DAY-1 OS

What is OS?

An Operating System (OS) is an interface between a computer user and computer hardware. An operating system is a software which performs all the basic tasks like file management, memory management, process management, handling input and output, and controlling peripheral devices such as disk drives and printers.

Some popular Operating Systems include Linux Operating System, Windows Operating System, VMS, OS/400, AIX, z/OS, etc.

Some of important functions of an operating System

Processor Management: An operating system manages the processor's working by allocating various jobs to it and ensuring that each process receives enough time from the processor to function properly.

Memory Management: An operating system manages the allocation and deallocation of the memory to various processes and ensures that the other process does not consume the memory allocated to one process.

Device Management: There are various input and output devices. An operating system controls the working of these input-output devices. It receives the requests from these devices, performs a specific task, and communicates back to the requesting process.

File Management: An operating system keeps track of information regarding the creation, deletion, transfer, copy, and storage of files in an organized way. It also maintains the integrity of the data stored in these files, including the file directory structure, by protecting against unauthorized access.

Security: The operating system provides various techniques which assure the integrity and confidentiality of user data. Following security measures are used to protect user data:

- Protection against unauthorized access through login.

- Protection against intrusion by keeping Firewall active.
- Protecting the system memory against malicious access.
- Displaying messages related to system vulnerabilities.

Error Detection: From time to time, the operating system checks the system for any external threat or malicious software activity. It also checks the hardware for any type of damage. This process displays several alerts to the user so that the appropriate action can be taken against any damage caused to the system.

Job Scheduling: In a multitasking operating system where multiple programs run simultaneously, the operating system determines which applications should run in which order and how time should be allocated to each application.

Interaction with OS

The application programs make use of the operating system by making requests for services through a defined application program interface (API). In addition, users can interact directly with the operating system through a user interface, such as a command-line interface (CLI) or a graphical UI (GUI).

Why is OS hardware dependent?

We should understand that the following things are hardware dependent:

- System start-up/reset
- Interrupt handling
- Virtual memory management & protection
- Device I/O
- System-level protections for code access and security
- Some mutual exclusion primitives

Now if we see closely, all above are functions of an OS. Hence OS is hardware dependent.

Platform dependent typically refers to applications that run under only one operating system in one series of computers (one operating environment); for example, Windows running on x86 hardware or Solaris running on SPARC hardware. Sometimes, it means the same as "hardware dependent" or "machine dependent" and refers to applications that run in only one hardware series with the operating system not being relevant.

In contrast, "platform independent" means that the application can run in different operating environments. Applications written in Java are a prime example

What did we learn so far about an OS (yeah in simple terms)

---

1. A program which controls the execution of all other programs (applications).

2. Acts as an intermediary between the user(s) and the computer.

3. Objectives:

   – Convenience,
   – Efficiency,
   – Extensibility

4. The Operating System (OS):

   – Controls all execution.
   – Multiplexes resources between applications.
   – Abstracts away from complexity.

Components of an OS

---

1. Shell
2. Kernel

1. SHELL:

Shell handles user interactions. It is the outermost layer of the operating system and manages the interaction between user and operating system by:

- Prompting the user to give input
- Interpreting the input for the operating system
- Handling the output from the operating system

2. KERNEL:

Kernel is the core component of an operating system which acts as an interface between applications, and the data is processed at the hardware level.

The kernel is responsible for performing the following tasks:

- Input-Output management
- Memory Management
- Process Management for application execution.
- Device Management
- System calls control

~~~~~~

Ques 1 - Difference between kernel and OS (10 min)

Good reference: https://www.geeksforgeeks.org/difference-between-operating-system-and-kernel/

Ans.:

Kernel:

A kernel is the core component of an operating system. It is the part of operating system which converts user commands into machine language.

Operating System (OS):

It is a system program that provides interface between user and computer. When computer boots up operating system is the first program that loads.

Difference between kernel and OS:

- Kernel is the core part of OS while OS is a collection of software that manages computer hardware resources.
- Kernel is system software which is part of operating system whereas OS is a system software.
- Kernel provides interface between applications and hardware whereas OS provides interface between user and hardware.
- The main purpose of Kernel is memory management, disk management, process management and task management while OS provides protection and security.
- All system needs OS to run while all OS need Kernel to run.
- Type of Kernel includes monolithic and micro Kernel while type of OS includes single and multiuser OS, multiprocessor OS, real-time OS, distributed OS.
- Kernel is the first program to load when operating system loads while OS is the first program to load when computer boots up.

~~~~~~

Types of Operating Systems

- Batch OS
- Distributed OS
- Multitasking OS
- Network OS
- Real-OS
- Mobile OS

Ques2 - Write one-liner on each OS with examples

Ans. Types of OS:

Batch OS: This OS does not directly interact with the computer. Instead, an operator takes up similar jobs and groups them together into a batch, and then these batches are executed one by one based on the first-come, first, serve principle.

Examples of Batch OS: Payroll system, bank statements, data entry, etc.

Distributed OS: In a distributed OS, various computers are connected through a single communication channel. These independent computers have their memory unit and CPU and are known as loosely coupled systems. The system processes can be of different sizes and can perform different functions.

Examples of Distributed OS: LOCUS, MICROS, IRIS, DYNIX etc.

Multitasking OS: The multitasking OS is also known as the time-sharing operating system as each task is given some time so that all the tasks work efficiently. This system provides access to a large number of users, and each user gets the time of CPU as they get in a single system.

Examples of Multitasking OS: UNIX, IBM's OS/390, Linux, Windows 2000, Windows10, etc.

Network OS: Network operating systems are the systems that run on a server and manage all the networking functions.

Examples of Network OS: Microsoft Windows server 2008, LINUX, Mac OS X, Novell NetWare, etc.

Real-Time OS: Real-Time operating systems serve real-time systems. These operating systems are useful where many events occur in a short time or certain deadlines, such as real-time simulations.

Examples of Real-Time OS: Medical imaging systems, robots, Windows CE (Microsoft Windows), LynxOS, VxWorks (Wind River), etc.

Mobile OS: A mobile OS is an operating system for smartphones, tablets, and PDA's. It is a platform on which other applications can run on mobile devices.

Examples of Mobile OS: Android OS, iOS, Symbian OS, and windows mobile OS, etc.

Modes of operation in OS

User mode --> mode bit = 1

Kernel mode --> mode bit = 0

Important: The mode bit is set to 0 in the kernel mode. It is changed from 0 to 1 when switching from kernel mode to user mode.

Kernel Mode

In Kernel mode, the executing code has complete and unrestricted access to the underlying hardware. It can execute any CPU instruction and reference any memory address.

Kernel mode is generally reserved for the lowest-level, most trusted functions of the operating system. Crashes in kernel mode are catastrophic; they will halt the entire PC.

User Mode

In User mode, the executing code has no ability to directly access hardware or reference memory. Code running in user mode must delegate to system APIs to access hardware or memory. Due to the protection afforded by this sort of isolation, crashes in user mode are always recoverable. Most of the code running on your computer will execute in user mode.

Example

--

Regular user space programs evoke system calls all the time to get work done, for example:

- ls
- ps
- top
- bash

Digging one layer deeper, the following are some example system calls which are invoked by the above listed programs. Typically these functions are called through libraries such as glibc, or through an interpreter such as Ruby, Python, or the Java Virtual Machine.

- open (files)
- getpid (processes)
- socket (network)

A typical program gets access to resources in the kernel through layers of abstraction as follows:

- User Programs
- Library/Interpreter
- System Calls
- Kernel Space

--

DAY-2

Full screen mode -> Click inside the machine and Press Ctrl+Alt+Enter

To clear data on screen -> Ctrl+L

[lavishjhamb@localhost ~]$

[username@machinename <location>]

~ -> <home of the user>

$ --> user in action is a regular user

--> user in action is a root user

'root' user is main user is Linux with user id '0'. It has all the permissions possible.

[root@localhost ~]#

Kind of users in Linux

---

1. Root User

2. Regular User

3. Super User (Sudoer)

To Change location in Linux - To traverse in Linux

---

cd <location where you want to go>

File Hierarchy in Linux - Directory Structure in Linux

/

Folders inside '/'

bin dev home lib64 mnt proc run srv tmp var

boot etc lib media opt root sbin sys usr

Ques 1 - Explain what's inside each folder under '/'

/bin: Essential command binaries that need to be available in single-user mode; for all users, e.g., cat, ls, cp.

/dev: Essential device files, e.g., /dev/null.

/home: Users' home directories, containing saved files, personal settings, etc.

/lib64: Alternate format essential libraries

/mnt: Temporarily mounted filesystems.

/proc: Virtual filesystem providing process and kernel information as files. In Linux, corresponds to a procfs mount.Generally, automatically generated and populated by the system, on the fly.

/run: Run-time variable data: Information about the running system since last boot, e.g., currently logged-in users and running daemons.

/srv: Site-specific data served by this system, such as data and scripts for web servers, data offered by FTP servers, and repositories for version control systems (appeared in FHS-2.3 in 2004).

/tmp: Directory for temporary files (see also /var/tmp). Often not preserved between system reboots and may be severely size-restricted.

/var: Variable files: files whose content is expected to continually change during normal operation of the system, such as logs, spool files, and temporary e-mail files.

/boot: Boot loader files (e.g., kernels, initrd).

/etc: Host-specific system-wide configuration files.

/lib: Libraries essential for the binaries in /bin and /sbin.

/media: Mount points for removable media such as CD-ROMs (appeared in FHS-2.3 in 2004).

/opt: Add-on application software packages.

/root: Home directory for the root user.

/sbin: Essential system binaries (e.g., fsck, init, route).

/sys: Contains information about devices, drivers, and some kernel features.

/usr: Secondary hierarchy for read-only user data,contains the majority of (multi-)user utilities and applications. Should be shareable and read-only.

To see current location

---

pwd --> present working directory

Meaning of '.' and '..'

. --> same location

.. --> previous directory

[root@localhost audit]# cd .

[root@localhost audit]# pwd

/var/log/audit

[root@localhost audit]# cd ..

[root@localhost log]# cd ..

[root@localhost var]# cd ..

[root@localhost /]# cd /var/log/audit/

[root@localhost audit]# cd ../../..

[root@localhost /]#

Folder (Directory) creation

---

mkdir <foldername>

[root@localhost edac_os]# pwd

/root/edac_os

[root@localhost edac_os]# mkdir dir1

[root@localhost edac_os]# mkdir dir2

[root@localhost edac_os]# mkdir dir3

[root@localhost edac_os]# ls

dir1 dir2 dir3

Folder removal

---

rm -rf <absolute path of the folder>

rm -> remove

r -> recursively

f -> forcefully

[root@localhost edac_os]# rm -rf /root/edac_os/dir3

[root@localhost edac_os]# ls

dir1 dir2

Remove multiple folders at a time

[root@localhost edac_os]# rm -rf *

How to create files

---

touch <filename> -> it will create a blank file

Scenario we are working on

We are under /root/edac_os

and this contains - dir1 and dir2

and dir1 contains - file1 and file2

and dir2 contains - file3 and file4

Condition: Current location -> '/'

Task

Delete file1 under dir1

[root@localhost /]# rm -rf /root/edac_os/dir1/file1

[root@localhost ~]# cd /root/edac_os/dir1

[root@localhost dir1]# ls

file2

Create file5 under dir2

[root@localhost /]# touch /root/edac_os/dir2/file5

[root@localhost /]# cd /root/edac_os/dir2

[root@localhost dir2]# ls

file3 file4 file5

Editing of a file using 'vi'

---

vi <filename>

Press i --> insert

<Fill in the content>

Press Esc

:wq! --> Save my data and exit

:q! --> Exit

:w! --> just saves the current data

To see the contents of file

---

cat <filename>

[root@localhost dir2]# vi file3

[root@localhost dir2]# cat file3

I'm editing file3 because Im bound to

Edit file2 and write "I disturb during tasks" in it

vi /root/edac_os/dir1/file2

Added the data, :wq!

