OPERATING SYSTEMS

Lecture # 4

Razi Uddin

COMMAND-LINE INTERPRETER (SHELLS)

- Most important system programs for an operating system.
- Interface between the user and operating system.
- Its purpose is to read user commands and try to execute them.
- This program is sometimes called the command-line interpreter and is often known as the shell.
- Its function is simple: to get the next command statement and execute it.
- Famous shells for UNIX and Linux are

✔ Bourne shell (sh),

✔ C shell (csh),

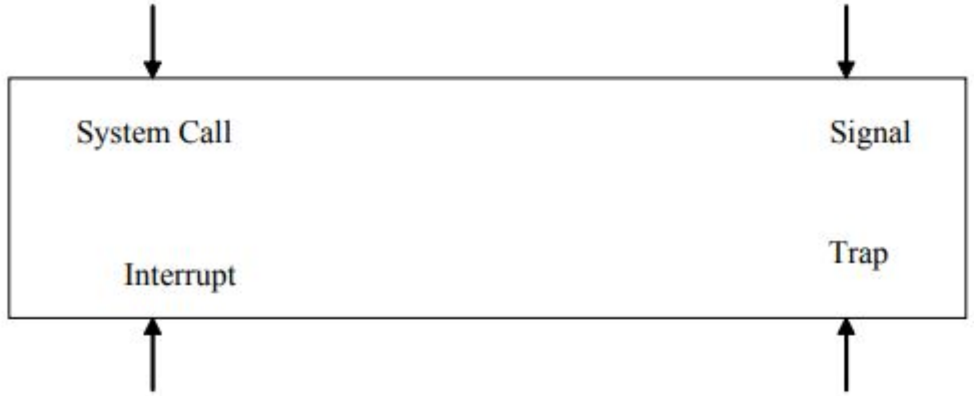
✔ Bourne Again shell (bash),

✔ TC shell (tcsh),

✔ and Korn shell (ksh).

ENTRY POINTS INTO KERNEL

- There are four events that cause the execution of a piece of code in the kernel.
- These events are interrupts, trap, system call, and signal.
- In the case of all of these events, some kernel code is executed to service the corresponding event.



SYSTEM CALLS

- Provide the interface between a process and the OS.
- These calls are generally available as assembly language instructions.
- The system call interface layer contains entry point in the kernel code; because all system resources are managed by the kernel any user or application request that involves access to any system resource must be handled by the kernel code, but user process must not be given open access to the kernel code for security reasons.
- So that user processes can invoke the execution of kernel code, several openings into the kernel code, also called system calls are provided. System calls allow processes and users to manipulate system resources such as files and processes.

SYSTEM CALLS

- System calls can be categorized into the following groups:

