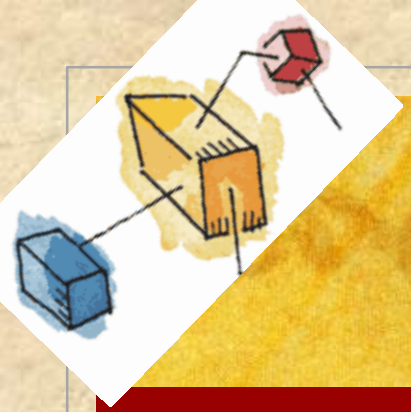
IT 540 Operating Systems ECE519 Advanced Operating Systems

Prof. Dr. Hasan Hüseyin BALIK

(5th Week)

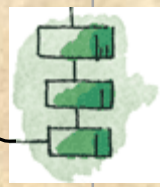
(Advanced) Operating Systems

5. Concurrency: Mutual Exclusion and Synchronization



5. Outline

- Principles of Concurrency
- Mutual Exclusion: Hardware Support
- Semaphores
- Monitors
- Message Passing



Multiple Processes

- Operating System design is concerned with the management of processes and threads:
  - Multiprogramming
  - Multiprocessing
  - Distributed Processing

Concurrency Arises in Three Different Contexts:

Multiple Applications

- invented to allow processing time to be shared among active applications

Structured Applications

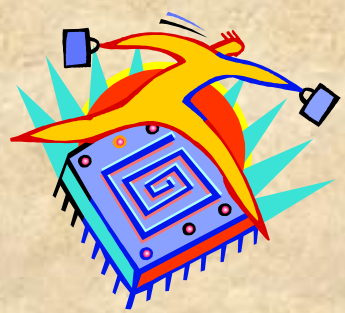
- extension of modular design and structured programming

Operating System Structure

- OS themselves implemented as a set of processes or threads

Principles of Concurrency

- Interleaving and overlapping
  - can be viewed as examples of concurrent processing
  - both present the same problems
- Uniprocessor – the relative speed of execution of processes cannot be predicted
  - depends on activities of other processes
  - the way the OS handles interrupts
  - scheduling policies of the OS



Difficulties of Concurrency

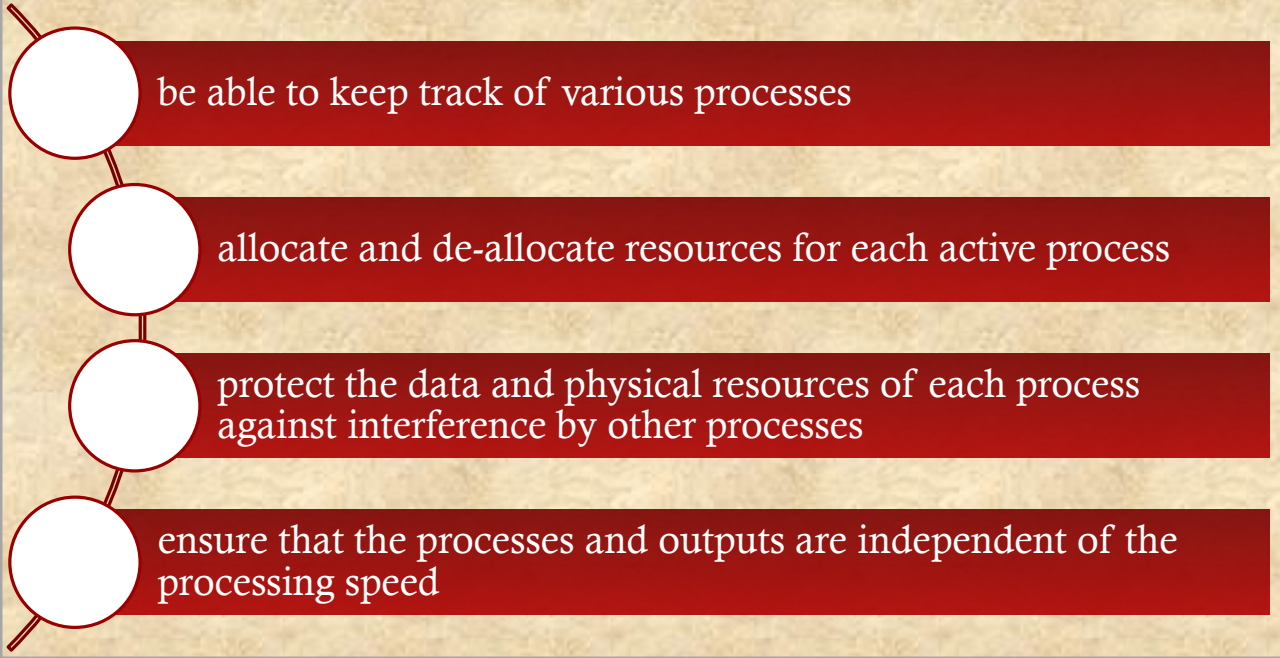
- Sharing of global resources
- Difficult for the OS to manage the allocation of resources optimally
- Difficult to locate programming errors as results are not deterministic and reproducible