[root@localhost /]# cat /root/edac_os/dir1/file2

I disturb during tasks

Renaming a file

---

mv <old name> <new name>

Copying/ renaming of file

---

cp <source location> <destination location>

[root@localhost dir1]# ls

file1

[root@localhost dir1]# cd ../dir2

[root@localhost dir2]# ls

file3 file4 file5

Task - Copy file5 to dir1

Condition: You are under '/'

Solution:

[root@localhost /]# cp /root/edac_os/dir2/file5 /root/edac_os/dir1

[root@localhost /]# ls /root/edac_os/dir2

file3 file4 file5

[root@localhost /]# ls /root/edac_os/dir1

file1 file5

Delete file5 under dir1 and move all files from dir2 to dir1

Expected o/p under dir1

file3, file4, file1, file5

[root@localhost /]# cp /root/edac_os/dir2/file5 /root/edac_os/dir1

[root@localhost /]# ls /root/edac_os/dir2

file3 file4 file5

[root@localhost /]# ls /root/edac_os/dir1

file1 file5

[root@localhost /]# rm -rf /root/edac_os/dir1/file5

[root@localhost /]# mv /root/edac_os/dir2/* /root/edac_os/dir1

[root@localhost /]# ls /root/edac_os/dir2

[root@localhost /]# ls /root/edac_os/dir1

file1 file3 file4 file5

DAY-3 OS

Type of shells

---

bash

sh

csh

ksh

nologin

To know which shell you are working in: **echo $SHELL**

Users in Linux

---

root user -> 0

super user - sudoer

regular user -> is above 1000 (centOS7)

This min UID is decided in /etc/login.defs

UID_MIN 1000

Where can I see the list of all users in Linux?

- Whenever we create a user in Linux, a folder is created with same name as that of user in /home

There is a file where we can see the info of all users - /etc/passwd

username:x:UID:GID:comment:<home location>:<shell that user will use>

How to create a user in Linux

CentOS

useradd <username>

passwd <username>

adding user with a comment

useradd -c 'comment' username

adding user with a UID

useradd -u <UID> username

--

useradd -c 'someshwar' alice

useradd -u 1002 user1

--

Ubuntu

adduser <username>

How to delete a user

---

userdel -r <username>

Switching of users

---

su - <username>

Root can switch to any user without providing password

Other users require password for switching

Importance of 'x'

Where are users password stored? - /etc/shadow

The 2nd field 'x' in /etc/passwd is a link to shadow file that contains encrypted password

x ----------> encrypted password

Without x in /etc/passwd - password won't be asked

File Permissions

Owner - User who created the file

Group - The group which user is a part of

Others - rest of users

Permissions

---

r Read

w Write

x Execute

4 Read

2 Write

1 Execute

Full permissions over a file means

rwx or 7 (4+2+1)

How permission of a file looks like

--- --- ---

Owner Group Other users

Umask

By default OS gives this permission

Folder - 777

File - 666

[root@localhost edac_os]# touch file

[root@localhost edac_os]# mkdir folder

[root@localhost edac_os]# ls

file folder

[root@localhost edac_os]# ls -l

total 0

-rw-r--r--. 1 root root 0 Sep 24 09:17 file -> 644

drwxr-xr-x. 2 root root 6 Sep 24 09:18 folder --> 755

This Umask gets subtracted from the default permissions set

Now our umask value is 022

Folder

777

022

755

If umask was not there, then any file created would have got 666 as permission set whoch means

Owner - rw

Group - rw

Others - rw

How to modify the permissions of a file

---

chmod <permission set> filepath

How to change the ownership of a file

---

chown <new owner:new group owner> filename/path

[root@localhost rahul]# chown shruti:shruti file_r

[root@localhost rahul]# ls -l

total 0

-rw-rw-r--. 1 shruti shruti 0 Sep 24 09:28 file_r

Note: Here we changed the owner as well as group owner

[root@localhost rahul]# chown tom folder_r

[root@localhost rahul]# ls -l

total 0

drwxrwxr-x. 2 tom rahul 6 Sep 24 09:29 folder_r

Note: Here we just changed the owner and not group owner

[root@localhost rahul]# chown :alice folder_r

[root@localhost rahul]# ls -l

total 0

drwxrwxr-x. 2 tom alice 6 Sep 24 09:29 folder_r

Note: Here we just changed the group owner

Important: Only 'owner or root' can change the permissions and ownership of a file

ACL - Access Control List

[root@localhost edac_os]# ls -l file2 ## here we are listing permissions of file2

-rw-r--r--. 1 root root 0 Sep 24 09:41 file2 ## here we saw that others can just read

[root@localhost edac_os]# chmod 647 file2 ## here we provided full access to others

[root@localhost edac_os]# ls -l file2 ## here we are listing permissions of file2

-rw-r--rwx. 1 root root 0 Sep 24 09:41 file2 ## here we saw that others can read, write and execute

Issue: root wanted to give full access to Rahul ONLY but via chmod there is no provision for such advanced request

ACl --> setfacl -m u:<username>:<permission set> filename/path

[root@localhost edac_os]# getfacl file1

file: file1

owner: root

group: root

user::rw-

user:rahul:rwx

group::r--

mask::rwx

other::r--

Getting permissions in numeric format

[root@localhost edac_os]# ls -l file3

-rw-r--r--. 1 root root 0 Sep 24 09:41 file3

[root@localhost edac_os]# stat -c %a file3

644

SSH

Secure Shell - Used to take shell of different Linux machines

Synatx: ssh <username of the other machine>@<IP of the other machine>

Usage:

[root@localhost edac_os]# ssh ljhamb@192.168.91.147

It will ask for password

Transfer all files on other machine via copying

---

scp - secure copy, copy over ssh

scp <location of files you want to transfer> <username of the other machine>@<IP of the other machine>:<location where you want to transfer>

Usage

[root@localhost edac_os]# scp -r /root/edac_os/* ljhamb@192.168.91.147:/home/ljhamb/test/

ljhamb@192.168.91.147's password:

file1 100% 0 0.0KB/s 00:00

file2 100% 0 0.0KB/s 00:00

file3 100% 0 0.0KB/s 00:00

file4 100% 0 0.0KB/s 00:00

file5 100% 0 0.0KB/s 00:0

DAY 4 OS

Redirection

[root@localhost edac_os]# echo "hello" ## display on screen

hello

[root@localhost edac_os]# echo "hello" > file ## redirect content in the file

[root@localhost edac_os]# cat file

hello

[root@localhost edac_os]# echo "world" > file ##overwrite content of file

[root@localhost edac_os]# cat file

world

[root@localhost edac_os]# rm -rf *

##repeat the same scenario for appending data

```
[root@localhost edac_os]# echo "hello" > file
[root@localhost edac_os]# cat file
hello
[root@localhost edac_os]# echo "world" >> file
[root@localhost edac_os]# cat file
hello
world
```

Running multiple commands in a single line

Create a file
Display permissions of the file
Change permissions of the file to 755
Display new permissions of the file
Enter content "hello edac" in this file
Display content of the file

```
touch file
stat -c %a file
chmod 755 file
stat -c %a file
echo "hello edac" > file
cat file
```

Syntax: command1;command2;command3

```
touch file;stat -c %a file;chmod 755 file;stat -c %a file;echo "hello edac" > file;cat file
```

Another way to do this is: command1 && command2 && command3

Difference - if you use && - then it means the next command would run only if previous command ran successfully

[root@localhost edac_os]# stat -c %a file && chmod 755 file && stat -c %a file && echo "hello edac" > file && cat file

stat: cannot stat ‘file’: No such file or directory

[root@localhost edac_os]# ls

[root@localhost edac_os]#

Usage of grep

---

grep is used to display a line on the basis of a word/pattern

Syntax: grep "word/pattern"

Options in grep

--

[root@localhost edac_os]# cat things

apple

APPLE

Apple

ApplE

Mango

mangoes

kiwi

banana

grape

grapes

GRapes are sour

tomato.

orange.

Orange is nice fruit
I like Taj mahal
red fort is in Delhi
My phone number is 9643546697
My IP adress is 8.8.8.8
Your IP adress is 192.168.10.142

Usecase: We want to fetch apple
[root@localhost edac_os]# cat things | grep "apple"
apple
Linux is case sensitive
[root@localhost edac_os]# cat things | grep -i "apple"
apple
APPLE
Apple
ApplE
##Case sensitivity is ignored

[root@localhost edac_os]# cat things | grep "mango"
mangoes
[root@localhost edac_os]# cat things | grep -i "mango"
Mango
mangoes

[root@localhost edac_os]# cat things | grep -o "mango"
mango

##o is for only

```
[root@localhost edac_os]# cat things | grep -i -o "mango"
Mango
mango
--
[root@localhost edac_os]# cat things | grep "orange"
orange.
[root@localhost edac_os]# cat things | grep -i "orange"
orange.
Orange is nice fruit
[root@localhost edac_os]# cat things | grep -i -o "orange"
orange
Orange
##just orange - without sentence

#############
^ - starting
$ - ending
#############

[root@localhost edac_os]# cat things | grep "^A"
APPLE
Apple
ApplE
## all that started with cap A

[root@localhost edac_os]# cat things | grep "\."
tomato.
orange.
My IP adress is 8.8.8.8
```

Your IP adress is 192.168.10.142

all lines which have '.'

[root@localhost edac_os]# cat things | grep "\.$"

tomato.

orange.

##ending with a dot

[root@localhost edac_os]# cat things | grep -P "\d{10}"

My phone number is 9643546697

[root@localhost edac_os]# cat things | grep -Po "\d{10}"

9643546697

[root@localhost edac_os]# cat things | grep -Po "\d\.\d\.\d\.\d"

8.8.8.8

[root@localhost edac_os]# cat things | grep -Po "\d+\.\d+\.\d+\.\d+"

8.8.8.8

192.168.10.142

[root@localhost edac_os]# cat things | grep "^$"

##it brings all the blank lines

[root@localhost edac_os]# cat things | grep tomato

tomato.

[root@localhost edac_os]# cat things | grep -v tomato ##inverse

apple

APPLE

Apple

ApplE

Mango

mangoes

kiwi

banana

grape

grapes

GRapes are sour

orange.

Orange is nice fruit

I like Taj mahal

red fort is in Delhi

My phone number is 9643546697

My IP adress is 8.8.8.8

Your IP adress is 192.168.10.142

Pipe in commands

---

syntax: command1 | command2 | command3 .....

What it means?

