Chapter 2

Using the Operating system

Resource Descriptors

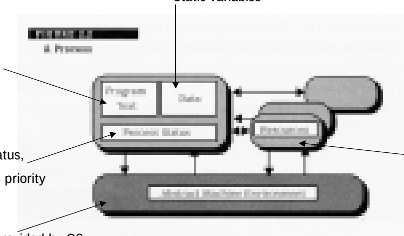
- The OS implements Abstraction of each of this
  - Unit of Computation is a ‘process’
  - Unit of information storage is a ‘file
- For each resource abstraction (file, memory, processor), OS maintains a resource descriptor
- Resource descriptor:
  - Identify resources
  - Current state
  - What process it is associated with, if it is allocated
  - Number and identity of available units

Resource Descriptors...

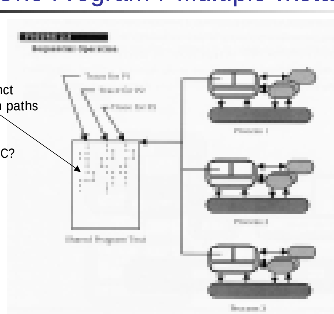
- File descriptor:
  - File name
  - File type (Sequential, Indexed, ...)
  - Owner
  - State (Open, Closed)
  - Extents (mapping to the physical storage)
- Process descriptor
  - Object program (Program text)
  - Data segment
  - Process Status Word (PSW) – executing, waiting, ready
  - Resources acquired

Process & Process Descriptor

Contents of a descriptor maps directly to the Abstract Machine provided by the OS



One Program / Multiple Instantiations



Note:

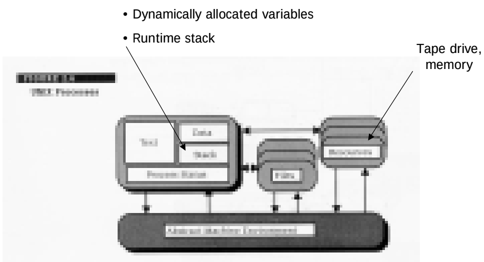
Each Process has its **own** descriptor
- text (shared), data...

Only **one** process active at a time (context switching)

Process

- 3 units of computations:
  - Process
  - Thread
  - Object
- Process: ‘heavy-weight’ process
  - OS overhead to **create and maintain descriptor** is expensive
- Thread: “light-weight” process
  - OS maintains minimal internal state information
- Objects: ‘heavy-weight’ process
  - Instantiation of a class

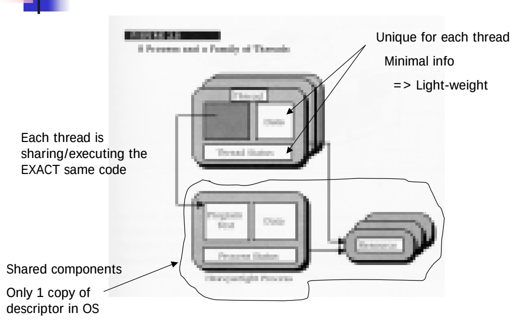
UNIX Processes



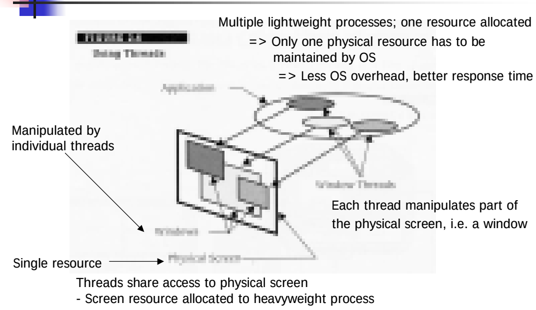
Thread

- Thread: light-weight process
  - OS maintains minimal internal state information
- Usually instantiated from a process
- Each thread has its OWN unique descriptor
  - Data, Thread Status Word (TSW)
- SHARES with the parent process (and other threads)
  - Program text
  - Resources
  - Parent process data segment

Thread ...



Threads... example



Objects

- Objects:
  - Derived from SIMULA '67
  - Defined by classes
  - Autonomous
- Classes
  - Abstract Data Types (ADT)
  - Private variables
- An instantiation of a class is an Object

Objects

- Objects are heavy-weight processes
  - have full descriptors
- Object communicate via Message passing
- OOP:
  - Appeals to intuition
  - Only recently viable
    - Overhead of instantiation and communication

Computational Environment

- When OS is started up
  - Machine abstraction created
    - Hides hardware from User and Application
  - Instantiates processes that serve as the user interface or “Shell”
    - Shell (UI) instantiates user processes
- Consider UNIX:

  UNIX → getty → shell → user process

- What are the advantages & disadvantages of so many processes just to execute a program ?

Advantages & Disadvantages

- Advantages...

  Each process (UNIX, getty, shell, ...) has its own ‘protected’ execution environment

  - If child process fails from fatal errors, no (minimal) impact on parent process
- Disadvantages...

  OS overhead in

  - Maintaining process status
  - Context switching

Process Creation – UNIX fork()

- Creates a child process that is a ‘**Thread**’
- Child process is duplicate (initially) of the parent process – except for the process id
- Shares access to all resources allocated at the time of instantiation and Text
- Has duplicate copy of data space BUT is its own copy and it can modify only its own copy

If a child Process requests / receives a resource, does the parent or other children have access to it ?

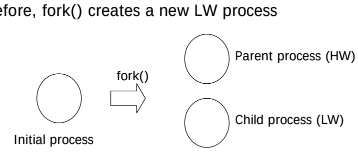
Process creation - fork()... example

```
int pidValue;
..
pidValue = fork();                /* creates a child process        */
If(pidValue == 0) {
        /* pidValue is ZERO for child, nonzero for parent           */
        /* The child executes this code concurrently with Parent    */
        childsPlay(..);           /* A locally-liked procedure      */
        exit(0);                  /* Terminate the child            */
}
/* The Parent executes this code concurrently with the child        */
..
wait(..);                /* Parent waits for Child’s to terminate   */
```

UNIX process creation : fork() facility

Process creation – Unix fork()...

- Child/Parent code executed based on the pid value in “local” data space
  - For parent process, pid value returned is that of the *child* (non-zero)
  - For child process, pid value returned is 0
- pidvalue returned to parent process is non-Zero
- Therefore, fork() creates a new LW process



Process Creation – Unix exec()

- Turns LW process into autonomous HW process
- fork()
  - Creates new process
- exec()
  - Brings in new program to be executed by that process
  - New text, data, stack, resources, PSW, etc.

    BUT using same (expanded) process descriptor entries

  In effect, the “exec’ed” code overlays “exec’ing” code

Process creation – exec()... example

```
int pid;
..
        /* Setup the argv array for the child      */
..
if((pid = fork()) == 0) {         /* Create a child                  */
        /* The child process executes changes to its own program    */
        execve( new_program.out , argv , 0 );
        /*Only return from an execve call if it fails               */
        printf("Error in execve");
        exit(0);                  /* Terminate the child            */
}
        /* Parent executes this code      */
..
wait(..);                /* Parent waits for Child's to terminate   */
```

UNIX process creation: exec() facility