Subject : Operating System2
Fourth Class
Computer Science Dept.
College of Computer Science and Mathematics
2023-2024

Lecturer: Dr. Nadia Tarik
Email: nadia_tarik@uomosul.edu.iq

Course Syllabus:

1- Introduction to process.
2- Synchronization problem, The Critical-Section Problem, Synchronization Examples, Peterson's Solution, Hardware Semaphore.
3- Monitors Deadlock Characterization, Methods for Handling Deadlocks, Deadlock Prevention, Deadlock Avoidance, Deadlock Detection, Recovery From Deadlock
4- Introduction to Memory, Management Swapping and Contiguous Memory Allocation, Paging and Structure of the Page Table, Segmentation.
5- Overview of Mass-Storage, RAID Structure, Stable-Storage Implementation, Disk Structure and Disk Attachment, Disk Scheduling, Disk Management.

Reference

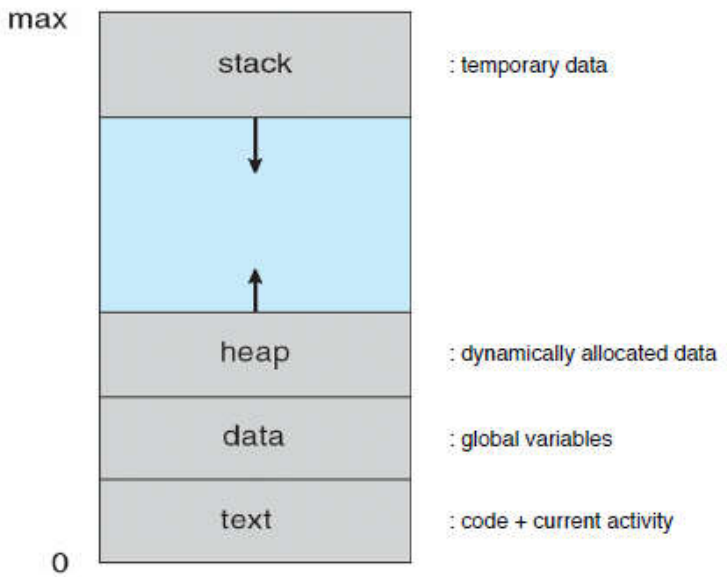
A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 9th ed., USA: John Wiley & Sons, Inc., 2013.

- **Process** – a program in execution; process execution must progress in sequential fashion.
- A process willneed certain resources-such as CPU time, memory, files, and I/O devices-to accomplish its task. These resources are allocated to the process either when it is created or while it is executing.
- Program is ***passive*** entity stored on disk (**executable file**), process is ***active.***
- A computer program is usually written by a computer programmer in a programming language as a text file.
- Program becomes process when executable file loaded into memory.
- When we compare a program with a process, we can conclude that a process is a dynamic instance of a computer program.
- Execution of program started via GUI mouse clicks, command line entry of its name, etc.
- One program can be several processes.
- A process is an instance of a running program; it can be assigned to, and execute on, a processor.

❖ Process Parts

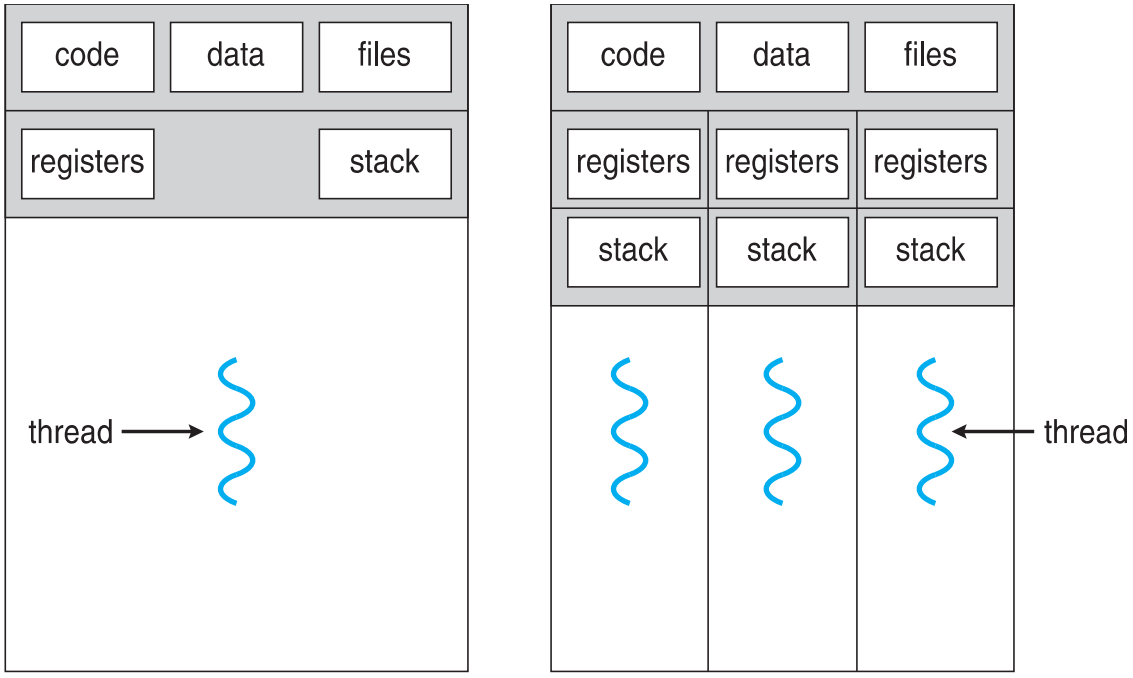
- **Process** has multiple parts:
  1. The program code, also called **text section** (stored on a non-volatile storage).
  2. Current activity including **program counter**, **processor registers**.
  3. **Stack** containing temporary data (Function parameters, return addresses, local variables)
  4. **Data section** containing global variables.
  5. **Heap** containing memory dynamically allocated during run time (and is managed via calls to *new*, *delete*, *malloc*, *free*, etc.)

