A process is an operating system concept that captures the idea of a program in execution. A process has memory for its program and data, it has resources to compute with e.g. open files, and gets assigned a processor to do the computing. A process is always executing at some point in a program. When a given process is not running, since CPU is currently executing instructions of another process, the operating system keeps a copy of the hardware registers at the moment of interruption, program instructions, data and all other relevant information that describe a point in the computation that process is at. Thus, when this process should continue running all necessary information can be restored.

At any point in time, a processor executes some instruction, and the next instruction to be executed is either the next instruction in the flow of the user program (or operating system code) or if an interrupt happened, the next instruction to be executed is one with address that is loaded in PC during hardware processing of interrupt. Note that CPU can not distinguishes between instructions of an operating system and user programs.

The basic idea of the implementation of processes in an operating system rests on a few simple ideas:

1. the operating system is not a process but implements processes;
2. each process is represented by a data structure called a process control block (PCB) and PCBs are kept in the operating system memory;
3. Each process runs on the actual hardware with some restrictions:
   - CPU is shared between all processes and is switched between them, and the memory area of each process is restricted,
Process Implementation (cont.)

- A process is prevented from using some machine instructions since it runs in user mode,
- A process communicates with the operating system through system calls,
- A running process can be interrupted at any time by hardware (I/O) interrupts (from a disk controller or timer).

4. When an interrupt occurs (from an I/O controller or as result of those special interrupt causing instructions, such as Syscall in MIPS or Int n in Intel), the operating system gets control, handles the interrupt and then assigned CPU back to some process (not necessary to interrupted process).

Note: This is just a basic principle. We shall see later that parts of operating system may be mapped in process address space.

Process States

- As a process executes, it changes its state:
  - new: a process is being created.
  - running: instructions of a process are being executed.
  - waiting (blocked): a process is waiting for some event to occur.
  - ready: a process is waiting to be assigned to a CPU.
  - terminated: a process has finished execution.

CPU Switch From Process to Process

Process Control Block (PCB)

PCB contains information associated with a given process:
- Process state
- Process identification (Pid)
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information
- Pid of parent process
Unix/Linux Process Related System Calls

<table>
<thead>
<tr>
<th>System Call</th>
<th>Parameters</th>
<th>Returns</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>fork</td>
<td>none</td>
<td>Pid or 0</td>
<td>Duplicated process</td>
</tr>
<tr>
<td>exec</td>
<td>programFile, args</td>
<td>None???</td>
<td>Execute program in process</td>
</tr>
<tr>
<td>exit</td>
<td>return code</td>
<td>None</td>
<td>Destroy process</td>
</tr>
<tr>
<td>wait</td>
<td>return code address</td>
<td>pid</td>
<td>Wait for child process to exit</td>
</tr>
</tbody>
</table>

Unix Process Related System Calls (cont.)

```c
int exec(char * programName, char *argv[]);
```

The program `programName` is loaded in the calling process address space. Thus, the old program in the calling process is overwritten and will no longer exist. The arguments are in the argument vector `argv`, which is an array of strings, that is, an array of pointers to characters. (There are 6 versions of exec system call)

```c
void exit(int returnCode);
```

This system call causes a process to exit. The `returnCode` is returned to the parent process if it waits for its child process to terminate.

```c
int wait(int * returnCode);
```

This system call will cause the calling process to wait until any process created by the calling process exits. The return values is the process identifier of the process that exited. The return code is stored in `returnCode`.

Unix Processes: Example 1

```c
int exec(char * programName, char *argv[]);
```

Write C code for a process `A` which creates another process with code from `ProgB` file and then process `A` exits.

```c
ProgA
{
    int x; char *arg1={0};
    x=fork();
    if (x==0) execv("ProgB", arg1);
    exit(3);
}
```

If a child process does not terminate before its parent terminates, the `init` process "reparents" the orphan children. Note, the `init` process has Pid = 0.
Unix Processes: Example 2

Write C code for a process A which creates another process with code from ProgB file, then the process A waits for its child process to terminate, before it exits.

```c
ProgA
{   
    int x,y,z; char *arg[1]={0};
    x=fork();
    if (x==0) execv("ProgB",arg);
    y=wait(&z);
    exit(3);
}
```

TARGET

ProgB
{   
    exit(5); /*point A*/
    exit(1); /*point B*/
}

If the child process exits at point A, then z=5, while if it exits at point B then z=1.

