Computer System Software Course
CE 403

Dr. Mohamed Abd El-Latif Mowad
Lectures -3

Processes and Communication

and synchronization.

Dr. Mohamed Abd El-Latif Mowad

Email: Turbocode2000@yahoo.com

Cell Phone: 01007532006
Process: is a program under execution.

- A user may be able to run several programs at one time such as word-web browser – an E-mail package. Even if the user can execute a program the O-S may need to support its own internal programmed activates, such as memory management. We call all of the previous programs processes.

- A process is more general than the program code, which is sometimes is called the text section.

- process also includes the current activity, as represented by the value of the program counter and the contents of the processor registers.
A process in RAM includes

1- **Program Counter:**
   specifying the next instruction to be executed. Or the register that holds a pointer to the next instruction to be executed.

2- **Stack:**
   Containing temporary data such as:
   (function parameters – Local variables).

3- **Data Section:** (containing global variables).

   It is also include a Heap, which is the memory that is dynamically allocated during process run time.

4- **Text Section / Code Section** or text - code segments
   Is the program code itself.
Stack

LIFO stacks are the simplest way of saving information in a temporary storage location.
• When a **program begins executing** in the function `main()`, **space is allocated** on the stack for all variables declared within `main()`, as seen in Figure -1(a).

• **If main() calls a function, func1(),** additional storage is allocated for the variables in `func1()` at the top of the stack as shown in Figure -1 (b).

• **When func1() returns, storage for its local variables is deallocated,** and the top of the stack returns to position shown in Figure -1(c)
for nEN = 1:length(EbNOdb)
    en = 10^(EbNOdb(nEN)/10);    % convert Eb/NO from unit db to normal numbers
    L_c = 4*a*en*rate;            % reliability value of the channel
    sigma = 1/sqrt(2*rate*en);   % standard deviation of AWGN noise

    % Clear bit error counter and frame error counter
    errs(nEN,1:niter) = zeros(1,niter);
    nferr(nEN,1:niter) = zeros(1,niter);

    nframe = 0;                   % clear counter of transmitted frames

    while nferr(nEN,niter)<ferrlim
        nframe = nframe + 1;

        x = round(rand(1, L_total-m)); % info. bits
        [temp, alpha] = sort(rand(1,L_total)); % random interleaver

        en_output = encoderm(x, g, alpha, puncture); % encoder output (+1/-1)
function en_output = encoderm( x, g, alpha, puncture )

% Copyright Nov. 1998 Yufei Wu
% MPRG lab, Virginia Tech.
% for academic use only

% uses interleaver map 'alpha'
% if puncture = 1, unpunctured, produces a rate 1/3 output of fixed length
% if puncture = 0, punctured, produces a rate 1/2 output
% multiplexer chooses odd check bits from RSC1
% and even check bits from RSC2

% determine the constraint length (K), memory (m)
% and number of information bits plus tail bits.

[n,K] = size(g);
m = K - 1;
L_info = length(x);
L_total = L_info + m;

% generate the codeword corresponding to the 1st RSC coder
% end = 1, perfectly terminated;
input = x;
output1 = rsc_encode(g,input,1);
Definitions

**Heap** is an area of
(1) **Dynamically-allocated memory** that is
(2) **Managed automatically** by the **operating system**.

The **heap** segment provides more stable storage of data for a program;
(3) **Memory allocated in the heap remains in existence for the duration of a program**.
Therefore,
(4) **Global variables are allocated on the heap**.
(5) **No limit in memory size**.
Definitions

Stack:

(1) Grows and shrinks as functions push and pop local variables.

(2) There is no need to manage the memory yourself, variables are allocated and freed automatically.

(3) Stack has size limits and have a fast access.

(4) Stack variables only exist while the function that created them, is running.
The structure of the process in memory is shown in the following figure.
• The program by itself is not a process, A program is a passive entity, such as a file containing a list of instructions stored on disk always called (Executable file).

• A process is Active entity, with the program counter specifying the next instruction to execute and a set of associated resources.

A program becomes a process when it is loaded into memory.
As a process be executed, it changes its state. The state of the process is defined by the current activity of the process.

Each process may be in one of the following state:

1- **New**: The process is being created.

2- **Running**: Instruction are being executed.

3- **Waiting**: The process is waiting for some event to occur such as (I/O completion or reception of signal)

4- **Ready**: The process is waiting to be assigned to the processor.

5- **Terminated**: The process has finished execution.
Process State Diagram (PSD)

- **New**
- **Ready**
- **Running**
- **Waiting**
- **Admitted**
- **Exit**

- **I/O or event completion**
- **Scheduler dispatch**
- **I/O or event wait**

- **Interrupt** (Time slice)

**States:**
- New
- Ready
- Running
- Waiting
- Terminated
New → Ready
Admitted to **ready queue**; can now be considered by CPU scheduler.

