Operating Systems – IPC – Part – 2

Hussein Al-Rikabi
rhussein@mit.bme.hu

Budapest University of Technology and Economics (BME)
Department of Measurement and Information Systems (MIT)

IPC - (recap)

- basic forms of communication
  - Shared memory model
  - Message-passing
- Synchrony
  - Blocking/ Non-blocking
  - Direct / Indirect
- Communications in Networks
  - Sockets
  - RPC
- Examples
  - POSIX – PRAM model
  - Sockets – Server/Clint

Direct and asymmetric communications

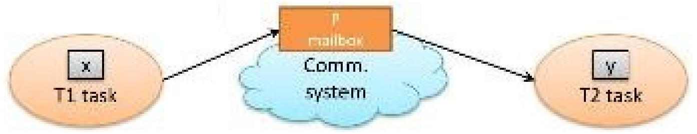
- Socket communication
- **Remote Procedure Call** (RPC)
  - Is another layer above the Socket communication.
  - Generally Calling a function in another task
    - ❑ The sender transfers the function name and arguments and receives the return value
    - ❑ The recipient performs the operations with the provided arguments and returns the results
    - ❑ Ex. In a **domain name lookup**: you pass a message to your DNS server, **looking** for the **IP address** of that domain name, the server does the look up for you and returns the IP required.

    Domain name : www.facebook.com ---- > 157.240.214.35
  - It is based on network communications, so it can be used between different machines
  - Besides the communication protocol, the data semantics also determined
    - Exact data types, structures
    - Automatic conversion may be performed between the participants
  - Higher level OS services based on RPC

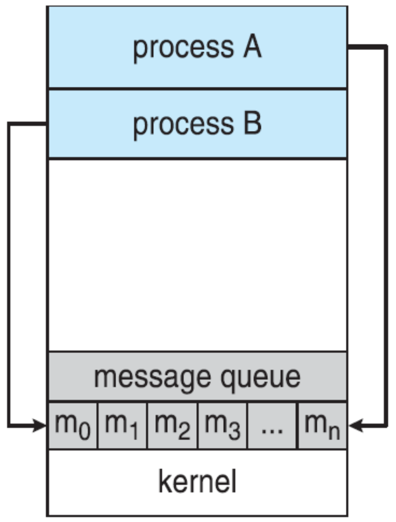
Remote Procedure Call (RPC)

- Distributed system infrastructure based on network communications, that provides:
  - High level communication between processes
  - Remote function calls in different processes (even on a different machine) and
  - even different languages (sending task: **C++**, Receiving task: **Python**) due to
  - Simple implementation due to common API-s
- Needed Structures
  - Communication infrastructures
    - Protocols for transferring functions calls
    - Addressing for identification of the processes and functions
    - Format of the data (standard, independent of the HW, the OS and the implementation)
    - Portmapper: assigning process ID-s and ports
  - RPC language: the description of the procedures and data types
    - Procedure names, argument list (types) and return types

Indirect communication solutions



- Mailbox
  - Problem: It can contain a limited number (may be just one) of messages
  - Also the size of a message is determined by the system
  - The mailbox has a direct address (which is not recipient address)
- Message queue
  - It can store and transfer infinite number of messages (of course is limited in the reality)
  - It may operate between different machines also, via network
  - It can also filter the messages
  - The message queue has a direct address (which is not recipient address)
  - Can be an extension of Mailbox; like FIFO queue. It is **used** when you want to preserve the order of messages.



Interprocess communication in practice

UNIX signals

- Recipient: a process or a process group
  - Notification about events in kernel, other processes, or in itself (e.g. errors)
- There are many types of signals (SIGINT, SIGCHLD, SIGKILL, ...see: `kill -l`)
  - User: ctrl+c, ctrl+z
- Overview of the operation
  - Generating a signal (initiated by an interrupt, system call)
  - The kernel notifies the recipients about the signal
  - The recipient **handles** the signal in a procedure: it may ignore it, terminates, starts further processing, etc. (not all signals can be ignored)
- Problems
  - Generation and delivery of the signal is delayed

Generating and handling signals

- Generating signals (by a process)

```
#include <signal.h>         /* kill() */
kill(pid, SIGSTOP );         /* sending signal */
```