Unix Processes: Example 3 (Cse660/Unix.ls.c)

Similar to the program in Figure 3.10 and it illustrates how Unix command " ls –l –a " can be issued from a program.

```c
void main ()
{   
    int pid;
    pid =fork();
    if (pid<0) { printf("fork error"); exit (0); }
    if (pid==0)
    { execlp("/bin/ls", "ls", "-l", "-a", 0);
      printf("exec failed");
      exit(0);
    }
    wait();
    printf("nDone");
    exit(0);
}
```

Unix Processes: Example 4 (Cse660/Calling.co2.c)

Process A creates another process with code from file "co2" with a list of 4 parameters passed. Then the process A doesn’t wait for its child process to terminate, before it exits.

```c
void main()
{   
    int x;
    char *argv[]={"ccc", "ooo", "1234",0};
    x=fork();
    if(x<0) { printf("fork faild"); exit(1); }   
    if (x==0)
    { execv("co2", argv);
      printf("execv faild");
      exit(1);   }
    printf("%d Parent terminating\n", getpid());
    return;
}
```

Unix: Loading a Process

- The Unix operating system begins the execution of a process with a special startup function loaded when C/C++ program is compiled. This function results in various internal data structures being allocated for the Unix process, such as process table, and file table entries.
- The startup routine is also responsible for establishing the argument count and the argument vector.
- Upon completion of this initialization, the main function of the C/C++ program is then called.
This program receives any number of arguments, prints their number and then prints each argument;
Compilation command: gcc -o co2 co2.c

```c
void main (int arg_count, char **arg_pointer)
{
    int i;
    i=0;
    printf ("arg_count = %d\n", arg_count);
    while (arg_count > 0)
    {
        printf ("arg_value[%d] = %s\n", i, *arg_pointer);
        arg_pointer++;
        arg_count--;
        i++;
    }
    exit();
}
```

Example of one execution:
x$i$|~>co2 a 2 eee 56 76 ad
arg_count = 7
arg_value[0] = co2
arg_value[1] = a
arg_value[2] = 2
arg_value[3] = eee
arg_value[4] = 56
arg_value[5] = 76
arg_value[6] = ad
xi|~>

All operating system services are available through system calls. But in order to issue a system call you have to be running a program.

If we wrote an interactive program that communicated with the user through terminal and with the operating system through the system calls, we would make all the operating system services available interactively to a user sitting at a terminal.

That is exactly what a Unix shell is. It is a process that provides a user the system call interface (through Unix command) for the operating system.

---

**Shell**

- The shell executes commands of the form:
  
  `% CommandName [arg [arg ...]] [&]`  
  (% is the shell prompt)

  This executes the executable program CommandName unless it is `wait` or `cd`. The arguments are passed to the invoked command (program). The shell waits for the command to complete unless the & is present in which case it returns immediately.

  Examples:  
  
  `%lab1
  %ls -a -l

  The `wait` command waits for a program executed earlier with line ending in & to complete.
  
  The `cd` command changes current directory.

---

**Shell (cont.)**

- Many users are not aware of the process concept because they just run a program in a single process. Only a few users create processes using the actual system calls for process creation because normally other programs create processes for you. The shell gives your first process "for free": it is automatically created for you and started up running the program you requested. However, many users actually use multiple processes in their work.

- Effects of command: `%ls | more

  => two processes and a pipe created

  A pipe is a channel (a buffer created by O.S.) between two processes in which one process can write a stream of characters, while the another can read them.
Shell (cont)

"Stripped-down and simple" shell code:
while (true) {
  type_prompt();
  read_command(command, parameters);
  pid = fork();
  if (pid == 0) execv(command, parameters, 0);
  x = wait(&y);
}

Basically, the shell collects words, puts them in an argument array and forks off a process to execute the command. It also handles output redirection, and pipes.