Ready → Running
CPU scheduler chooses that process to execute next, according to some scheduling algorithm

Running → Ready
Process has used up its current time slice (interrupted)

Running → waiting
Process is waiting for some event to occur (e.g. waiting for I/O operation to complete, ...etc.) for example save operation

waiting → Ready
I/O or event completion
Process Control Block (PCB) on the O-S

• Each process is represented in the O-S by a process control block (PCB) also called task control block.

• A (PCB) shown in the following figure contains many pieces of information associated with a specific process as follows:

1- **Process state** :
   The state may be (New -Ready –Running- waiting).

2- **Program Counter**:
   The counter indicates the address of the next instruction to be executed for this process.
Process Control Block (PCB)

- process state
- process number
- program counter
- registers
- memory limits
- list of open files
  
  
  • • •
3- **(CPU) Register**: The registers vary in number and type depending on the computer architecture. They include **Accumulators**, **Index registers**, **stack pointer**, and **general purpose registers**. Along with the **program counter**, this state information must be saved when an **interrupt occurs**, to allow the process to be continued correctly.

The following diagram shows the CPU switch from process to another process.
Accumulators:
- Are registers that can be used for arithmetic, logical, shift, or other similar operations.

Index register:
- Are used to provide more flexibility in addressing modes, allowing the programmer to create a memory address by combining the contents of an address register with the contents of an index register.

Stack pointer:
- The most common use is to store return addresses, processor state information, and temporary variables for subroutines.

General purpose registers:
- can be used as either data or address registers.
CPU Switch From Process to Another Process

- process state
- process number
- program counter
- registers
- memory limits
- list of open files

Diagram:

- Process $P_0$:
  - Executing
  - Interrupt or system call
    - Save state into PCB$_0$
    - Reload state from PCB$_1$
    - Idle
    - Executing

- Process $P_1$:
  - Idle
  - Executing
  - Interrupt or system call
    - Save state into PCB$_1$
    - Reload state from PCB$_0$
    - Idle
The objective of **Multi-programming** is to have **some** processes running at all times, to max CPU utilization.

The objective of **Time sharing** is to **switch CPU among processes so frequently that users can interact with each program while it is running**.

To meet this objectives, the **process scheduler** select an available process for program execution on the CPU.

For a **single processor** system, there will never be more than one running process. If there is more processes, the rest will have to wait until the CPU is free.
Scheduling Queues

- As processes enter the system, they are put into a **job queue**, which consists of all processes in the system.
- The processes that are residing in main memory and are ready and waiting to execute are kept on a list called **ready queue**.
- The system also includes other queues such as the set of **device queues** (I/O).
- After the process is allocated to the CPU one of the following events will be happened
1- The process could issue I/O request, and then be placed in an **I/O queue**.

2- The process could create a new sub-process and **wait for its termination**.

3- The process could be removed from the CPU, as a result of an interrupt, and be put back in the **ready queue**.
A process migrates among the various scheduling queues throughout its lifetime. The operating system must select processes from these queues in some fashion. The selection process is carried out by the appropriate scheduler.

In many cases more processes are spooled to a mass storage device (disk) where they are kept for later execution.

The long term scheduler:

Jobs which get resources (e.g. get allocated memory) and the chance to compete for cycles (be in the ready queue) it is invoked infrequently (seconds – minutes) may be slow. The long term scheduler controls the degree of multiprogramming.
The short term scheduler: or CPU scheduler:

Which of the ready jobs gets the next slice of CPU time. It is invoked very frequently (milliseconds) (must be fast)

Long and Short Term Scheduling
Inter-Process Communication (IPC)

- **Processes** executing concurrently in the operating system may be *either independent processes or cooperative processes*. A process is *independent process* 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 *cooperative* 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*.

- There is *several reasons* for providing an environment that allows process cooperation.
1-Information Sharing:
Since several users may be interested in the same piece of information (shared file) we must provide an environment to allow concurrent access to such information.

2-Computation Speed up:
If we want a particular task to run faster, we must break it into sub-tasks, each of which will be executing in parallel with the others. Notice that such a speed-up can be achieved only if the computer has multiple processing elements.
3-Convenience:

- Even an **individual user** may work on **many tasks at the same time**. For instance, a user may be **editing**, **printing**, and **compiling** in parallel.

- Cooperating processes require an interprocess communication (**IPC**) mechanism that will allow them to exchange data and information. There are **two fundamental modes** of (**IPC**).
1- **Shared Memory:**

- A **region of memory** that is by cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region.

2- **Message Passing:**

- Communication takes place by means of **message exchanged** between cooperating processes.
The two communication methods are shown in the following figure:

**Communication Methods**

(a) **Message passing**  (b) **Shared memory**
Message passing:
- Useful for exchanging smaller amount of data
- Typically implemented using system calls and thus require more time consuming task for Kernel intervention (slow method)

Shared memory:
- Allow max speed and convenience of communication as it can be done at memory speed,
- Shared memory is faster than message passing systems.
In contrast, in shared memory systems, system calls are required only to establish shared memory regions. Once shared memory is established, all accesses are treated as routine memory access, and no assistance from the Kernel is required.