Output of command1 will act as input of command2

Usage

[root@localhost edac_os]# cat /etc/passwd | grep "rahul"

rahul:x:1005:1005::/home/rahul:/bin/bash

Domain names from a random site

[root@localhost edac_os]# curl -s "https://www.paltalk.com/" | grep -Po "\w+\.com" | sort | uniq

~~~~~~

To show []

[root@localhost edac_os]# cat things | grep "^[ap].*"

apple

[root@localhost edac_os]# cat things | grep -i "^[ap].*"

Putting command output in a variable

Syntax: variablename=$(command)

[root@localhost edac_os]# fruits=$(cat things | grep -i "^[ap].*")

[root@localhost edac_os]# echo "$fruits"

apple

APPLE

Apple

ApplE

Syntax: variablename=`command`

myfruit=`cat things | grep -i "^[ap].*"`

[root@localhost edac_os]# echo "$myfruit"

apple

APPLE
~~~~~~

Apple

ApplE

################

SHELL Scripting

################

Steps:

1. Enter the commands in a file

Note: When you enter the commands in the file - the first thing that you need to enter is this -- #!/bin/bash

This tells your kernel to execute this code in bash shell

2. Save the file

3. Give executable permissions to the file

chmod +x <filename>

4. To run this executable(script) - ./<filename>

[Task 1]

1. Create a directory named test

2. Create 100 files and 100 directories under test

3. Display permissions of test

4. Modify permissions of test to 777

5. Display new permissions of test

--

[root@localhost scripts]# cat task1.sh

#!/bin/bash

#Create a directory named test

#Create 100 files and 100 directories under test
#Display permissions of test
#Modify permissions of test to 777
#Display new permissions of test

mkdir test
touch test/file{1..100}
mkdir test/folder{1..100}
stat -c %a test
#ls -l test
chmod 777 test
stat -c %a test
#ls -l test

[root@localhost scripts]# vi task2.sh
[root@localhost scripts]# chmod +x task2.sh
[root@localhost scripts]# ./task2.sh
--

[Task 2]
Create a script that brings out all the users who are using bash shell

--
[root@localhost scripts]# cat task2.sh
#!/bin/bash

#Create a script that brings out all the users who are using bash shell

echo "Following users are using bash shell on this Linux box"

sleep 2s

cat /etc/passwd | grep "bash"
--

cut and awk
---

cut

Syntax: cut -d'<delimiter>' -f<fieldname>

Usage
[root@localhost scripts]# cat /etc/passwd | grep bash| cut -d':' -f1
root
lavishjhamb
tom
alice
user1
rahul
shruti
yash

awk
---
awk -F'<field separator>' '{print $<fieldname>}'

Usage

[root@localhost scripts]# cat /etc/passwd | grep bash| awk -F':' '{print $1}'

root

lavishjhamb

tom

alice

user1

rahul

shruti

yash

[Task3]

Create a script that stores the following information about all users using bash shell in a varibale named "info_user": 'username' and it's 'shell'.

Display the contents of the variable after initiating a wait period of 5s

--

[root@localhost scripts]# cat task3.sh

#!/bin/bash

#Create a script that stores the following information about all users using bash shell in a varibale named "info_user": 'username' and it's 'shell'.

#Display the contents of the variable after initiating a wait period of 5s

#syntax to put in variable is - var=$(command)

info_user=$(cat /etc/passwd | grep "bash" | awk -F':' '{print $1, $7}')

sleep 5s

echo "$info_user"

--

User input in scripts

'read' is used to take input from user in a script

Ex:

read -p "Enter your favourite monument name: " monument

So, here -p is used to prompt the content present in "" and read is used take input where the input is stored in the variable(like monument in this case)

[Task4]

Create a script that asks for a user name displays the user ID of the user provided as input.

#!/bin/bash

#Create a script that asks for a user name displays the user ID of the user provided as input.

read -p "Enter the username: " usr

echo "The user ID of $usr is as follows:"

sleep 3s

echo ""

cat /etc/passwd | grep "bash" | grep "$usr" | awk -F':' '{print $3}'

--

If loop

[Task 5]

Create a script that asks for a user name displays the user ID of the user provided as input. If the user does not exists, then display the following error: "Entered user is not present on you Linux machine"

--

[root@localhost scripts]# cat task5.sh

```
#!/bin/bash


#Create a script that asks for a user name displays the user ID of the user provided as input.


#If the user does not exist, then display the following error: "Entered user is not present on you Linux machine"

###############

read -p "Enter the username: " usr


##Following variable will give the user ID only when user is presnt, else the variable will be blank.


var=$(cat /etc/passwd | grep "bash" | grep -w "$usr" | awk -F':' '{print $3}')


##using if condition to check if var is blank or not


if [[ -n "$var" ]]
        then
       echo "User ID is: $var"
        else
       echo "User does not exist"
fi
```

##Create a script that displays all Linux users with ID greater than 1000

[root@localhost scripts]# cat task6.sh

#!/bin/bash

##Create a script that displays all Linux users with ID greater than 1000

#https://ryanstutorials.net/bash-scripting-tutorial/bash-if-statements.php

cat /etc/passwd| awk -F':' '{if ($3 > 1000) {print $1, $3}}'

##$3 is the userID

##$1 is the username

DAY 5 OS

[Task1]

Create a script that takes a number as user input and tells if prvided number is a two digit number or single digit number.

[Task2]

Create a command named as 'myos' - running this command should display the OS you are working on

[Task3]

Create a command named as 'myshell' - running this command should display the shell you are working on

[Task4]

Create a script that takes a path and tells you if provoided path is a file or a directory

[Task5]

Create a script that takes a directory name and lists down all the files (only files) under that dir.

[Task6]

Create a script to get the current date, time, username and current working directory.

[Task7]

Create a script that creates a dir named test and then creates 5 files under it named as file1, file2..file5 and then renames all files by adding .txt extension to all file names

Creating a command

[root@localhost scripts]# alias kernelinfo="uname -a"

[root@localhost scripts]#

[root@localhost scripts]# kernelinfo

Linux localhost.localdomain 3.10.0-1160.el7.x86_64 #1 SMP Mon Oct 19 16:18:59 UTC 2020 x86_64 x86_64 x86_64 GNU/Linux

Q - How to make the alias permanent?

Ans. Goto Terminal .

Type $ vi ~/.bashrc and hit Enter

Add your new aliases

examples-

#My custom aliases

alias gpuom='git push origin master'

alias gplom='git pull origin master'

Save and Exit (Press Esc then type :wq).

To See a List of All the aliases type alias hit Enter.

All new aliases will be available next time you login using a new ssh/terminal session. To load changes immediately, type the following source command:

$ source ~/.bash_aliases

Usage of find command

How will search for a file name in a directory?

find <top level location> -name '<name of file>'

[root@localhost ~]# find /root -name 'myfile'

/root/edac_os/folder_new/folder_new1/folder_new2/folder_new3/folder_new4/myfile

Calling script with arguments

We run the script as follows:

./<scriptname>

bash <scriptname>

./scriptName "arg1" "arg2"..."argn"

ex: if folloiwng is the script:

#!/bin/bash

echo "First parameter is $1"

echo "Second parameter is $2"

echo "Third parameter is $3"

and you run it like this: ./parameters.sh 50 51 52

The output will be:

First parameter is 50

Second parameter is 51

Third parameter is 52

Prompt Statement variables

There are several variables that can be set to control the appearance of the bash command prompt: PS1, PS2, PS3, PS4 and PROMPT_COMMAND the contents are executed just as if they had been typed on the command line.

PS1 – Default interactive prompt (this is the variable most often customized)

PS2 – Continuation interactive prompt (when a long command is broken up with \ at the end of the line) default=">"

PS4 – Prompt used when a shell script is executed in debug mode (“set -x” will turn this on) default ="++"

To see the current value of PS1

--

[root@localhost edac_os]# echo $PS1

[\u@\h \W]\$

--

For ex: We can set the prompt by changing the value of the PS1 environment variable, as follows:

$ export PS1='My prompt$'

My prompt$

--

Special prompt variable characters:

\d The date, in "Weekday Month Date" format (e.g., "Tue May 26").

\h The hostname, up to the first . (e.g. deckard)

\H The hostname. (e.g. deckard.SS64.com)

\j The number of jobs currently managed by the shell.

\l The basename of the shell's terminal device name.

\s The name of the shell, the basename of $0 (the portion following the final slash).

\t The time, in 24-hour HH:MM:SS format.

\T The time, in 12-hour HH:MM:SS format.

\@ The time, in 12-hour am/pm format.

\u The username of the current user.

\v The version of Bash (e.g., 2.00)

\V The release of Bash, version + patchlevel (e.g., 2.00.0)

\w The current working directory.

\W The basename of $PWD.

\! The history number of this command.

\# The command number of this command.

\$ If you are not root, inserts a "$"; if you are root, you get a "#" (root uid = 0)

\nnn The character whose ASCII code is the octal value nnn.

\n A newline.

\r A carriage return.

\e An escape character (typically a color code).

\a A bell character.

\\ A backslash.

\[Begin a sequence of non-printing characters. (like color escape sequences). This allows bash to calculate word wrapping correctly.

\] End a sequence of non-printing characters.

Note: Using single quotes instead of double quotes when exporting your PS variables is recommended, it makes the prompt a tiny bit faster to evaluate.

For example : Question is - Set up a prompt like: [username@hostname:Currentlocation]$

Solution: export PS1='[\u@\h:\w]\$'

Here 'export' helps us to Set an environment variable.

PS2 example

[root@localhost edac_os]# echo "This is a very long comment. How long is it? \

> It's so long that I continued it on the next line."

This is a very long comment. How long is it? > It's so long that I continued it on the next line.

[root@localhost edac_os]# touch \

> filename

[root@localhost edac_os]# ls

filename

For loop

Create 5 files in a dir named 'test' with filenames uch as file1, file2, file3, file4 and file5 - Use for loop

--

[root@localhost scripts]# cat for.sh

#!/bin/bash

#Create 5 files in a dir named 'test' with filenames uch as file1, file2, file3, file4 and file5 - Use for loop

##remove test if already present

rm -rf test

echo "My current location is `pwd`"

mkdir test;cd test ## created test folder and went inside it

for i in {1..5} ## initiated for loop

 do

   touch file$i

 done

ls ##listed all files created under test folder

cd .. ## went out of test folder

--

~~for is mostly used when values in range are space separated

~~while is mostly used when values in range are line separated

While loop

You have a list of users. Traverse through the list and display the shell each user is using?

File with user names is as userlist.txt and contents of it are as follows:

root

lavishjhamb

tom

alice

user1

rahul

shruti

yash

tommy

Syntax:

--

while <condition>

do

<commands>

done

--

```
[root@localhost while]# cat while.sh
#!/bin/bash

while read line

do

shell=$(cat /etc/passwd | grep "^$line\b"  | awk -F':' '{print $7}')
echo "$line ----> $shell"


done < userlist.txt
```

--

DAY 6 OS

Process and Process Management

Whenever we execute a command, it creates a new process.

Process - program in execution or instance of a running program.

Each process in the system has a unique PID.