Race Condition

- Occurs when multiple processes or threads read and write data items
- The final result depends on the order of execution
  - the "loser" of the race is the process that updates last and will determine the final value of the variable

Operating System Concerns

- Design and management issues raised by the existence of concurrency:
  - The OS must:



Resource Competition

- Concurrent processes come into conflict when they are competing for use of the same resource
  - for example: I/O devices, memory, processor time, clock

In the case of competing processes three control problems must be faced:

- **the need for mutual exclusion**
- **deadlock**
- **starvation**

Requirements for Mutual Exclusion

- Must be enforced
- A process that halts must do so without interfering with other processes
- No deadlock or starvation

- A process must not be denied access to a critical section when there is no other process using it
- No assumptions are made about relative process speeds or number of processes
- A process remains inside its critical section for a finite time only

Mutual Exclusion: Hardware Support

- **Interrupt Disabling**
  - uniprocessor system
  - disabling interrupts guarantees mutual exclusion
- **Disadvantages:**
  - the efficiency of execution could be noticeably degraded
  - this approach will not work in a multiprocessor architecture

Mutual Exclusion: Hardware Support

- Compare&Swap Instruction
  - also called a "compare and exchange instruction"
  - a **compare** is made between a memory value and a test value
  - if the values are the same a **swap** occurs
  - carried out atomically

Special Machine Instruction: Advantages

- Applicable to any number of processes on either a single processor or multiple processors sharing main memory
- Simple and easy to verify
- It can be used to support multiple critical sections; each critical section can be defined by its own variable

Special Machine Instruction: Disadvantages

- Busy-waiting is employed, thus while a process is waiting for access to a critical section it continues to consume processor time
- Starvation is possible when a process leaves a critical section and more than one process is waiting
- Deadlock is possible

Common Concurrency Mechanisms

`Semaphore`	An integer value used for signaling among processes. Only three operations may be performed on a semaphore, all of which are atomic: initialize, decrement, and increment. The decrement operation may result in the blocking of a process, and the increment operation may result in the unblocking of a process. Also known as a **counting semaphore** or a **general semaphore**
Binary Semaphore	A semaphore that takes on only the values 0 and 1.
Mutex	Similar to a binary semaphore. A key difference between the two is that the process that locks the mutex (sets the value to zero) must be the one to unlock it (sets the value to 1).
Condition Variable	A data type that is used to block a process or thread until a particular condition is true.
Monitor	A programming language construct that encapsulates variables, access procedures and initialization code within an abstract data type. The monitor's variable may only be accessed via its access procedures and only one process may be actively accessing the monitor at any one time. The access procedures are *critical sections*. A monitor may have a queue of processes that are waiting to access it.
Event Flags	A memory word used as a synchronization mechanism. Application code may associate a different event with each bit in a flag. A thread can wait for either a single event or a combination of events by checking one or multiple bits in the corresponding flag. The thread is blocked until all of the required bits are set (AND) or until at least one of the bits is set (OR).
Mailboxes/Messages	A means for two processes to exchange information and that may be used for synchronization.
Spinlocks	Mutual exclusion mechanism in which a process executes in an infinite loop waiting for the value of a lock variable to indicate availability.