✓ Process Control

✓ File Management

✓ Device Management

✓ Information maintenance

✓ Communications

SYSTEM CALLS

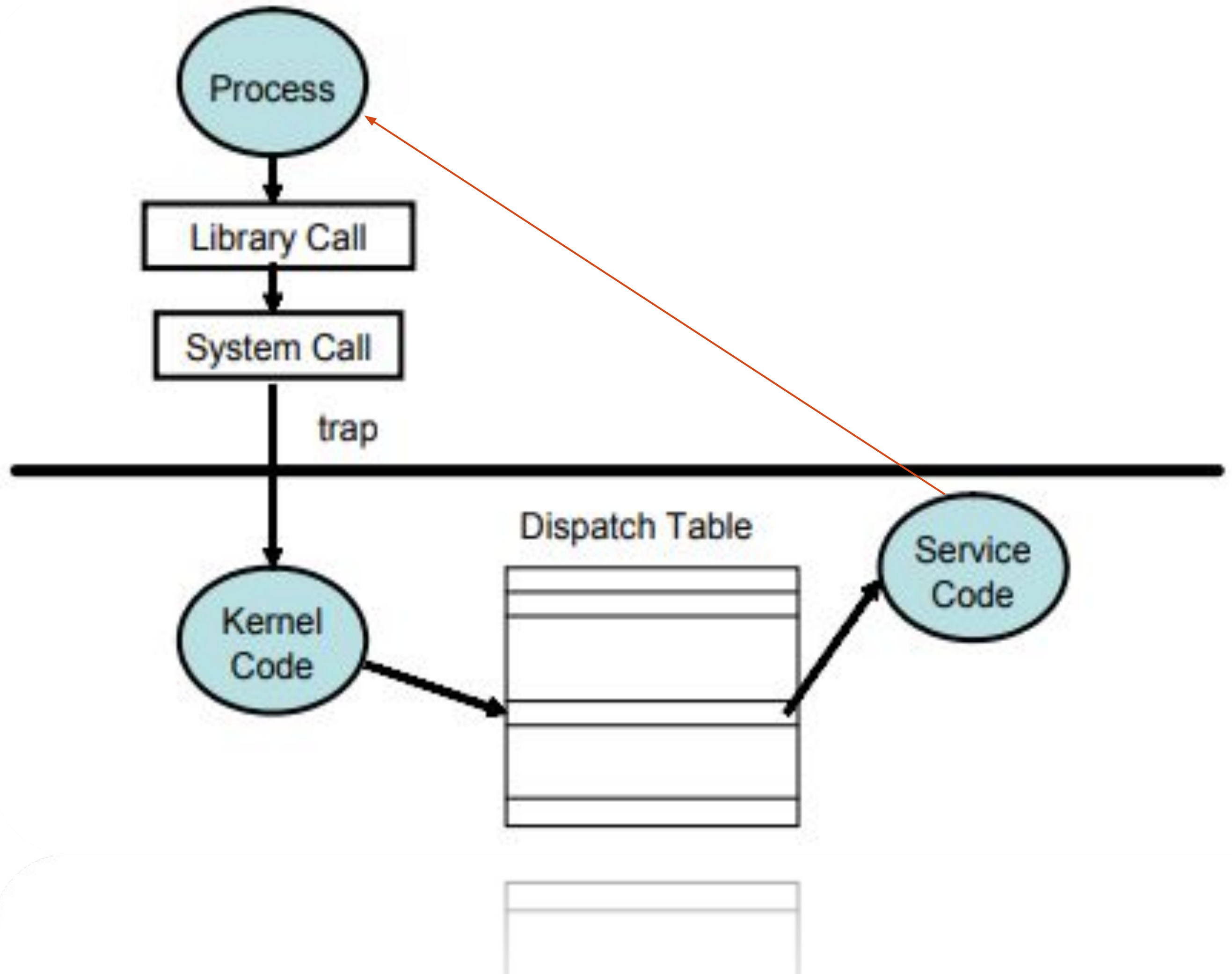
- The following sequence of events takes place when a process invokes a system call:
- The user process makes a call to a library function.
- The library routine puts appropriate parameters at a well-known place, like a register or on the stack. These parameters include arguments for the system call, return address, and call number.
- Three general methods are used to pass parameters between a running program and the operating system.

1. Pass parameters in registers.
2. Store the parameters in a table in the main memory and the table address is passed as a parameter in a register.
3. Push (store) the parameters onto the stack by the program, and pop off the stack by the operating system.