How to the pid of a process? pidof <processname>

Starting of a process

When we start a process (run a command), there are 2 ways of running it:

Foreground Process - by default, every process runs in foreground - For ex: sleep 60s
Background Process - user adds and & at the end of command - For ex: sleep 60s &

Bring a background process to foreground

[root@ljhamb ~]# sleep 60s & ## sent the process to bg
[1] 63592
[root@ljhamb ~]# fg %1 ## brought it back to fg using job id
sleep 60s

[root@ljhamb ~]# sleep 90s ## ran the process in fg
^Z   ## sent it to bg using ctrl+z
[1]+ Stopped   sleep 90s ## it went to bg
[root@ljhamb ~]# fg %1 ## brought it back to fg using job id
sleep 90s

Description of fields of ps -f command

[root@ljhamb scripts]# ps -f
UID   PID PPID C STIME TTY   TIME CMD
root   59405 59361 0 07:50 pts/3   00:00:00 -bash
root   64467 59405 0 09:00 pts/3   00:00:00 sleep 10s
root   64650 59405 0 09:03 pts/3   00:00:00 /bin/bash ./job.sh
root   64651 64650 0 09:03 pts/3   00:00:00 sleep 60s
root   66385 59405 0 09:27 pts/3   00:00:00 ps -f

UID - User who ran the process
PID - Prpcess ID

PPID - Parent process ID

C - CPU utilization of the process

STIME - Starting time of process

TTY - termnal --> tty command can show u the terminal

TIME - CPU time taken by process

CMD - command that started the process

Stop a process

Kill

Ctrl+C

Usage of kill

--

kill -9 <PID>

Initially we ran a process called forefox - It opened up a browser

--

[root@ljhamb ~]# pidof firefox ##we found relevant process IDs for the initiated process

67782 67621 67560 67535 67266

[root@ljhamb ~]# kill -9 67782 ##killed the process using SIGKILL

[root@ljhamb ~]# kill -9 67621 ##killed the process using SIGKILL

[root@ljhamb ~]# kill -9 67560 ##killed the process using SIGKILL

[root@ljhamb ~]# kill -9 67535 ##killed the process using SIGKILL

[root@ljhamb ~]# kill -9 67266 ##killed the process using SIGKILL

List of all the kill signals

[root@ljhamb ~]# kill -l

1) SIGHUP 2) SIGINT 3) SIGQUIT 4) SIGILL 5) SIGTRAP

6) SIGABRT 7) SIGBUS 8) SIGFPE 9) SIGKILL** 10) SIGUSR1

11) SIGSEGV 12) SIGUSR2 13) SIGPIPE 14) SIGALRM 15) SIGTERM

16) SIGSTKFLT 17) SIGCHLD 18) SIGCONT 19) SIGSTOP 20) SIGTSTP

21) SIGTTIN 22) SIGTTOU 23) SIGURG 24) SIGXCPU 25) SIGXFSZ

26) SIGVTALRM 27) SIGPROF 28) SIGWINCH 29) SIGIO 30) SIGPWR

31) SIGSYS 34) SIGRTMIN 35) SIGRTMIN+1 36) SIGRTMIN+2 37) SIGRTMIN+3

38) SIGRTMIN+4 39) SIGRTMIN+5 40) SIGRTMIN+6 41) SIGRTMIN+7 42) SIGRTMIN+8

43) SIGRTMIN+9 44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13

48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12

53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9 56) SIGRTMAX-8 57) SIGRTMAX-7

58) SIGRTMAX-6 59) SIGRTMAX-5 60) SIGRTMAX-4 61) SIGRTMAX-3 62) SIGRTMAX-2

63) SIGRTMAX-1 64) SIGRTMAX

Killing the process if it has multiple PIDs

[root@ljhamb ~]# var=$(pidof firefox)

[root@ljhamb ~]# echo "$var"

68889 68819 68773 68675

[root@ljhamb ~]# echo "$var"; for i in "$var";do kill -9 $i;done

68889 68819 68773 68675

Parent and Child Process

Each process has two IDs in linux

- pid

- ppid

Zombie and Orphan processes

Orphan processes

--

Normally children exit and the parent exits

Now ques is - How the parent process comes to know that child process has exited?

Ans - When a child process is killed, the parent process is updated with SIGCHLD signal.

However, if the parent process is killed before its child is killed, then the child process are called as orphan processes.

Whenever a process gets executed, the process entry is removed from the process table/process db

Zombie

--

But if the process is executed/dead/killed but it's entry is not removed from process table/db - this is called as a zombie process.

--

Ques - Explain the second field in ps -elf out put?

Ans. - The second filed in ps -elf is: S

S The state of the process. This column uses the following codes:

D Uninterruptible sleep

R Running

S Interruptible sleep

T Stopped or traced

Z Zombie

DAY 7 OS

Job ID Versus Process ID

Background and suspended processes are usually manipulated via job number (job ID). This number is different from the process ID and is used because it is shorter.

In addition, a job can consist of multiple processes running in a series or at the same time, in parallel. Using the job ID is easier than tracking individual processes.

How to run the c program in bash

[root@ljhamb scripts]# cat hello.c

#include <stdio.h>

int main()

{

printf("hello world\n");

return 0;

}

[root@ljhamb scripts]# gcc hello.c -o hello.sh

[root@ljhamb scripts]# ./hello.sh

hello world

[Processes]

fork() - is used to create the process

Fork system call is used for creating a new process, which is called child process, which runs concurrently with the process that makes the fork() call (parent process). After a new child process is created, both processes will execute the next instruction following the fork() system call. A child process uses the same pc(program counter), same CPU registers, same open files which use in the parent process

Ques

--

1. Calculate number of times hello is printed:

```
#include <stdio.h>
#include <sys/types.h>
int main()
{
   fork();
   fork();
   fork();
   printf("hello\n");
   return 0;
}
```

2. Predict the Output of the following program:.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main() {
```

```
    // make two process which run same
    // program after this instruction
    fork();
    printf("Hello world!\n");
    return 0;
}
```

How to find my process

Run [root@centos7 ~]# sleep 100s -- in one shell and find details of it in the other shell

--

[root@centos7 ~]# ps -elf | grep "sleep"

0 S root 4857 3111 0 80 0 - 27013 hrtime 09:04 pts/1 00:00:00 sleep 100s

Initiated the process "sleep 90s" and then tried to find the pid of it

--

[root@centos7 ~]# ps -elf | grep sleep

0 S root 5086 3267 0 80 0 - 27013 hrtime 09:12 pts/2 00:00:00 sleep 90s

0 S root 5094 796 0 80 0 - 27013 hrtime 09:12 ? 00:00:00 sleep 60

0 S root 5096 3111 0 80 0 - 28202 pipe_w 09:12 pts/1 00:00:00 grep --color=auto sleep

Once pid was found - we gave a signal to it - here signal was to kill i.e 9

[root@centos7 ~]# kill -9 5086

[root@centos7 ~]# ps -elf | grep sleep

0 S root 5094 796 0 80 0 - 27013 hrtime 09:12 ? 00:00:00 sleep 60

0 R root 5098 3111 0 80 0 - 28202 - 09:12 pts/1 00:00:00 grep --color=auto sleep

[root@centos7 ~]# sleep 90s

Killed

Signalling a process [Process management]

--

[root@centos7 ~]# kill -l

1) SIGHUP 2) SIGINT 3) SIGQUIT 4) SIGILL 5) SIGTRAP

6) SIGABRT 7) SIGBUS 8) SIGFPE 9) SIGKILL 10) SIGUSR1

11) SIGSEGV 12) SIGUSR2 13) SIGPIPE 14) SIGALRM 15) SIGTERM

16) SIGSTKFLT 17) SIGCHLD 18) SIGCONT 19) SIGSTOP 20) SIGTSTP

21) SIGTTIN 22) SIGTTOU 23) SIGURG 24) SIGXCPU 25) SIGXFSZ

26) SIGVTALRM 27) SIGPROF 28) SIGWINCH 29) SIGIO 30) SIGPWR

31) SIGSYS 34) SIGRTMIN 35) SIGRTMIN+1 36) SIGRTMIN+2 37) SIGRTMIN+3

38) SIGRTMIN+4 39) SIGRTMIN+5 40) SIGRTMIN+6 41) SIGRTMIN+7 42) SIGRTMIN+8

43) SIGRTMIN+9 44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13

48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12

53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9 56) SIGRTMAX-8 57) SIGRTMAX-7

58) SIGRTMAX-6 59) SIGRTMAX-5 60) SIGRTMAX-4 61) SIGRTMAX-3 62) SIGRTMAX-2

63) SIGRTMAX-1 64) SIGRTMAX

Hierarchical structure of the processes can be seen by 'pstree'

--

[root@centos7 ~]# pstree

Killing the process

[root@centos7 ~]# pgrep chronyd

763

[root@centos7 ~]# pidof chronyd

763

[root@centos7 ~]# kill -9 763

[root@centos7 ~]#

Foreground and Background processes

[root@centos7 edac]# touch file1; mkdir dir1 --> we ran two commands in sequence and they got executed successfully as the other command didnt have to wait long for the first command's execution

[root@centos7 edac]# ls

dir1 file1

[root@centos7 edac]# sleep 1000s& touch file2 --> Now we know that second command will have to wauit for 1000s - so in order to avoid that we sent first command to background using '&'

[1] 5688

[root@centos7 edac]# ls

dir1 file1 file2

[root@centos7 edac]# jobs -->>> is to check the jobs

[1]+ Running   sleep 1000s & ->> job id can be seen in []

[root@centos7 edac]# fg %1 ------->> syntax to bring the process in fg

sleep 1000s

Arrays in Linux

[root@localhost ~]# linux_arr=(vipin imran) -> declaring

[root@localhost ~]# linux_arr[0]=vipin --> initialising

[root@localhost ~]# echo ${linux_arr[*]}

vipin imran

[root@localhost ~]# echo ${linux_arr[1]}

imran

[root@localhost ~]# echo ${linux_arr[0]}

vipin

Usage of arrays in loops

[root@localhost ~]# for i in "${linux_arr[@]}"; do echo "$i";done

vipin

imran

[root@localhost ~]# echo "${linux_arr[@]}"

vipin imran

[root@localhost ~]# for i in "${linux_arr[@]}"; do echo $i;done

vipin

imran

[root@localhost ~]# for i in "${linux_arr[*]}"; do echo $i;done

vipin imran

[root@localhost ~]# for i in "${linux_arr[*]}"; do echo "$i";done

vipin imran

Sticky Bits

A sticky bit is permission given to file/dir that lets only the owner of the file (or root) to delete/rename the file

root@ubuntu:~/edac#

root@ubuntu:~/edac# su - vipin

vipin@ubuntu:~$ cd /root/edac/test/

vipin@ubuntu:/root/edac/test$ touch file1

vipin@ubuntu:/root/edac/test$ exit

logout

root@ubuntu:~/edac# su - harshit

harshit@ubuntu:~$ cd /root/edac/test/

harshit@ubuntu:/root/edac/test$ touch file2

harshit@ubuntu:/root/edac/test$ exit

logout

root@ubuntu:~/edac# cd test/

root@ubuntu:~/edac/test# ls -l

total 0

-rw-rw-r-- 1 vipin   vipin   0 May 12 19:47 file1

-rw-rw-r-- 1 harshit harshit 0 May 12 19:47 file2

root@ubuntu:~/edac/test# su - vipin

vipin@ubuntu:~$ cd /root/edac/test/

vipin@ubuntu:/root/edac/test$ rm -rf file2

vipin@ubuntu:/root/edac/test$ ls -l

total 0

-rw-rw-r-- 1 vipin vipin 0 May 12 19:47 file1

vipin@ubuntu:/root/edac/test$ exit

logout

root@ubuntu:~/edac/test# chmod -R +t /root/edac/test/

root@ubuntu:~/edac/test# su - harshit

harshit@ubuntu:~$ cd /root/edac/test/

harshit@ubuntu:/root/edac/test$ ls -l

total 0

-rw-rw-r-T 1 vipin vipin 0 May 12 19:47 file1

harshit@ubuntu:/root/edac/test$ rm -rf file1

rm: cannot remove 'file1': Operation not permitted

How to trace system calls made by a process with strace on Linux

In order to inspect what a running application is doing under the hood, and what system calls it is performing during its execution, we can use the 'strace' utility

strace is a tool used to keep track of the system calls made by a running process and the signals received by it. System calls are the fundamental interface between an application and the Linux kernel; when we use strace, the name of the calls made by a process, along with their arguments and return values are displayed on stderr

Each line in the strace output contains:

The system call name

The arguments passed to the system call in parentheses

The system call return value

