppy disks
hd: ("classic") IDE driver
lp: line printers (compare lp)
parport, pp: parallel ports
pt: pseudo-terminals (virtual terminals)
SCSI driver, also used by libATA
sr: "ROM" driver
st: magnetic tape driver
tty: terminals. ttyS: serial port driver, ttyUSB
```

Device names are usually not portable between different Unix-like system variants. For example, on some BSD systems, the IDE devices are named /dev/wd0,/dev/wd1...

Char Device Driver

- **Major and minor numbers**

```
hagar@hagar-laptop:~$ ls -l /dev
total 0
crw-rw----  1 root video    10, 175 Oct 22 09:08 agpgart
crw------T  1 root root     10, 235 Oct 22 09:08 autofs
drwxr-xr-x  2 root root         660 Oct 22 04:08 block
drwxr-xr-x  2 root root          60 Oct 22 04:08 bsg
drwxr-xr-x  3 root root          60 Oct 22 04:08 bus
drwxr-xr-x  2 root root        3940 Oct 22 18:04 char
crw-------  1 root root      5,   1 Oct 22 09:08 console
drwxr-xr-x  2 root root          60 Oct 22 09:08 cpu
drwxr-xr-x  6 root root         120 Oct 22 04:08 disk
drwxr-xr-x  2 root root         120 Oct 22 09:08 dri
crw-------  1 root root     10,  61 Oct 22 09:08 ecryptfs
crw-rw----  1 root video    29,   0 Oct 22 09:08 fb0
crw-rw----  1 root video    29,   1 Oct 22 09:08 fb1
```

Device type

Major number

Minor number

Char Device Driver

- **Major and minor numbers**
  - The major number identifies the driver associated with the device.
  - The **kernel uses the major** number at *open* time to **dispatch** execution to the **appropriate driver**.
  - The **minor number is used only by the driver** specified by the major number.
  - **It is common for a driver to control several devices**
    - the minor number provides a way for the driver to differentiate among them.

Char Device Driver

- **Major and minor numbers**
  - Both major and minor number are restricted between 0 and 255 in the version 2.4 kernel.
  - Numbers 0 and 255 are reserved for future uses.
  - ***mknod*** is used to create a device file, superuser privileges are required for this operation.

```
usage: mknod device type major minor
```

  - For example:

```
# mknod /dev/hello c 254 0
```

Char Device Driver

- **Register a char device driver**
  - Adding a new driver to the system means assigning a major number to it.
  - The assignment should be made at driver (module) initialization by calling the following function

```
int register_chrdev(unsigned int major, const char
        *name, struct file_operations *fops);
```

  - register_chrdev is defined in *<linux/fs.h>.*

Char Device Driver

- **Unregister a char device driver**
  - When a module is unloaded from the system, the major number must be released.
  - This is accomplished with the following function, which you call from the module's cleanup function

```
int unregister_chrdev(unsigned int major,
        const char *name);
```

  - The kernel compares the name to the registered name for the major number, if they differ, -EINVAL is returned.

Char Device Driver

- **File Operations**
  - An **open device** is identified internally by a ***file structure***, and the kernel uses the ***file_operations*** structure to access the driver's functions.
  - The *file_operations* structure, defined in *<linux/fs.h>*, is an array of function pointers.
  - **Each field in the structure must point to the function in the driver that implements a specific operation**, or be left *NULL* for unsupported operations.

Char Device Driver

- **File Operations**
  - Operations appear in struct *file_operations:*

```
  loff_t (*llseek) (…);
• ssize_t (*read) (…);
• ssize_t (*write) (…);
  int (*readdir) (…);
  unsigned int (*poll) (…);
• int (*ioctl) (…);
  int (*mmap) (…);
• int (*open) (…);
  int (*flush) (…);
• int (*release) (…);
  int (*fsync) (…);
  int (*fasync) (…);
  int (*lock) (…);
  ssize_t (*readv) (…);
  ssize_t (*writev) (…);
```

Char Device Driver

- **File Operations**
  - You can implement only the most important device methods, and use the tagged format to declare the *file_operations* structure.