SYSTEM CALLS

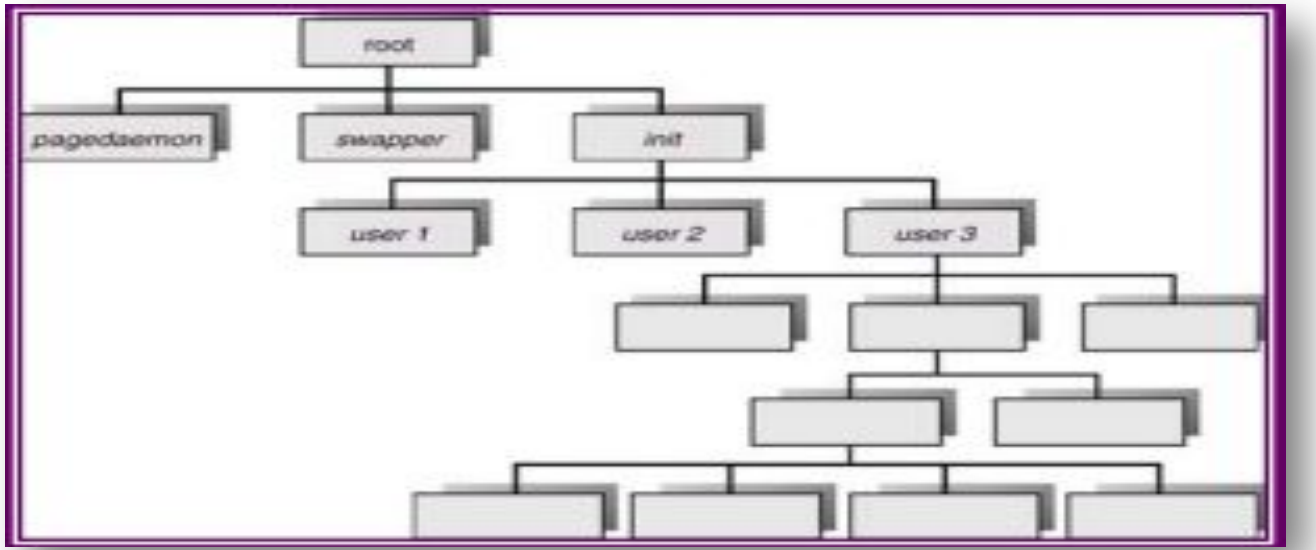
- A trap instruction is executed to change the mode from user to kernel and give control to the operating system.
- The operating system then determines which system call is to be carried out by examining one of the parameters (the call number) passed to it by library routine.
- The kernel uses call number to index a kernel table (the dispatch table) which contains pointers to service routines for all system calls.
- The service routine is executed and control is given back to the user program via return from trap instruction; the instruction also changes mode from system to user.
- The library function executes the instruction following trap; interprets the return values from the kernel and returns to the user process.

SYSTEM CALLS



PROCESS CREATION

- A process may create several new processes via a create-process system call during the course of its execution.
- The creating process is called a parent process while the new processes are called the children of that process.
- Each of these new processes may in turn create other processes, forming a tree of processes.



PROCESS CREATION

- A process will need certain resources (such as CPU time, memory files, I/O devices) to accomplish its task.
- When a process creates a subprocess, also known as a child, that subprocess may be able to obtain its resources directly from the operating system or maybe constrained to a subset of the resources of the parent process.
- The parent may have to partition its resources among several of its children. Restricting a process to a subset of the parent's resources prevents a process from overloading the system by creating too many sub processes.
- When a process is created it obtains in addition to various physical and logical resources, initialization data that may be passed along from the parent process to the child process

PROCESS CREATION

- When a process creates a new process, two possibilities exist in terms of execution:

1. The parent continues to execute concurrently with its children.
2. The parent waits until some or all of its children have terminated.

- There are also two possibilities in terms of the address space of the new process:

1. The child process is a duplicate of the parent process.
2. The child process has a program loaded into it.

PROCESS CREATION

- In UNIX its process identifier identifies a process, which is a unique integer.
- A new process is created by the fork system call.
- The new process consists of a copy of the address space of the parent.
- This mechanism allows the parent process to communicate easily with the child process.
- Both processes continue execution at the instruction after the fork call, with one difference, the return code for the fork system call is zero for the child process,
- While the process identifier of the child is returned to the parent process

PROCESS CREATION

- The execlp system call is used after a fork system call by one of the two processes to replace the process’ memory space with a new program.
- The execlp system call loads a binary file in memory (destroying the memory image of the program containing the execlp system call) and starts its execution.
- In this manner, the two processes are able to communicate and then go their separate ways.
- The parent can then create more children, or if it has nothing else to do while the child runs, it can issue a wait system call to move itself off the ready queue until the termination of the child.
- The parent waits for the child process to terminate, and then it resumes from the call to wait where it completes using the exit system call.