Semaphore

A variable that has an integer value upon which only three operations are defined:

- There is no way to inspect or manipulate semaphores other than these three operations

1) May be initialized to a nonnegative integer value
2) The semWait operation decrements the value
3) The semSignal operation increments the value

Consequences

There is no way to know before a process decrements a semaphore whether it will block or not

There is no way to know which process will continue immediately on a uniprocessor system when two processes are running concurrently

You don't know whether another process is waiting so the number of unblocked processes may be zero or one

Strong/Weak Semaphores

😐 A queue is used to hold processes waiting on the semaphore

Strong Semaphores

- the process that has been blocked the longest is released from the queue first (FIFO)

Weak Semaphores

- the order in which processes are removed from the queue is not specified

Monitors

- Programming language construct that provides equivalent functionality to that of semaphores and is easier to control
- Implemented in a number of programming languages
  - including Concurrent Pascal, Pascal-Plus, Modula-2, Modula-3, and Java
- Has also been implemented as a program library
- Software module consisting of one or more procedures, an initialization sequence, and local data

Monitor Characteristics

Local data variables are accessible only by the monitor's procedures and not by any external procedure

Process enters monitor by invoking one of its procedures

Only one process may be executing in the monitor at a time

Synchronization

- Achieved by the use of **condition variables** that are contained within the monitor and accessible only within the monitor
  - Condition variables are operated on by two functions:
    - cwait(c): suspend execution of the calling process on condition c
    - csignal(c): resume execution of some process blocked after a cwait on the same condition

Message Passing

- When processes interact with one another two fundamental requirements must be satisfied:

synchronization	communication
• to enforce mutual exclusion	• to exchange information

- Message Passing is one approach to providing both of these functions
  - works with distributed systems *and* shared memory multiprocessor and uniprocessor systems

Message Passing

- The actual function is normally provided in the form of a pair of primitives:

  send (destination, message)

  receive (source, message)

- A process sends information in the form of a *message* to another process designated by a *destination*
- A process receives information by executing the `receive` primitive, indicating the *source* and the *message*

Synchronization
- Send
  - blocking
  - nonblocking
- Receive
  - blocking
  - nonblocking
  - test for arrival

Addressing
- Direct
  - send
  - receive
    - explicit
    - implicit
- Indirect
  - static
  - dynamic
  - ownership

Format
- Content
- Length
  - fixed
  - variable

Queueing Discipline
- FIFO
- Priority

Design Characteristics of Message Systems for Interprocess Communication and Synchronization

Synchronization

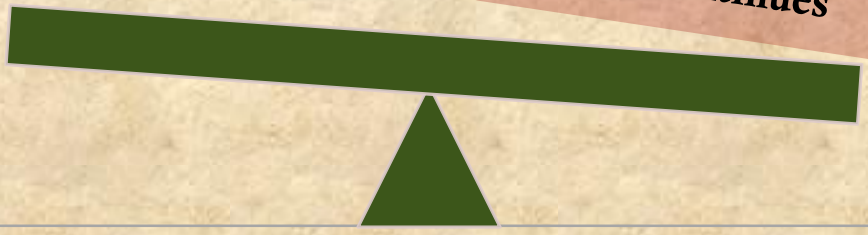
Communication of a message between two processes implies synchronization between the two

the receiver cannot receive a message until it has been sent by another process

When a receive primitive is executed in a process there are two possibilities:

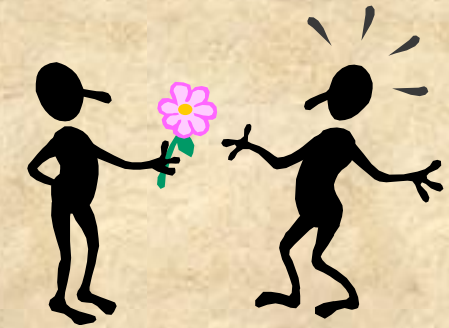
if there is no waiting message the process is blocked until a message arrives or the process continues to execute, abandoning the attempt to receive

if a message has previously been sent the message is received and execution continues



Blocking Send, Blocking Receive

- Both sender and receiver are blocked until the message is delivered
- Sometimes referred to as a *rendezvous*
- Allows for tight synchronization between processes