```
struct file_operations my_fops = {
    owner: THIS_MODULE,
    read: my_read,
    write: my_write,
    ioctl: my_ioctl,
    open: my_open,
    release: my_release,
};
```

Char Device Driver

- **Open Method**
  - The ***open*** method is provided for a driver to do any initialization in preparation for later operations.
  - In most drivers, *open* should perform the following tasks:
    - Increment the usage count *(Not necessary in kernel 2.6)*
    - **Check for device-specific errors**
    - **Initialize the device**, if it is being opened for the first time
    - **Identify the minor number** and update the f_op pointer, if necessary
    - Allocate and fill any data structure to be put in filp->private_data

Char Device Driver

- **Release Method**
  - The role of the ***release*** method is the reverse of *open*.
  - *Release* method should perform the following tasks:
    - **Deallocate anything that *open* allocated** in filp->private_data
    - **Shut down the device** on last close
    - Decrement the usage count *(Not necessary in kernel 2.6)*

Char Device Driver

- **Read and Write Method**
  - The ***read*** and ***write*** methods perform a similar task, that is, **copying data from and to** application code.
  - Their prototypes are pretty similar

```
ssize_t read(struct file *filp, char *buff, size_t
        count, loff_t *offp);
ssize_t write(struct file *filp, const char *buff,
        size_t count, loff_t *offp);
```

  - ***filp*** is the file pointer and ***count*** is the size of the requested data transfer.
  - The ***buff*** points to the user buffer holding the data to be written or the empty buffer where the newly read data should be placed.
  - ***offp*** is a pointer to a "long offset type" object that indicates the file position the user is accessing.

Char Device Driver

- **Data transfer between kernel space and user space**
  - *Cross-space* data transfer is accomplished by the following two functions that are defined in <asm/uaccess.h>

```
unsigned long copy_to_user(void *to,
                    const void *from, unsigned long count);
unsigned long copy_from_user(void *to,
                    const void *from, unsigned long count);
```

  - Where,
    - to → the destination address
    - from → the source address
    - count → data count
  - Note that,
    - **Read method** is used to transfer data from kernel to user → **copy_to_user().**
    - **Write method** is used to transfer data from user to kernel → **copy_from_user().**

Char Device Driver

- **A real world example:**
- *7-seg driver for Netstart S3C4510B EV-Board.*
  - **Functionalities**
    - Major number: 120
    - Init: Register a device and initialize it
    - Cleanup: Unregister a device
    - Open: Increase use count
    - Close: Decrease use count
    - Read: Get current value of 7-seg
    - Write: Set 7-seg to specific value

Char Device Driver

- **A real world example**
  - Module skeleton

```
#include <linux/config.h>
#include <linux/version.h>
#include <linux/module.h>
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/errno.h>
#include <linux/fs.h>
#include <asm/uaccess.h>
#include "NET-Start.h"

#define MAJOR_NUM 120

… sniped

module_init(_7seg_init);
module_exit(_7seg_cleanup);
```

Char Device Driver

- **A real world example**
  - Init function

```
static int   __init _7seg_init(void)
{
    int result;
    result = register_chrdev(MAJOR_NUM, "Netstart_7seg",
                                        &_7seg_fops);
    if(result < 0) {
        printk(KERN_WARNING "Can't get major %d\n",
                                        MAJOR_NUM);
        return result;
    }
    *(unsigned int *)EXTACON0 = rEXTACON0;
    *(unsigned int *)EXTACON1 = rEXTACON1;
    Set7Segment(8);
    printk("Netstart 7-Segment display driver initialized!\n");
    return 0;
}
```

Char Device Driver

- **A real world example**
  - Cleanup function and file_operation structure

```
static void  __exit _7seg_cleanup(void)
{
    unregister_chrdev(MAJOR_NUM, "Netstart_7seg");
    printk("Netstart 7-Segment display driver removed!\n");
}

struct file_operations _7seg_fops = {
    read:       _7seg_read,
    write:      _7seg_write,
    ioctl:         _7seg_ioctl,
    open:       _7seg_open,
    release:      _7seg_release,
    owner:     THIS_MODULE,
};
```

Char Device Driver

- **A real world example**
  - Open and Release method