Init Process

- The init process is created automatically by the Unix as the first process in the system.
- The init process then creates an interactive process for each terminal in the system, as well as for any remote access to the system is initiated through a network.
- Each of those interactive processes is executing the login program (code) in it. After successful login, the interactive process obtains a code of the appropriate (for the logged in user) shell.

Unix Processes: Example 5

- Consider the following program:
  (int x,y; x=20, y=35;
   x = fork();
   y = fork();
   if (x==0) execv("A",...);
   if (y==0) execv("B",...);
   ...
 )
- This is the part of the code of a process P. When this code is executed some number of children will be created and initially all of them will have code P. This code or any part of it should be executed completely in all child processes, as well as in any grandchild (or grand-grand child) process. At the end of all those executions, indicate all processes created, parent-child- grandchild relationships, values of variables and the final code in each process.
special instruction will cause an exception, and CPU will start executing the appropriate routine in the operating system.

5. Now, CPU is executing (in kernel mode) code of operating system. The parameters of the system call are examined and the disc controller is instructed to read the certain block from the disc and to copy it (using DMA) into an operating system buffer.

6. The operating system declares Process A as a blocked process, thus ineligible for CPU, since Process A has to wait for a disk I/O to finish (that may take up to 20 milliseconds)

7. The operating system chooses one of ready to run processes, say Process B for running. Then, the operating system switches CPU mode of operation into user mode and makes that the next instruction CPU executes is the appropriate instruction in the code of Process B.

8. Now, CPU is executing (in user mode) code of Process B. We assume that Process B is such that for long time (longer that 20 milliseconds) its code would execute ordinary instructions without system calls.

9. At a certain point in time, while CPU is executing code of Process B, the disc controller finishes the task in 5. above. The controller then causes a hardware interrupt (by activating certain IRQ line). The interrupt will cause that CPU starts execution, in kernel mode, of the appropriate operating system routine. The operating system finds cause of the interrupt and after some processing it will declare Process A as a ready to run, thus eligible for CPU.

10. At this point, the operating system should decide which of ready to run processes should get CPU. Note that Process B is still in ready to run state, since it was interrupted during its normal execution.

Typical File System Data Structures

- As we know, each process is represented in the operating system by a process control block – PCB.
- When a process opens a file, an open file structure is allocated in the (system-wide) open file table and an open file pointer to that structure is placed in the PCB. An open file identifier returned by the open system call is an index into the array of open file pointers in the PCB. The open file identifier is assigned to the process to identify the channel between the process and the open file.
**Typical File System Data Structures (cont.)**

- The open file table contains a structure for each open file including:
  - the current file position, i.e. the file location for the next read or write,
  - status information about the open file, e.g. whether it is open for read or write, the type of file (file, device or pipe), whether the file is locked, etc.
  - a pointer to the file descriptor or in-node (in the system-wide file descriptor table) for the file that is open.

**Unix File System Data Structures (cont.)**

- A file descriptor data structure contains information that includes:
  - the owner of the file,
  - file protection information,
  - location of the file (e.g. address of blocks on the disc),
  - time of creation, last modification, and the last use, etc.
- Note that when a file is opened, the file descriptor is read from a disc into memory and put in a file descriptor table.
- A pointer to the file descriptor for the file that is open can, instead of the file, also be the device driver if the open file is connected to a device, or to a pipe structure if the open file is connected to a pipe.

---

**Process Interaction**

Processes interact in two distinct ways:

1. Through competition for resources; here processes are unaware of each other.
   Problems to solve:
   - scheduling of resources (CPU, memory, I/O controllers)
   - deadlock (involving resource requests)

2. Through cooperation;
   Problems to solve:
   - sharing common variables (critical section problem); here processes are indirectly aware of each other, and they may not necessarily know their respective process ID’s
   - inter-process communication through message exchange
   - synchronization, i.e. situations in which progress of one process depends upon the progress of another process.

In the previous two cases, processes are directly aware of each other, and may be using process ID’s. But conditions are not always clear cut.