- Handling signals
  - Multiple handlers are possible by default:
    - Term: terminate (exit())
    - Ign: ignore
    - Stop: suspend
    - Cont: continue from suspend
  - The application may register custom handlers (by passing a function pointer)

```
signal(SIGALM, alarm); //register the handler
void alarm(int signum){…} //the handler function
```

  - The possibilities are limited by the type of the signal
    - E.g. SIGKILL cannot ignored, and custom handler cannot registered

UNIX signals: examples

```
#include <signal.h>        /* signal(), kill() */
#include <unistd.h>                 /* getpid() */
#include <sys/types.h>              /* pid_t */
pid_t pid = getpid();               /* own PID */

kill(pid, SIGSTOP);        /* sending STOP signal */
```

(Equivalent terminal command: `kill -STOP <PID>`)

```
signal(SIGCLD, SIG_IGN);  /* children ignored */

signal(SIGINT, SIG_IGN);  /* ctrl+c signal ignored */

signal(SIGINT, SIG_DFL);  /* registering default handler */

signal(SIGALRM, myalarm); /* registering custom handler */
void myalarm(int signum) { ... }  /* the handler */
alarm(30);                 /* sending alarm after 30 seconds */
```

man -s 7 signal (part)

Signal	Value	Action	Comment
SIGHUP	1	Term	Hangup detected on controlling terminal or death of controlling process
SIGINT	2	Term	Interrupt from keyboard
SIGQUIT	3	Core	Quit from keyboard
SIGILL	4	Core	Illegal Instruction
SIGABRT	6	Core	Abort signal from abort(3)
SIGFPE	8	Core	Floating point exception
SIGKILL	9	Term	Kill signal
SIGSEGV	11	Core	Invalid memory reference
SIGPIPE	13	Term	Broken pipe: write to pipe with no readers
SIGALRM	14	Term	Timer signal from alarm(2)
SIGTERM	15	Term	Termination signal
SIGUSR1	30,10,16	Term	User-defined signal 1
SIGUSR2	31,12,17	Term	User-defined signal 2
SIGCHLD	20,17,18	Ign	Child stopped or terminated
SIGCONT	19,18,25	Cont	Continue if stopped
SIGSTOP	17,19,23	Stop	Stop process
SIGTSTP	18,20,24	Stop	Stop typed at tty
SIGTTIN	21,21,26	Stop	tty input for background process
SIGTTOU	22,22,27	Stop	tty output for background process