```
[root@centos7 edac]# echo "Hello World" > file_a
[root@centos7 edac]# touch file_b
[root@centos7 edac]# strace cp file_a file_b
execve("/usr/bin/cp", ["cp", "file_a", "file_b"], 0x7ffcb3021d70 /* 27 vars */) = 0
brk(NULL)                               = 0xbad000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f5b4b050000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=86240, ...}) = 0
mmap(NULL, 86240, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f5b4b03a000
close(3)                                = 0
open("/lib64/libselinux.so.1", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\220j\0\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=155744, ...}) = 0
mmap(NULL, 2255216, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f5b4ac09000
mprotect(0x7f5b4ac2d000, 2093056, PROT_NONE) = 0
mmap(0x7f5b4ae2c000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x23000) = 0x7f5b4ae2c000
mmap(0x7f5b4ae2e000, 6512, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f5b4ae2e000
close(3)                                = 0
open("/lib64/libacl.so.1", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0p\37\0\0\0\0\0\0"..., 832) = 832
```

fstat(3, {st_mode=S_IFREG|0755, st_size=37064, ...}) = 0

mmap(NULL, 2130560, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f5b4aa00000

mprotect(0x7f5b4aa07000, 2097152, PROT_NONE) = 0

mmap(0x7f5b4ac07000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x7000) = 0x7f5b4ac07000

close(3) = 0

open("/lib64/libattr.so.1", O_RDONLY|O_CLOEXEC) = 3

read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\320\23\0\0\0\0\0\0"..., 832) = 832

fstat(3, {st_mode=S_IFREG|0755, st_size=19896, ...}) = 0

mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f5b4b039000

mmap(NULL, 2113904, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f5b4a7fb000

mprotect(0x7f5b4a7ff000, 2093056, PROT_NONE) = 0

mmap(0x7f5b4a9fe000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x3000) = 0x7f5b4a9fe000

close(3) = 0

open("/lib64/libc.so.6", O_RDONLY|O_CLOEXEC) = 3

read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0`&\2\0\0\0\0\0"..., 832) = 832

fstat(3, {st_mode=S_IFREG|0755, st_size=2156240, ...}) = 0

mmap(NULL, 3985920, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f5b4a42d000

mprotect(0x7f5b4a5f0000, 2097152, PROT_NONE) = 0

mmap(0x7f5b4a7f0000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1c3000) = 0x7f5b4a7f0000

mmap(0x7f5b4a7f6000, 16896, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f5b4a7f6000

close(3) = 0

open("/lib64/libpcre.so.1", O_RDONLY|O_CLOEXEC) = 3

read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\360\25\0\0\0\0\0\0"..., 832) = 832

fstat(3, {st_mode=S_IFREG|0755, st_size=402384, ...}) = 0

mmap(NULL, 2494984, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f5b4a1cb000

mprotect(0x7f5b4a22b000, 2097152, PROT_NONE) = 0

mmap(0x7f5b4a42b000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x60000) = 0x7f5b4a42b000

close(3) = 0

open("/lib64/libdl.so.2", O_RDONLY|O_CLOEXEC) = 3

read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P\16\0\0\0\0\0\0"..., 832) = 832

fstat(3, {st_mode=S_IFREG|0755, st_size=19248, ...}) = 0

mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f5b4b038000

mmap(NULL, 2109744, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f5b49fc7000

mprotect(0x7f5b49fc9000, 2097152, PROT_NONE) = 0

mmap(0x7f5b4a1c9000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x2000) = 0x7f5b4a1c9000

close(3) = 0

open("/lib64/libpthread.so.0", O_RDONLY|O_CLOEXEC) = 3

read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\200m\0\0\0\0\0\0"..., 832) = 832

fstat(3, {st_mode=S_IFREG|0755, st_size=142144, ...}) = 0

mmap(NULL, 2208904, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f5b49dab000

mprotect(0x7f5b49dc2000, 2093056, PROT_NONE) = 0

mmap(0x7f5b49fc1000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x16000) = 0x7f5b49fc1000

mmap(0x7f5b49fc3000, 13448, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f5b49fc3000

close(3) = 0

mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f5b4b037000

mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f5b4b035000

arch_prctl(ARCH_SET_FS, 0x7f5b4b035840) = 0

mprotect(0x7f5b4a7f0000, 16384, PROT_READ) = 0

mprotect(0x7f5b49fc1000, 4096, PROT_READ) = 0

mprotect(0x7f5b4a1c9000, 4096, PROT_READ) = 0

mprotect(0x7f5b4a42b000, 4096, PROT_READ) = 0

mprotect(0x7f5b4a9fe000, 4096, PROT_READ) = 0

mprotect(0x7f5b4ac07000, 4096, PROT_READ) = 0

mprotect(0x7f5b4ae2c000, 4096, PROT_READ) = 0

mprotect(0x623000, 4096, PROT_READ) = 0

mprotect(0x7f5b4b051000, 4096, PROT_READ) = 0

munmap(0x7f5b4b03a000, 86240) = 0

set_tid_address(0x7f5b4b035b10) = 6618

set_robust_list(0x7f5b4b035b20, 24)   = 0

rt_sigaction(SIGRTMIN, {sa_handler=0x7f5b49db1860, sa_mask=[], sa_flags=SA_RESTORER|SA_SIGINFO, sa_restorer=0x7f5b49dba630}, NULL, 8) = 0

rt_sigaction(SIGRT_1, {sa_handler=0x7f5b49db18f0, sa_mask=[], sa_flags=SA_RESTORER|SA_RESTART|SA_SIGINFO, sa_restorer=0x7f5b49dba630}, NULL, 8) = 0

rt_sigprocmask(SIG_UNBLOCK, [RTMIN RT_1], NULL, 8) = 0

getrlimit(RLIMIT_STACK, {rlim_cur=8192*1024, rlim_max=RLIM64_INFINITY}) = 0

statfs("/sys/fs/selinux", {f_type=SELINUX_MAGIC, f_bsize=4096, f_blocks=0, f_bfree=0, f_bavail=0, f_files=0, f_ffree=0, f_fsid={val=[0, 0]}, f_namelen=255, f_frsize=4096, f_flags=ST_VALID|ST_RELATIME}) = 0

statfs("/sys/fs/selinux", {f_type=SELINUX_MAGIC, f_bsize=4096, f_blocks=0, f_bfree=0, f_bavail=0, f_files=0, f_ffree=0, f_fsid={val=[0, 0]}, f_namelen=255, f_frsize=4096, f_flags=ST_VALID|ST_RELATIME}) = 0

stat("/sys/fs/selinux", {st_mode=S_IFDIR|0755, st_size=0, ...}) = 0

brk(NULL)   = 0xbad000

brk(0xbce000)   = 0xbce000

access("/etc/selinux/config", F_OK)   = 0

open("/usr/lib/locale/locale-archive", O_RDONLY|O_CLOEXEC) = 3

fstat(3, {st_mode=S_IFREG|0644, st_size=106172832, ...}) = 0

mmap(NULL, 106172832, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f5b43869000

close(3)   = 0

geteuid()   = 0

stat("file_b", {st_mode=S_IFREG|0644, st_size=0, ...}) = 0

stat("file_a", {st_mode=S_IFREG|0644, st_size=12, ...}) = 0

stat("file_b", {st_mode=S_IFREG|0644, st_size=0, ...}) = 0

```
open("file_a", O_RDONLY)              = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=12, ...}) = 0
open("file_b", O_WRONLY|O_TRUNC)      = 4
fstat(4, {st_mode=S_IFREG|0644, st_size=0, ...}) = 0
fadvise64(3, 0, 0, POSIX_FADV_SEQUENTIAL) = 0
read(3, "Hello World\n", 65536)       = 12
write(4, "Hello World\n", 12)         = 12
read(3, "", 65536)                    = 0
close(4)                              = 0
close(3)                              = 0
lseek(0, 0, SEEK_CUR)                 = -1 ESPIPE (Illegal seek)
close(0)                              = 0
close(1)                              = 0
close(2)                              = 0
exit_group(0)                         = ?
+++ exited with 0 +++
```

The first system call we can see in the output is execve.

This call is used to execute a program with a specified array of arguments. The first argument accepted by execv is the path of the file we want to execute; the second is an array of strings which represents the arguments that will be passed to the program

Let's understand this - execve("/usr/bin/cp", ["cp", "file_a", "file_b"], 0x7ffcb3021d70 /* 27 vars */) = 0

Here: the binary that is called is /usr/bin/cp, and the array of arguments passed to the call are: the name of the program (cp), the source and the destination paths

The /* 27 vars */ notation means that 27 variables where inherited from the calling process

Return value = 0

Filtering only specific system calls

==========

Use -e option followed by an expression which indicates what system calls should be traced

Usage:

To just check for execve calls

[root@centos7 edac]# strace -e execve cp file_a file_b

execve("/usr/bin/cp", ["cp", "file_a", "file_b"], 0x7ffce150e8c0 /* 27 vars */) = 0

+++ exited with 0 +++

To just check for read calls

[root@centos7 edac]# strace -e read cp file_a file_b

read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\220j\0\0\0\0\0\0"..., 832) = 832

read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0p\37\0\0\0\0\0\0"..., 832) = 832

read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\320\23\0\0\0\0\0\0"..., 832) = 832

read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0`&\2\0\0\0\0\0"..., 832) = 832

read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\360\25\0\0\0\0\0\0"..., 832) = 832

read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P\16\0\0\0\0\0\0"..., 832) = 832

read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\200m\0\0\0\0\0\0"..., 832) = 832

read(3, "Hello World\n", 65536) = 12

read(3, "", 65536) = 0

+++ exited with 0 +++

Note: read system call takes three arguments: the first is a file descriptor associated with the file that should be read; the second is the buffer into which the file should be read, and third is the number of bytes that should be read

Trace an existing and already running process

==========

strace -p <pid of the process> --> this is called attaching of strace to a process

Tracing signals

==========

We can trace the signals passed to the process if we attach strace to a process

<Demo of initiating top, attaching strace to top and killing top - each in a new shell>

Summary of the system calls

==========

Use 'strace -c'

++++++++++++++

Theory Concepts

++++++++++++++

Scheduling Algorithm examples

==========

Non-preemptive examples:

1. FCFS : First come First Serve

2. SJF : Shortest Job First

3. Priority Scheduling : Scheduled based on priority

Pre-emptive examples:

1. Round Robin

2. Shortest Remaining Time First

FIFO - First come first served

1. FIFO is non pre-emptive

2. The context switch happens only on process termination

3. No Starvation observed

Shortest Job First

1. Non pre-emptive algorithm

2. Total execution time must be known before execution

3. Starvation can be observed

Priority Scheduling

1. Dynamic priority Scheduling algorithm

2. Used in Real time O.S

3. Queue will be searched for the process closest to its deadline.

Round Robin

1. Pre-emptive algorithm

2. Employs Time-sharing algorithm

3. Gives each job its time-slice a.k.a quantum.

Shortest Remaining Time First

1. Pre-emptive algorithm

2. Pre-emptive version of Shortest Job First Algorithm

3. The process with least remaining time is executed first.

Thread

======

A thread of execution is often regarded as the smallest unit of processing that a scheduler works on.

A thread is also called a lightweight process

Threads enable true parallelism on multiple processor machines

Threads are created like normal tasks, with the exception that the clone() system call is passed flags corresponding to specific resources to be shared:

clone(CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND, 0);

Meaning of above Flags with clone()

CLONE_VM Parent and child share address space.

CLONE_FS Parent and child share filesystem information.

CLONE_FILES Parent and child share open files.

CLONE_SIGHAND Parent and child share signal handlers and blocked signals.

Note: Linux has support for hundreds to thousands of threads.

Process vs Thread

==========

Process is heavy weight or resource intensive.

Thread is light weight, taking lesser resources than a process.

Process switching needs interaction with operating system.

Thread switching does not need to interact with operating system.

In multiple processing environments, each process executes the same code but has its own memory and file resources.

All threads can share same set of open files, child processes.

If one process is blocked, then no other process can execute until the first process is unblocked.

While one thread is blocked and waiting, a second thread in the same task can run.

Multiple processes without using threads use more resources.

Multiple threaded processes use fewer resources.

In multiple processes each process operates independently of the others.

One thread can read, write or change another thread's data.

Advantages of Thread

==========

Threads minimize the context switching time.

Efficient communication.

It is more economical to create and context switch threads.

Threads allow utilization of multiprocessor architectures

How Linux OS treats threads

==========

Linux has a unique implementation of threads. To the Linux kernel, there is no concept of a thread.

Linux implements all threads as standard processes.

The Linux kernel does not provide any special scheduling semantics or data structures to represent threads. Instead, a thread is merely a process that shares certain resources with other processes.

Each thread has a unique task_struct and appears to the kernel as a normal process

Kernel Level Threads

==========

Kernel threads are processes that exist only in kernel space. Kernel threads can only be created by other kernel threads.

Lifecycle

You can create a kernel thread with the kthread_create() function. The thread will be created in an no-running state.

You can create and start a kernel thread with kthread_run().

Once started a kernel thread continues to exist until either it calls do_exit() or another thread calls kthread_stop()

Show threads per process

==========

We can count threads with the list of available sub directories inside /proc/<PID>/task/

For example to check some thread count --> ls /proc/$(pidof process)/task/

Running c program via Shell

==========

You would need the compiler - gcc

gcc --version

yum -y install gcc (Centos)

apt-get install gcc (Ubuntu)

==========

nano hello.c