```
int  _7seg_open(struct inode *inode, struct file *filp)
{
    MOD_INC_USE_COUNT;
    return 0;
}

Int  _7seg_release(struct inode *inode, struct file *filp)
{
    MOD_DEC_USE_COUNT;
    return 0;
}
```

Char Device Driver

- **A real world example**
  - Write method

```
ssize_t  _7seg_write(struct file *filp, const char *buff,
                              size_t count, loff_t *offp)
{
    int nbr;

    if(copy_from_user(&nbr, (void *)buff, sizeof(int)))
        return(-EINVAL);

    if(nbr >= 0 && nbr < 16)
        Set7Segment(nbr);
    else
        return -1;

    return 0;
}
```

Char Device Driver

- **A real world example**
  - Read Method

```
ssize_t  _7seg_read(struct file *filp, char *buff,
                         size_t count, loff_t *offp)
{
    int nbr;

    nbr = Get7Segment();

    if(copy_to_user((void *)buff, &nbr, sizeof(int)))
        return(-EINVAL);

    return 0;
}
```

Char Device Driver

- **A real world example**
  - Test Application

```
int main(void)
{
    int i, j;
    int fd = open("/dev/seg",O_RDWR);  ——→  Open method
    for(i = 0; i < 16; i++) {
        write(fd, &i, sizeof(int)); ——→  Write method
        sleep(1);
        read(fd, &j, sizeof(int)); ——→  Read method
        printf("Current value of 7-seg: %d\n", j);
    }

    close(fd); ——→  Release method
    return 0;
}
```

Getting Started Projects

1. **Setting up the Board**
   – Installing and starting the Intel Eclipse* C/C++ IDE, and connecting with the Galileo board.
2. **On-board LED blink (fixed frequency)**
   – No MRAA or UPM
3. **Analog Read and Digital Write**
   – Activate a digital output (buzzer) if the luminosity of the room is below a certain level.
   – Extend the functionality to continuously (PWM) control an LED to maintain a constant luminosity.
4. **Sockets**
   – Communicate two Galileo boards using sockets and exchanging temperature information.
5. **Simple Webserver**
   – Start a simple web server in the Galileo board and display a webpage.
6. **Sending Data to the Cloud**
   – Acquire temperature information and centralize it using the Intel IoT Analytics platform.
   – Extend the functionality by including customized alerts (email) when the temperature is above 30°C.

References

- **OPERATING SYSTEM CONCEPTS**, Slides. Silberschatz, Galvin and Gagne. Wiley 8th Edition, 2009.
- **C/C++ SOCKETS TUTORIAL.** Linux HowTos. Webpage: http://www.linuxhowtos.org/C_C++/socket.htm
- **INTRODUCTION TO SOCKET PROGRAMMING WITH C.** CPSC 441 Tutorial – January 18, 2012. TA: Maryam Elahi
- **NETWORKED APPLICATIONS: SOCKETS.** COS 461: Computer Networks Spring 2006. Webpage: Http://www.cs.princeton.edu/courses/archive/spring06/cos461/
- **LINUX DEVICE DRIVER.** Writing a real world device driver for Embedded Linux. Slides webpage: http://wenku.baidu.com/view/7157b91efc4ffe473368ab38.html?re=view

A Test Application on the Intel Galileo Board

(no mraa)

```
#include <iostream>
#include <unistd.h>
#include <stdio.h>
using namespace std;

int main()
{
          cout << "LED Flash start" << endl;
          FILE *LEDhandler = NULL;
          char *LEDvalue = "/sys/class/gpio/gpio3/value";

          for (int i=0; i<10; i++){
                    if( (LEDhandler = fopen(LEDvalue, "r+")) != NULL ){
                              fwrite("1", sizeof(char), 1, LEDhandler);
                              fclose(LEDhandler);
                    }
                    sleep(3);
                    if( (LEDhandler = fopen(LEDvalue, "r+")) != NULL ){
                              fwrite("0", sizeof(char), 1, LEDhandler);
                              fclose(LEDhandler);
                    }
                    sleep(3);
          }
          cout << "LED Flash end" << endl;
          return 0;
}
```