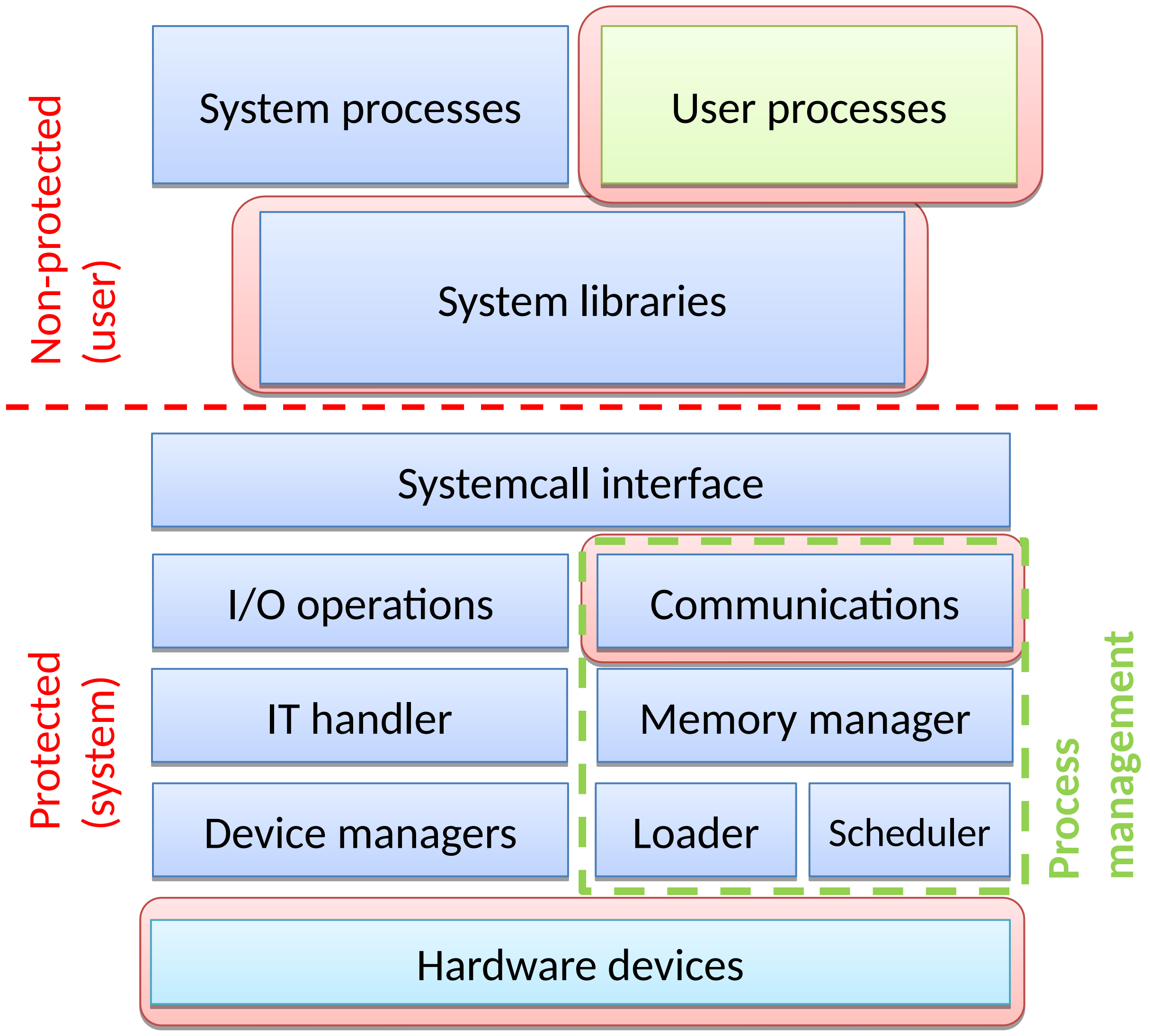
End of IPC Lecture

Operating Systems – Synchronization between tasks

Hussein Al-Rikabi
rhussein@mit.bme.hu

Budapest University of Technology and Economics (BME)
Department of Measurement and Information Systems (MIT)

The main blocks of the OS and the kernel (recap)



Parallel job execution

- The basic goal of the OS is to support user job execution
  - Jobs are executed by tasks (maybe more than one)
  - Executing jobs may require the executor tasks to **cooperate**
  - A modern OS is multiprogrammed ▭ executing jobs parallel
- Task implementations: processes and threads
  - Threads in the same process are using shared memory ▭ **competitive** environment
  - Processes are separated
    - Communication has to be synchronized
    - Parallel running processes may have to compete for the resources
- Current systems require parallel programming
  - The clock frequency is almost reached its technological boundary
  - Multithreaded execution is the design principle ▭ Multicore CPUs
  - This also requires a new programming principle

Competition and cooperation between tasks

- Tasks may operate independently from each others
  - Not influencing each others operation
  - Asynchronous execution
  - This separation is made possible by the OS
  - The resources (CPU, RAM, HW devices) are used by more than one task ▭
    - Conflicts may appear
    - Execution **dependencies** created
  - These conflicts have to be solved by the OS

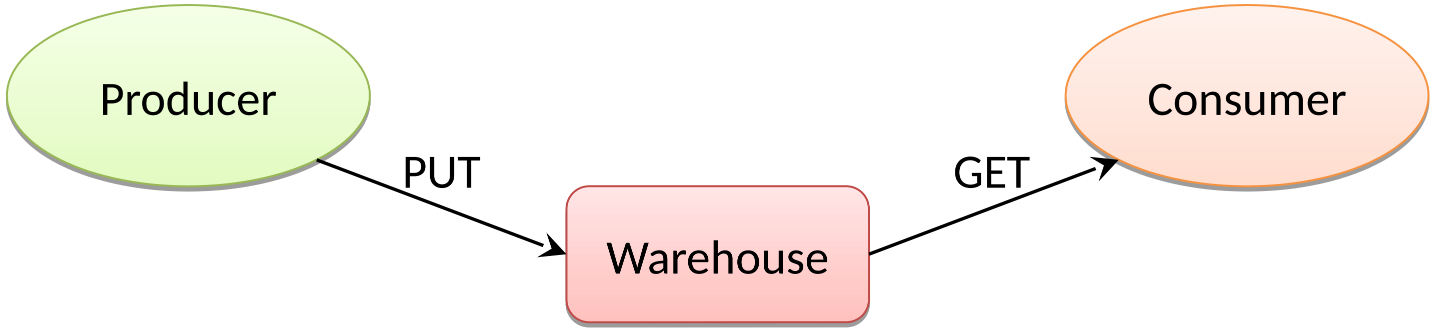
- The user jobs also require the tasks to cooperate
  - The job is decomposed to separate tasks
  - These tasks have to cooperate ▭ communicate and synchronize with each other
  - The OS provide services for this