```
#include <stdio.h>
int main()
{
  printf("hello world\n");
  return 0;
}
```

Compile

gcc hello.c -o hello

To execute program:

./hello

==========

Task

Write a script that checks for the number of users with userID greater than '1000' (using awk) in the Linux system and continuosuly monitors for security reasons and displays as soon as the count increases or decreases i.e if user is added/deleted.

Banker’s algorithm

Banker’s Algorithm is a deadlock avoidance algorithm. It is also used for deadlock detection. This algorithm tells that if any system can go into a deadlock or not by analyzing the currently allocated resources and the resources required by it in the future.

Bank example

--

Total amount -> Rs 1000

Account holders -> Rs 900

You go there for asking loan --> Rs 100

DAY 8 OS

Syntax of sed ## replacing contents of a file

---

sed 's|thing you want to replace|replace with what|g' filename

sed 's/replace me/replace with/g'

rename command

Rename the *.png to *.pri

[root@ljhamb scripts]# ls

Splunk_PC_App.png

Splunk_VM_App.png

[root@ljhamb scripts]# rename .png .pri *.png

[root@ljhamb scripts]# ls

Splunk_PC_App.pri

Splunk_VM_App.pri

Zombie state and importance of wait() system call

When a process is created in UNIX using fork() system call, the address space of the Parent process is replicated. If the parent process calls wait() system call, then the execution of parent is suspended until the child is terminated.

At the termination of the child, a ‘SIGCHLD’ signal is generated which is delivered to the parent by the kernel.

Parent, on receipt of ‘SIGCHLD’ stops the child process

Even though, the child is terminated, there is an entry in the process table corresponding to the child where the status is stored

When parent collects the status, this entry is deleted

But, if the parent decides not to wait for the child’s termination and it executes its subsequent task, then at the termination of the child, the exit status is not read.

Hence, there remains an entry in the process table even after the termination of the child. This state of the child process is known as the Zombie state.

Now run two Programs

First

```
#include<stdio.h>

#include<unistd.h>

#include<sys/wait.h>

#include<sys/types.h>

int main()

{
```

```
	int i;
	int pid = fork();
	if (pid == 0)
	{
		for (i=0; i<20; i++)
			printf("I am Child\n");
	}
	else
	{
		printf("I am Parent\n");
		while(1);
	}
}
```

Second

```
#include<stdio.h>
#include<unistd.h>
#include<sys/wait.h>
#include<sys/types.h>
int main()
{
	int i;
	int pid = fork();
	if (pid==0)
	{
```

```
        for (i=0; i<20; i++)
            printf("I am Child\n");
    }
    else
    {
        wait(NULL);
        printf("I am Parent\n");
        while(1);
    }
}
```

(Here zombie is stopped due to wait)

Signals

Signals are software interrupts sent to a program to indicate that an important event has occurred. The events can vary from user requests to illegal memory access errors. Some signals, such as the interrupt signal, indicate that a user has asked the program to do something that is not in the usual flow of control.

OR

Signals are a limited form of inter-process communication (IPC), typically used in Unix, Unix-like, and other POSIX-compliant operating systems. A signal is used to notify a process of an synchronous or asynchronous event.

To list all the signals

---

kill -l

issue the kill -l command and it would display all the supported signals

[root@ljhamb scripts]# kill -l

1) SIGHUP 2) SIGINT 3) SIGQUIT 4) SIGILL 5) SIGTRAP

6) SIGABRT 7) SIGBUS 8) SIGFPE 9) SIGKILL 10) SIGUSR1

11) SIGSEGV 12) SIGUSR2 13) SIGPIPE 14) SIGALRM 15) SIGTERM

16) SIGSTKFLT 17) SIGCHLD 18) SIGCONT 19) SIGSTOP 20) SIGTSTP

21) SIGTTIN 22) SIGTTOU 23) SIGURG 24) SIGXCPU 25) SIGXFSZ

26) SIGVTALRM 27) SIGPROF 28) SIGWINCH 29) SIGIO 30) SIGPWR

31) SIGSYS 34) SIGRTMIN 35) SIGRTMIN+1 36) SIGRTMIN+2 37) SIGRTMIN+3

38) SIGRTMIN+4 39) SIGRTMIN+5 40) SIGRTMIN+6 41) SIGRTMIN+7 42) SIGRTMIN+8

43) SIGRTMIN+9 44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13

48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12

53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9 56) SIGRTMAX-8 57) SIGRTMAX-7

58) SIGRTMAX-6 59) SIGRTMAX-5 60) SIGRTMAX-4 61) SIGRTMAX-3 62) SIGRTMAX-2

63) SIGRTMAX-1 64) SIGRTMAX

Assignment - Difference between kill -6,-9 and -15.

kill -6 - SIGABRT : Abort process. ABRT is usually sent by the process itself, when it calls the abort() system call to signal an abnormal termination, but it can be sent from any process like any other signal. SIGIOT is a synonym for SIGABRT. (IOT stands for input/output trap, a signal which originated on the PDP-11.)

kill -9 : **SIGKILL** : Forcefully terminate a process. With STOP, this is one of two signals which cannot be intercepted, ignored, or handled by the process itself.

kill -15 : SIGTERM : The TERM signal is sent to a process to request its termination. Unlike the KILL signal, it can be caught and interpreted or ignored by the process.

This signal allows the process to perform nice termination releasing resources and saving state if appropriate. It should be noted that SIGINT is nearly identical to SIGTERM.

Signal Description (Some important signals)

SIGHUP Hang-up detected on controlling terminal or death of controlling process.

SIGINT Issued if the user sends an interrupt signal (Ctrl + C).

SIGQUIT Issued if the user sends a quit signal (Ctrl + D).

SIGKILL If a process gets this signal, it must quit immediately and will not perform any clean-up operations.

SIGTERM Software termination signal (sent kill by default).

SIGALRM Alarm clock signal (used for timers)

Every signal has a default action associated with it. The default action for a signal is the action that a script or program performs when it receives a signal.

Some of the possible default actions are −

- Terminate the process.

- Ignore the signal.

- Dump core. This creates a file called core containing the memory image of the process when it received the signal.

- Stop the process.

- Continue a stopped process.

Sending a signal

Synatx --> kill -signal pid

For ex: kill -9 1001

Sending Signals Using The Keyboard

The most common way of sending signals to processes is using the keyboard. There are certain key presses that are interpreted by the system as requests to send signals to the process with which we are interacting:

Ctrl-C

Pressing this key causes the system to send an INT signal (SIGINT) to the running process. By default, this signal causes the process to immediately terminate.

#strace sleep 30s

^CNULL) = ? ERESTART_RESTARTBLOCK (Interrupted by signal)

strace: Process 42421 detached

##Here we initiated the process and interrupted it with **ctrl+c**

Ctrl-Z

Pressing this key causes the system to send a TSTP signal (SIGTSTP) to the running process. By default, this signal causes the process to suspend execution.

#strace sleep 30s

##Here we interrupted the above process using ctr+z and it sent the process to bg

^Z

[1]+ Stopped strace sleep 30s

[root@ljhamb scripts]# jobs

[1]+ Stopped strace sleep 30s

Sending Signals Using System Calls

Another way of sending signals to processes is by using the kill system call. This is the normal way of sending a signal from one process to another. This system call is also used by the 'kill' command or by the 'fg' command. Here is an example code that causes a process to suspend its own execution by sending itself the STOP signal:

#include <unistd.h> /* standard unix functions, like getpid() */

#include <sys/types.h> /* various type definitions, like pid_t */

#include <signal.h> /* signal name macros, and the kill() prototype */

/* first, find my own process ID */

pid_t my_pid = getpid();

/* now that i got my PID, send myself the STOP signal. */

kill(my_pid, SIGSTOP);

An example of a situation when this code might prove useful, is inside a signal handler that catches the TSTP signal (Ctrl-Z, remember?) in order to do various tasks before actually suspending the process

Start the firefox and interrupt it in 10s

---

lavish@ubuntu:~$ timeout 10s firefox &

To know where the binary is located

Use which command

[root@ljhamb ~]# which ls

/usr/bin/ls

To see the exit status

[root@ljhamb scripts]# mkdir test1

[root@ljhamb scripts]# echo $?

0 ##previous command ran successfully

[root@ljhamb scripts]# mkdir test

mkdir: cannot create directory ‘test’: File exists

[root@ljhamb scripts]# echo $?

1 ##previous command had some error

############

System Calls

############

How to trace system calls made by a process with strace on Linux

---

In order to inspect what a running application is doing under the hood, and what system calls it is performing during its execution, we can use the 'strace' utility

strace is a tool used to keep track of the system calls made by a running process and the signals received by it. System calls are the fundamental interface between an application and the Linux kernel; when we use strace, the name of the calls made by a process, along with their arguments and return values are displayed on stderr

Each line in the strace output contains:

The system call name

The arguments passed to the system call in parentheses

The system call return value

Usage:

[root@centos7 edac]# echo "Hello World" > file_a

[root@centos7 edac]# touch file_b

[root@centos7 edac]# strace cp file_a file_b

Filtering only specific system calls

Use -e option followed by an expression which indicates what system calls should be traced

Usage:

To just check for execve calls

[root@ljhamb edac_os]# strace -e execve cp file_a file_b

execve("/usr/bin/cp", ["cp", "file_a", "file_b"], 0x7ffdf62fcea0 /* 27 vars */) = 0

+++ exited with 0 +++

[root@ljhamb edac_os]# strace -e write cp file_a file_b

write(4, "Hello World\n", 12)   = 12

+++ exited with 0 +++

Trace an existing and already running process

---

strace -p <pid of the process> --> this is called attaching of strace to a process

We can trace the signals passed to the process if we attach strace to a process

<Demo of initiating top, attaching strace to top and killing top - each in a new shell>

Summary of the system calls

Use 'strace -c'

[root@ljhamb ~]# strace -c -p <pid of process>

##Arrays in Linux

[root@localhost ~]# linux_arr=(vipin imran) -> declaring

[root@localhost ~]# linux_arr[0]=vipin --> initialising

[root@localhost ~]# echo ${linux_arr[*]}

vipin imran

[root@localhost ~]# echo ${linux_arr[1]}

imran

[root@localhost ~]# echo ${linux_arr[0]}

vipin

Usage of arrays in loops

[root@ljhamb scripts]# linux_arr=(rahul yash)

[root@ljhamb scripts]# echo ${linux_arr[0]}

rahul

[root@ljhamb scripts]# echo ${linux_arr[1]}

yash

[root@ljhamb scripts]# for i in "${linux_arr[@]}"; do echo "$i";done

rahul

yash

[root@ljhamb scripts]# for i in "${linux_arr[*]}"; do echo "$i";done

rahul yash

##Thread

A thread of execution is often regarded as the smallest unit of processing that a scheduler works on.

A thread is also called a lightweight process

Threads enable true parallelism on multiple processor machines

Threads are created like normal tasks, with the exception that the clone() system call is passed flags corresponding to specific resources to be shared:

clone(CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND, 0);

Meaning of above Flags with clone()

CLONE_VM Parent and child share address space.

CLONE_FS Parent and child share filesystem information.

CLONE_FILES Parent and child share open files.

CLONE_SIGHAND Parent and child share signal handlers and blocked signals.

Note: Linux has support for hundreds to thousands of threads.

Process vs Thread

Process is heavy weight or resource intensive.

Thread is light weight, taking lesser resources than a process.

Process switching needs interaction with operating system.

Thread switching does not need to interact with operating system.

In multiple processing environments, each process executes the same code but has its own memory and file resources.

All threads can share same set of open files, child processes.

If one process is blocked, then no other process can execute until the first process is unblocked.

While one thread is blocked and waiting, a second thread in the same task can run.

Multiple processes without using threads use more resources.

Multiple threaded processes use fewer resources.

In multiple processes each process operates independently of the others.

One thread can read, write or change another thread's data.

Advantages of Thread

==========

Threads minimize the context switching time.

Efficient communication.

It is more economical to create and context switch threads.

Threads allow utilization of multiprocessor architectures

Types of Thread

Threads are implemented in following two ways

User Level Threads -- User managed threads

Kernel Level Threads -- Operating System managed threads acting on kernel, an operating system core

User Level Threads

In this case, application manages thread management kernel is not aware of the existence of threads. The application begins with a single thread and begins running in that thread.

Advantages

Thread switching does not require Kernel mode privileges.

User level thread can run on any operating system.

Scheduling can be application specific in the user level thread.

User level threads are fast to create and manage.

Kernel Level Threads

In this case, thread management done by the Kernel. There is no thread management code in the application area. Kernel threads are supported directly by the operating system. Any application can be programmed to be multithreaded. All of the threads within an application are supported within a single process.

Advantages

Kernel can simultaneously schedule multiple threads from the same process on multiple processes.

If one thread in a process is blocked, the Kernel can schedule another thread of the same process.

Kernel routines themselves can multithreaded.

Key Differences

#User level threads are faster to create and manage.

##Kernel level threads are slower to create and manage.

#Implementation is by a thread library at the user level.

##Operating system supports creation of Kernel threads.

#User level thread is generic and can run on any operating system.

##Kernel level thread is specific to the operating system.

#Multi-threaded application cannot take advantage of multiprocessing.

##Kernel routines themselves can be multithreaded.

How Linux OS treats threads

==========

Linux has a unique implementation of threads. To the Linux kernel, there is no concept of a thread.

Linux implements all threads as standard processes.

The Linux kernel does not provide any special scheduling semantics or data structures to represent threads. Instead, a thread is merely a process that shares certain resources with other processes.

Each thread has a unique task_struct and appears to the kernel as a normal process

Kernel Level Threads

==========

Kernel threads are processes that exist only in kernel space. Kernel threads can only be created by other kernel threads.

LifeCycle

You can create a kernel thread with the kthread_create() function. The thread will be created in a no-running state.

You can create and start a kernel thread with kthread_run().

Once started a kernel thread continues to exist until either it calls do_exit() or another thread calls kthread_stop()

Show threads per process

==========

We can count threads with the list of available sub directories inside /proc/<PID>/task/

For example to check some thread count --> ls /proc/$(pidof process)/task/

What did we learn?

Threads/ Processes are the mechanism by which you can run multiple code segments at a time, threads appear to run concurrently; the kernel schedules them asynchronously, interrupting each thread from time to time to give others chance to execute.

Identifying a thread

Each thread identified by an ID, which is known as Thread ID. Thread ID is quite different from Process ID. A Thread ID is unique in the current process, while a Process ID is unique across the system.

Thread ID is represented by type pthread_t

Thread Identification

Just as a process is identified through a process ID, a thread is identified by a thread ID. But interestingly, the similarity between the two ends here.

A process ID is unique across the system where as a thread ID is unique only in context of a single process.

A process ID is an integer value but the thread ID is not necessarily an integer value. It could well be a structure

A process ID can be printed very easily while a thread ID is not easy to print.

The above points give an idea about the difference between a process ID and thread ID.

Thread ID is represented by the type ‘pthread_t’.

Function that can compare two thread IDs

#include <pthread.h>

int pthread_equal(pthread_t tid1, pthread_t tid2);

The above function takes two thread IDs and returns nonzero value if both the thread IDs are equal or else it returns zero.

Function to know thread's own thread Id

---

function ‘pthread_self()’ is used by a thread for printing its own thread ID.

#include <pthread.h>

pthread_t pthread_self(void);

Thread Creation

Normally when a program starts up and becomes a process, it starts with a default thread. So we can say that every process has at least one thread of control. A process can create extra threads using the following function :

#include <pthread.h>

int pthread_create(pthread_t *restrict tidp, const pthread_attr_t *restrict attr, void *(*start_rtn)(void), void *restrict arg)

The above function requires four arguments:

- The first argument is a pthread_t type address. Once the function is called successfully, the variable whose address is passed as first argument will hold the thread ID of the newly created thread.

- The second argument may contain certain attributes which we want the new thread to contain. It could be priority etc.

- The third argument is a function pointer. This is something to keep in mind that each thread starts with a function and that functions address is passed here as the third argument so that the kernel knows which function to start the thread from.

- As the function (whose address is passed in the third argument above) may accept some arguments also so we can pass these arguments in form of a pointer to a void type. Now, why a void type was chosen? This was because if a function accepts more than one argument then this pointer could be a pointer to a structure that may contain these arguments.

Example of thread creation