Nonblocking Send

Nonblocking send, blocking receive

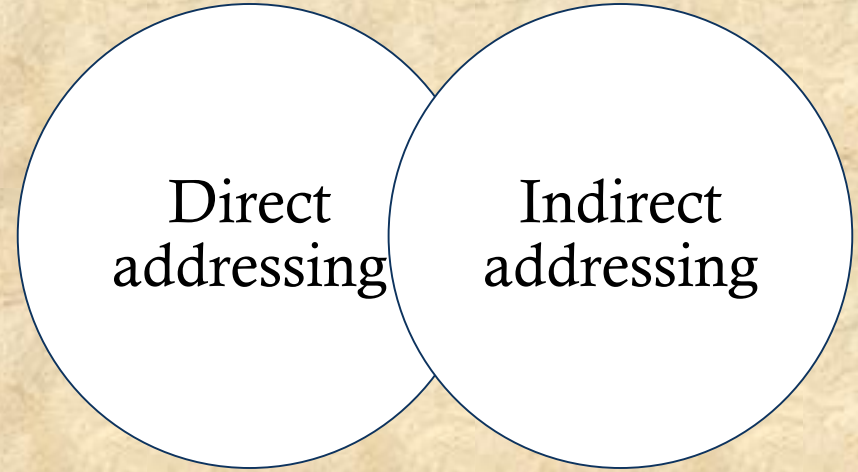
- sender continues on but receiver is blocked until the requested message arrives
- most useful combination
- sends one or more messages to a variety of destinations as quickly as possible
- example -- a service process that exists to provide a service or resource to other processes

Nonblocking send, nonblocking receive

- neither party is required to wait

Addressing

- Schemes for specifying processes in `send` and `receive` primitives fall into two categories:



Direct Addressing

- Send primitive includes a specific identifier of the destination process
- Receive primitive can be handled in one of two ways:
  - require that the process explicitly designate a sending process
    - effective for cooperating concurrent processes
  - implicit addressing
    - source parameter of the receive primitive possesses a value returned when the receive operation has been performed

Indirect Addressing

Messages are sent to a shared data structure consisting of queues that can temporarily hold messages

Queues are referred to as *mailboxes*

One process sends a message to the mailbox and the other process picks up the message from the mailbox