❖ Process in Memory



❖ Threads

- A thread is an execution unit that is part of a process. A process can have multiple threads, all executing at the same time (multiple program counters per process). It helps to improve the application performance using parallelism. The process of a single thread allows the process to perform only one task at atime.
- Most modern operating systems have extended the process concept to allow a process to have multiple threads of execution and thus to perform more than one task at a time. This feature is useful in multicore/ multiprocessor systems, where multiple threads can run in parallel.
- There are many advantage of threads such as resource sharing, responsiveness, economy, and utilization of multiprocessor architecture.



single-threaded process multithreaded process

❖ Process synchronization

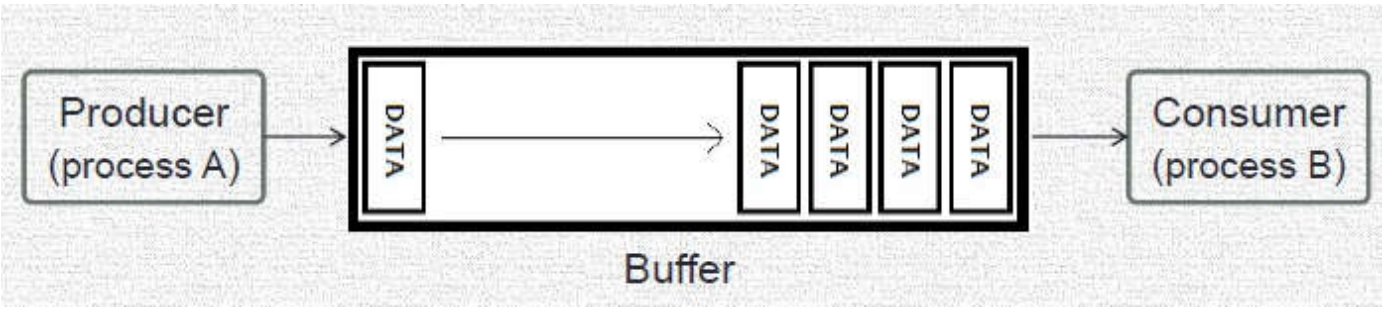
- Processes can execute concurrently in the operating system.
- These processes may be either independent processes or cooperating processes.
- A process is ***independent*** if it cannot affect or be affected by the other processes executing in the system. Any process that does not share data with any other process is independent.
- A process is ***cooperating*** if it can affect or be affected by the other processes executing in the system. Clearly, any process that shares data with other processes is a cooperating process.

- Cooperating processes can either directly share a logical address space (i.e., both code and data) or be allowed to share data only through files or messages.
- Concurrent access to shared data may result in data inconsistency.
- The problem of process synchronization arises on a cooperative process only.

❖ **Producer-Consumer Problem**

Suppose that we want to provide a solution to the consumer-producer problem or (bounded buffer problem). We want to ensure that the **Producer** should not add DATA when the Buffer is full and the **Consumer** should not take DATA when the Buffer is empty.

We can do so by having an integer **counter** that keeps track of the number of full buffers. Initially, **counter** is set to **0**. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.



❖ **Producer process**

```
while (true) {
   /* produce an item in next produced */
   while (counter == buffer_size);      /* Do nothing */
   buffer [in] = next_produced;
   in = (in+1) % BUFFER_SIZE;
   counter ++;
               }
```

❖ **Consumer process**

```
while (true)  {
   while (counter == 0);          /* Do nothing */
   next_consumed = buffer [out];
   out = (out+1) % BUFFER_SIZE;
   counter --;
   /* consume the item in next consumed */
}
```

- **in** variable is used by producer to identify the next empty slot in **buffer**.
- **out** variable is used by the consumer to identify where the consumed item is.
- **counter** is used by both processes to identify the number of filled slots in the **buffer**.
- The shared resources are: **buffer** and **counter**.

Although the producer and consumer routines shown above are correct separately, they may not function correctly when executed concurrently. As an illustration, suppose that the value of the variable counter is currently **5** and that the producer and consumer processes concurrently execute the statements "counter++" and "counter--".

After executing these two statements, the value of the variable counter may be 4, 5, or 6! The only correct result is counter == 5, which is generated correctly if the producer and consumer execute separately. The value of counter may be incorrect as shown:

counter++ could be implemented in machine language as follows:
register1 = counter
register1 = register1 + 1
counter = register1

counter-- could be implemented in machine language as follows:
register2 = counter
register2 = register2 – 1
counter = register2
where **register1** and **register2** are local CPU registers.

Consider this execution interleaving with "count = 5" initially:

S0: producer executes **register1 = counter**	{register1 = 5}
S1: producer executes **register1 = register1 + 1**	{register1 = 6}
S2: consumer executes **register2 = counter**	{register2 = 5}
S3: consumer executes **register2 = register2 – 1**	{register2 = 4}
S4: producer executes **counter = register1**	{counter = 6}
S5: consumer executes **counter = register2**	{counter = 4}

The incorrect state "counter == 4", indicates that four buffers are full, when in fact, five buffers are full. Reversing the order of the statements at S4 and S5, we get the incorrect state "counter == 6". This is because both processes are allowed to manipulate the variable counter concurrently. A situation where *several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called* a **race condition**. To guard against the race condition we need only one process at a time to manipulate the variable counter, hence processes must be synchronized in some way.