- Remark: A single processor system is also a competitive environment that needs synch.

Simple example: producer – consumer problem

- The description of the problem
  - The producer creates a product which is stored in a warehouse (in a variable)
  - The consumer consumes the product from the warehouse
  - The producer and the consumer are working simultaneously
    - They may work separately in time
    - They may work with different rates
    - This can cause a problem.

- Problems to solve
  - Granting the consistency of the warehouse data structure
  - The consumer shouldn't check for products in an infinite loop
  - The producer shouldn't place new product in the warehouse while it's full



Synchronization between tasks

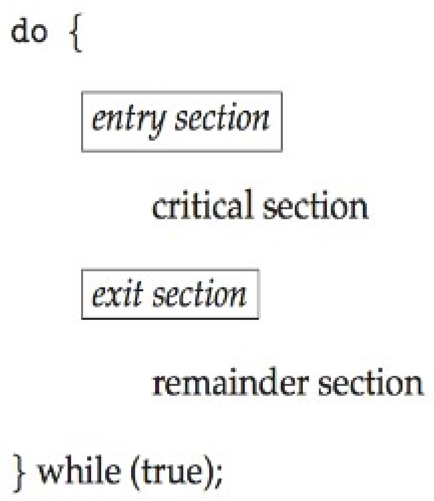
- Synchronization means coordination between tasks by constraining operation execution in time
  - The execution of specific tasks can be slowed down (temporally stopped) in order to achieve combined operation
- Basic application of synchronization
  - In competitive environments: using shared resources
  - In cooperation: communication
- The „price” of synchronization
  - It may cause performance degradation
  - The waiting tasks are not „useful”
    - They cannot wait for I/O operations, they are blocked
  - And if we have: **No sync**.: no waiting, erroneous behavior is possible
  - Bad sync. scheme: too much waiting, bad resource utilization

The basic forms of synchronization

- **Mutual exclusion**
  - **Critical section**: an instruction sequence of the tasks, which are cannot executed simultaneously
  - Shared resources are protected with this method, so **competitive situations** can be managed
  - Pessimistic method: locks the resource in every case
  - E.g.: while the printer is printing, cannot start a new print
- **Rendezvous**
  - Specific operations (part) of the tasks can start at the same time
  - **Cooperation** scheme to synchronize the operation of sub-tasks
  - E.g.: Send() and Receive() methods of direct (non buffered) messaging
- **Precedence**
  - The operations of the tasks should be executed in a predefined order
  - **Cooperation** scheme

The Critical-Section Problem

- The critical section is an instruction sequence with restricted execution: only one task can execute them at the same time
- Consider system of ***n*** processes {p_0, p_1, … p_{n-1}}
- Each process has **critical section** segment of code like:
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- ***Critical section problem*** is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**



```
do {
        entry section
            critical section
        exit section
            remainder section
} while (true);
```

The Critical-Section Problem

- The rules of the restriction
  - Entering
    - It is forbidden to enter, when another task is in the critical section
    - If no tasks in the critical section, only then that task can enter, which were executing it's own instructions before the critical section
  - Exiting
    - The critical section should finished in finite time
    - Common programming mistake: the tasks don't leave the critical section (releasing the resources)

Solution to Critical-Section Problem

1. **Mutual Exclusion** - If process ***P_i*** 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** - There exists a bound, or limit, on the number of times a task, can enter their critical sections

There are many solutions to the Synchronization problems (Consumer-Producer, Critical-section, Bounded-Buffer)

TBCed