Allows for greater flexibility in the use of messages

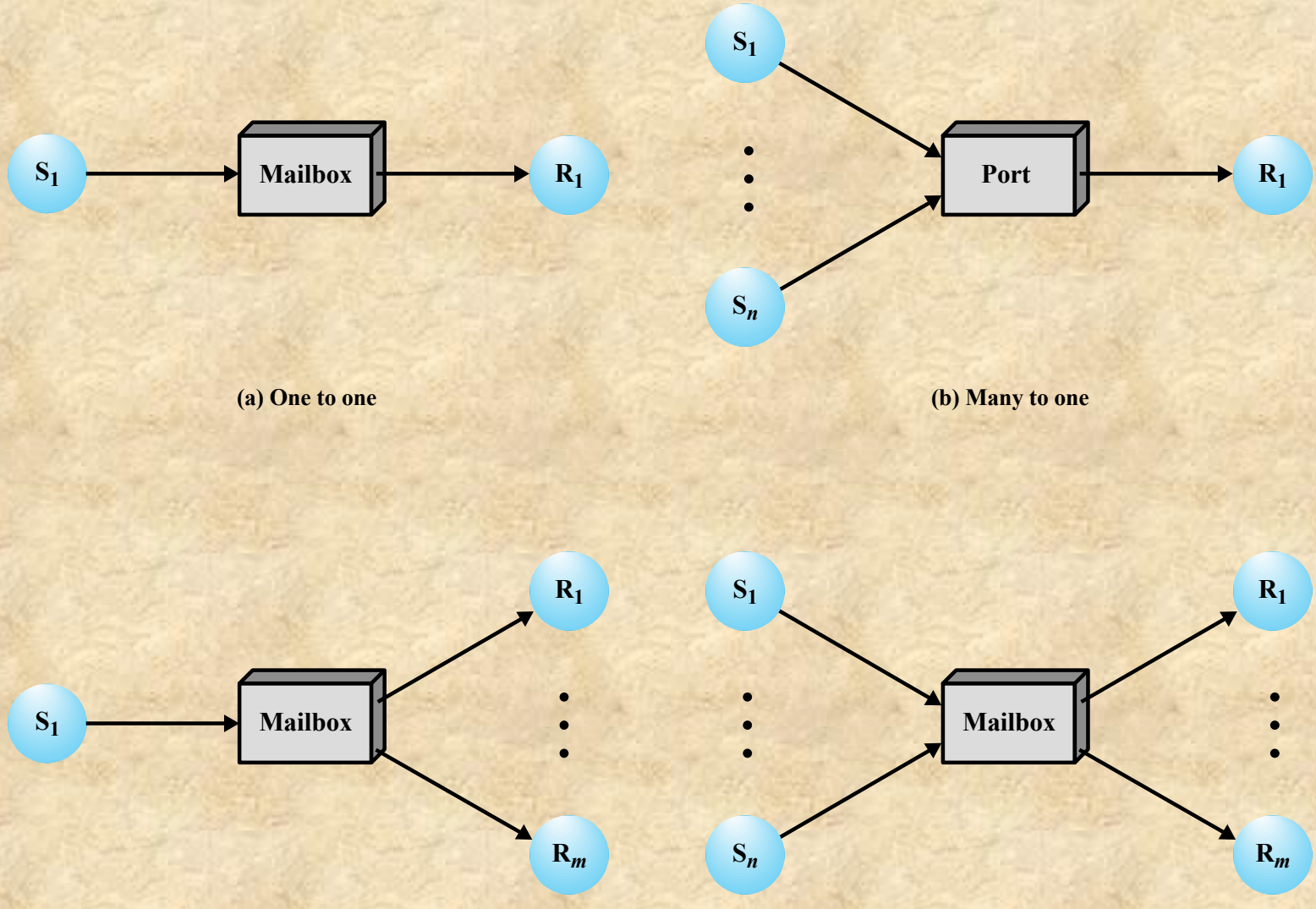


Figure 5.18 Indirect Process Communication

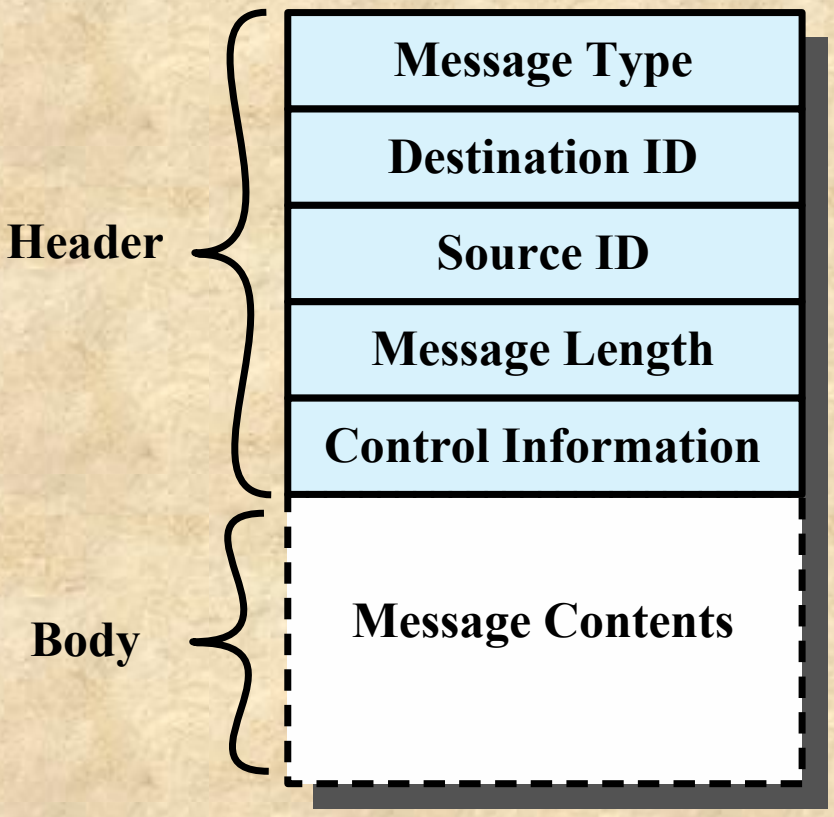


Figure 5.19 General Message Format