PROCESS TERMINATION

- A process terminates when it finishes executing its final statement and asks the operating system to delete it by calling the exit system call.

void exit(int status)

- At that point, the process may return data to its parent process.
- All the resources of the process including (physical and virtual memory, open files and I/O buffers) are deallocated by the operating system.
- Termination occurs under additional circumstances.
- A process can cause the termination of another via an appropriate system call (such as abort).
- Usually only the parent of the process that is to be terminated can invoke this system call.
- Therefore parents need to know the identities of their children, and thus when one process creates another process, the identity of the newly created process is passed to the parent.

PROCESS TERMINATION

- A parent may terminate the execution of one of its children for a variety of reasons, such as:

1. The child has exceeded its usage of some of the resources that it has been allocated. This requires the parent to have a mechanism to inspect the state of its children.
2. The task assigned to the child is no longer required.
3. The parent is exciting, and the operating system does not allow a child to continue if its parent terminates.

On such a system, if a process terminates either normally or abnormally, then all its children must also be terminated. This phenomenon referred to as cascading termination, is normally initiated by the operating system.

PROCESS TERMINATION

- In UNIX, we can terminate a process by using the exit system call, its parent process may wait for the termination of a child process by using the wait system call.
- The wait system call returns the process identifier of a terminated child so that the parent can tell which of its possibly many children has terminated.
- If the parent terminates however all its children have assigned as their new parent, the init process.
- Thus the children still have a parent to collect their status and execution statistics

SYSTEM CALLS

- Important UNIX/LINX System Calls:

✓ fork

✓ wait

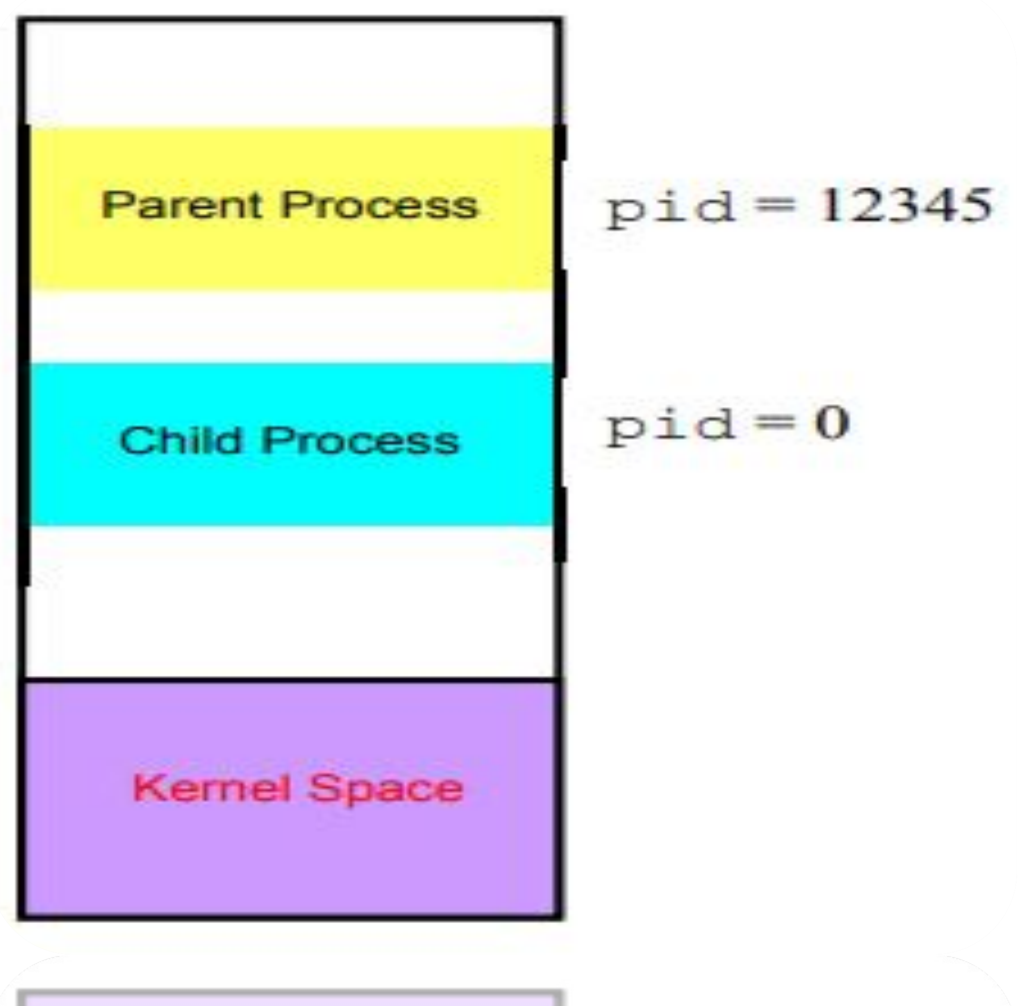
✓ execlp

✓ exit

FORK SYSTEM CALL

- When the fork system call is executed, a new process is created.
- The original process is called the parent process whereas the process is called the child process.
- The new process consists of a copy of the address space of the parent.
- This mechanism allows the parent process to communicate easily with the child process.
- On success, both processes continue execution at the instruction after the fork call, with one difference, the return code for the fork system call is zero for the child process, while the process identifier of the child is returned to the parent process.
- On failure, a -1 will be returned in the parent's context, no child process will be created, and an error number will be set appropriately.

FORK SYSTEM CALL



FORK SYSTEM CALL

- After the fork() system call the parent and the child share the following:
- Environment
- Open file descriptor table
- Signal handling settings
- Nice value
- Current working directory
- Root directory
- File mode creation mask (umask)

FORK SYSTEM CALL

The following things are different in the parent and the child:

- Different process ID (PID)
- Different parent process ID (PPID)
- Child has its own copy of parent's file descriptors

The fork() system may fail due to a number of reasons.

- One reason may be that the maximum number of processes allowed to execute under one user has exceeded, another could be that the maximum number of processes allowed on the system has exceeded.
- Yet another reason could be that there is not enough swap space.

WAIT() SYSTEM CALL

- The wait system call suspends the calling process until one of the immediate children terminates, or until a child that is being traced stops because it has hit an event of interest.
- The wait will return prematurely if a signal is received.
- If all child processes stopped or terminated prior to the call on wait, return is immediate.
- If the call is successful, the process ID of a child is returned.
- If the parent terminates however all its children have assigned as their new parent, the init process.
- Thus the children still have a parent to collect their status and execution statistics.

```
pid_t wait(int *stat_loc)
//take one arg status and returns a PID of dead children.
```

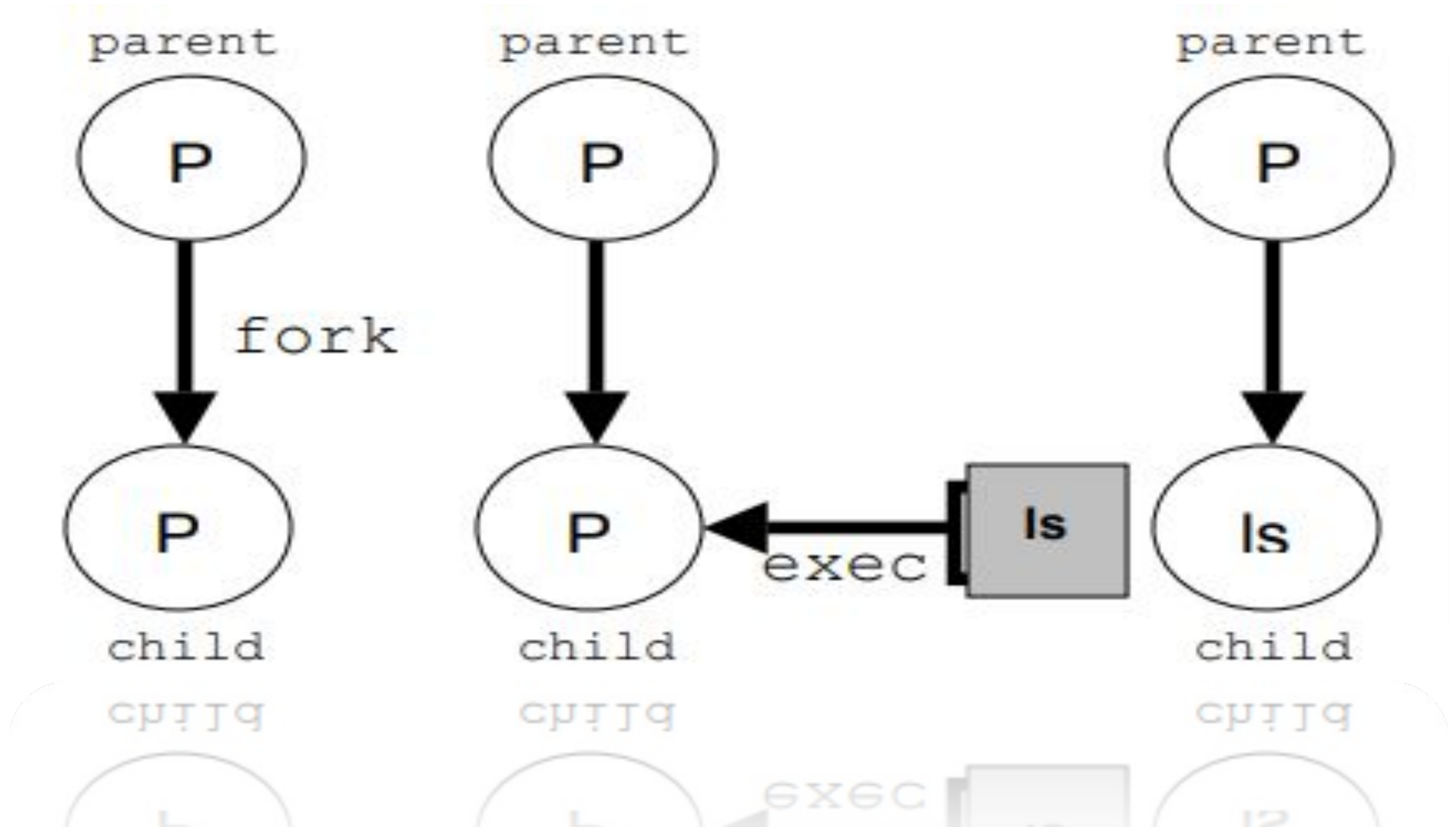
ZOMBIE PROCESS

A process that has terminated but whose exit status has not yet been received by its parent process or by init.

EXECLP() SYSTEM CALL

- Typically, the execlp() system call is used after a fork() system call by one of the two processes to replace the process' memory space with a new program.
- The new process image is constructed from an ordinary, executable file.
- This file is either an executable object file, or a file of data for an interpreter.
- There can be no return from a successful exec because the calling process image is overlaid by the new process image.
- In this manner, the two processes are able to communicate and then go their separate ways.

THE SEMANTICS OF FORK(), FOLLOWED BY AN EXECLP() SYSTEM CALL:


parent
parent
parent
P
P
P
fork
P
P
exec
ls
ls
child
child
child