```
#include<stdio.h>
#include<string.h>
#include<pthread.h>
#include<stdlib.h>
#include<unistd.h>
pthread_t tid[2];
void* doSomeThing(void *arg)
{
    unsigned long i = 0;
    pthread_t id = pthread_self();
    if(pthread_equal(id,tid[0]))
    {
        printf("\n First thread processing\n");
    }
    else
    {
        printf("\n Second thread processing\n");
    }
    for(i=0; i<(0xFFFFFFFF);i++);
    return NULL;
}
int main(void)
{
```

```
    int i = 0;
    int err;
    while(i < 2)
    {
        err = pthread_create(&(tid[i]), NULL, &doSomeThing, NULL);
        if (err != 0)
            printf("\ncan't create thread :[%s]", strerror(err));
        else
            printf("\n Thread created successfully\n");
        i++;
    }
    sleep(5);
    return 0;
}
```

What it does

It uses the pthread_create() function to create two threads

The starting function for both the threads is kept same.

Inside the function ‘doSomeThing()’, the thread uses pthread_self() and pthread_equal() functions to identify whether the executing thread is the first one or the second one as created.

Also, Inside the same function ‘doSomeThing()’ a for loop is run so as to simulate some time consuming work.

How to compile

---

gcc file.c -o file.sh -lpthread

Run the shell file and See the output - threads get created and then start processing. The order of execution of threads is not always fixed. It depends on the OS scheduling algorithm.

Exiting a thread

---

pthread_exit() is used to exit a thread. This function is usually written at the end of the starting routine.

Waiting for a thread

A parent thread is made to wait for a child thread using wait signal.

DAY 9 OS

CPU Scheduling Algorithms

In Multiprogramming systems, the Operating system schedules the processes on the CPU to have the maximum utilization of it and this procedure is called CPU scheduling.

The main purposes of scheduling algorithms are to minimize resource starvation and to ensure fairness amongst the parties utilizing the resources.

- Maximum utilization of CPU so that we can keep the CPU as busy as possible.
- Throughput means the number of processes which are completing their execution in per unit time. There must be maximum throughput.
- Turnaround time means that the time taken by the processes to finish their implementation. It must be a minimum.
- Waiting time is that time for which the process remains in the ready queue. It must be a minimum.

- There must be fare allocation of CPU.
- Response time is the time when the process gives its first response. It must be a minimum.

Terminology

- Arrival time is the time at which the process arrives in the ready queue for execution, and it is given in our table when we need to calculate the average waiting time.
- Completion time is the time at which the process finishes its execution.
- Turnaround time is the difference between completion time and arrival time, i.e. turnaround-time = Completion time – arrival time
- Burst time is the time required by the process for execution of the process by CPU.
- Waiting time (W.T) is the difference between turnaround time and burst time, i.e. waiting time=
- Turnaround time – Burst time.

Types of Scheduling Algo

First Come First Serve (FCFS) Scheduling

In this scheduling, the process which arrives first in front of CPU will be executed first by the CPU. It is a non-preemptive type of scheduling algorithm, i.e. in this scheduling algorithm priority of processes does not matter, or you can say that whatever the priority of the process is, the process will be executed in the manner they arrived in front of the CPU.

Shortest Job First (SJF) scheduling algorithm

It is also called the Shortest Job Next (SJN) scheduling. It is both preemptive and non-preemptive. In this scheduling algorithm, the process which has the shortest burst time will be processed first by the CPU.

Shortest Remaining Time First (SRTF) scheduling algorithm

It is the preemptive mode of Shortest Job First (SJF) scheduling. In this algorithm, the process which has the short burst time is executed by the CPU. There is no need to have the same arrival time for all the processes. If another process was having the shortest burst time then the current process which is executing get stopped in between the execution, and the new arrival process will be executed first.

Priority based scheduling algorithm

This is the scheduling algorithm which is based on the priority of the processes. In priority scheduling, the scheduler itself chooses the priority of the task, and then they will be executed according to their preference which is assigned to them by the scheduler. The process which has the highest priority will be firstly processed by the CPU, and the process which has the lowest priority will be executed by the CPU when the current process terminates its execution.

Round Robin scheduling algorithm

Round robin scheduling is the preemptive scheduling algorithm. Here particular time slice is allocated to each process to which we can say as time quantum. Every process, which wants to execute itself, is present in the queue. CPU is assigned to the process for that time quantum. Now, if the process completed its execution in that quantum of time, then the process will get terminated, and if the process does not achieve its implementation, then the process will again be added to the ready queue, and the previous process will wait for its turn to complete its execution.

Highest Response Ratio Next (HRRN) scheduling algorithm

It is a non-preemptive scheduling algorithm which is done on the basis of a new parameter called Response Ratio (RR). We will calculate the response ratio of all the processes and the process which has the highest response ratio will be implemented first.
Response Ratio is given by
Response Ratio= (Waiting time+ Burst time)/ Burst time

Memory Management

Imp reads: https://tldp.org/LDP/tlk/mm/memory.html

Introduction

- The computer is able to change only that data which is present in the main memory. Therefore, every program we execute and every file we access must be copied from a storage device into main memory.
- All the programs are loaded in the main memory for execution
- Main memory is also known as RAM (Random Access Memory). This is volatile memory.

Memory Management is the process of coordinating and controlling the memory in a computer.

- This technique decides which process will get memory at what time.
- It also keeps the count of how much memory can be allocated to a process.
- It tracks when memory is freed or unallocated and updates the status.
- The memory management function keeps track of the status of each memory location, either allocated or free

- Subdividing memory to accommodate multiple processes
- Memory needs to be allocated to ensure a reasonable supply of ready processes to consume available processor time.

Assignment ques - Need of memory management?

Ans. - The following are the reasons we need memory management-

Relocation

When we generally work on a multiprogramming system, several processes are running in the background. It isn't possible for us to know in advance which other programs will reside in the main memory and when we'll execute our processes.

To solve this, the memory manager takes care of the executed and to be executed processes and allocates and frees up memory accordingly, making the execution of processes smooth and memory efficient.

Protection

With the execution of multiple processes, one process may write in the address space of another process. This is why every process must be protected against unwanted interference by any other process. The memory manager, in this situation, protects the address space of every single process. Keeping in mind the relocation algorithm too. The protection aspect and the relocation aspect of the memory manager work in synchronization.

Sharing

When multiple processes run in the main memory, it is required to have a protection mechanism that must allow several processes to access the same portion of the main memory. Allowing each process the access to the same memory or the identical copy of a program rather than having a copy for each program has an advantage of efficient memory allocation. Memory management allows controlled access to the shared memory without compromising the protection.

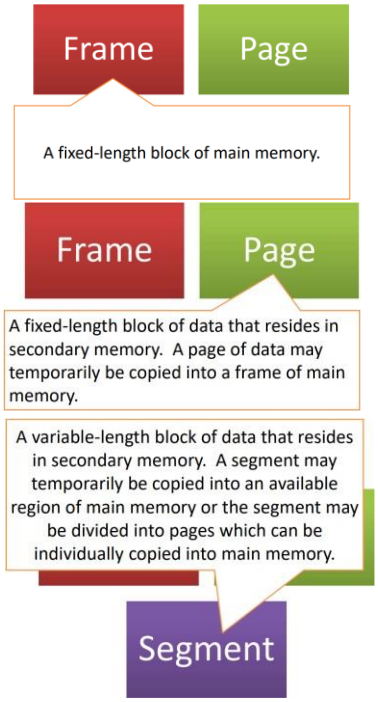
Logical Organization

Memory is a linear structure of storage that consists of some parts (of data), which can be modified along with those which can't be. The memory management allows the allocation, use, and access of memory to the user programs in a manner that does not make chaos by modifying some file which was not supposed to be accessed by the user. It supports a basic module that provides the required protection and sharing. The management modules are written and compiled independently so that all the references must be resolved by the system at run time. It provides different modules with different degrees of protection and also supports sharing based on the user specification.

Physical Organization

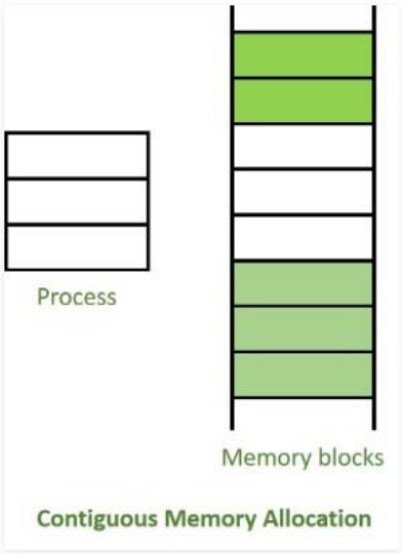
The structure of the memory consists of the volatile main memory and secondary non-volatile memory. Applications are stored in the secondary memory, which is the hard drive of your computer. But when you run an application, it moves to the main memory, the RAM of the system. To maintain the flow of these transfers from the main memory to the secondary memory with ease, proper management of memory is required.

Frame, Page and Segments



Contiguous memory allocation

In contiguous memory allocation, when a process requests for the memory, a single contiguous section of memory blocks is assigned to the process according to its requirement.



One of the simplest methods for allocating memory is to divide memory into several fixed-sized partitionsand each partition contains exactly one process.

Hole – block of available memory; holes of various size are scattered throughout memory When a process arrives, it is allocated memory from a hole large enough to accommodate it Operating system maintains information about:

a) allocated partitions b) free partitions (hole)

First-fit

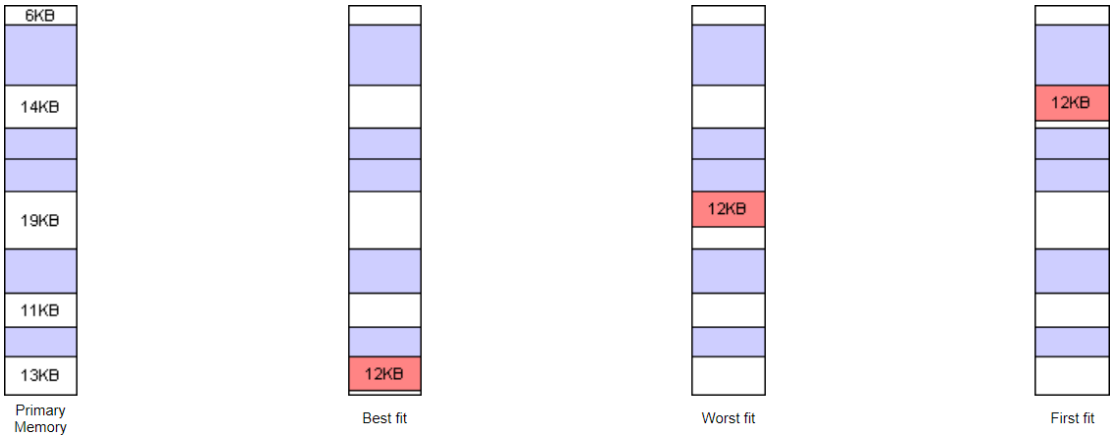
Allocate the first hole that is big enough

Best-fit

Allocate the smallest hole that is big enough; must search entire list, unless ordered by size. Produces thesmallest leftover hole.

Worst-fit

Allocate the largest hole; must also search entire list. Produces the largest leftover hole



Note: First-fit and best-fit better than worst-fit in terms of speed and storage utilization

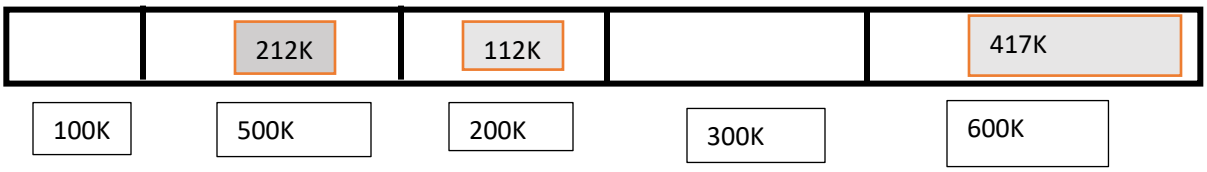
Notice in the diagram above that the Best fit and First fit strategies both leave a tiny segment of memory unallocated just beyond the new process. Since the amount of memory is small, it is not likely that any new processes can be loaded here. This condition of splitting primary memory into segments as the memory is allocated and deallocated is known as fragmentation. The Worst fit strategy attempts to reduce the problem of fragmentation by allocating the largest fragments to new processes.

Assignment: Given free memory partitions of 100K, 500K, 200K, 300K, and 600K (in order), how would each of the First-fit, Best-fit, and Worst-fit algorithms place processes of 212K, 417K, 112K, and 426K (in order)?

Which algorithm makes the most efficient use of memory?

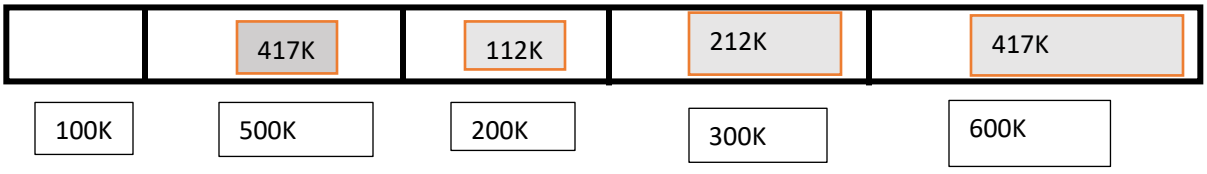
Ans.

Considering first-fit algorithm:



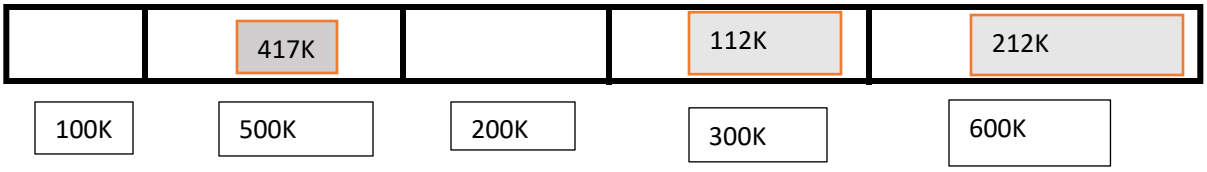
The first-fit algorithm does not accommodate process 426k as there is no partition left. This is a case of external fragmentation.

Considering Best-Fit algorithm



The Best-Fit algorithm can accommodate all the processes and there is no fragmentation.

Considering Worst-Fit algorithm:



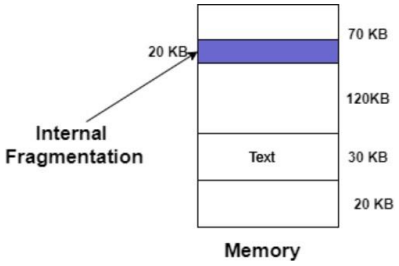
The Worst-Fit algorithm also does not accommodate process of 426k as there is no partition. This is a case of external fragmentation.
Therefore, Best-Fit algorithm is the best memory management algorithm for the above processes.

Fragmentation

A Fragmentation is defined as when the process is loaded and removed after execution from memory, it creates a small free hole. These holes cannot be assigned to new processes because holes are not combined or do not fulfill the memory requirement of the process.
It is of two types

- Internal fragmentation
- External fragmentation

Internal fragmentation



70 KB partition is used to load a process of 50 KB so the remaining 20 KB got wasted.
Suppose the size of the process is lesser than the size of the partition in that case some size of the partition gets wasted and remains unused. This wastage inside the memory is generally termed as Internal fragmentation.

External fragmentation

In external fragmentation, we have a free memory block, but we cannot assign it to process because blocks are not contiguous.

Example:
Suppose there is a fixed partitioning is used for memory allocation and the different size of block 3MB, 6MB, and 7MB space in memory
Three process p1, p2, p3 comes with size 2MB, 4MB, and 7MB respectively. Now they get memory blocks of size 3MB, 6MB, and 7MB allocated respectively. After allocating process p1 process and p2 process left 1MB and 2MB. Suppose a new process p4 comes and demands a 3MB block of memory, which is available, but we cannot assign it because free memory space is not contiguous. This is called external fragmentation.

Compaction

Reduce external fragmentation by compaction
Shuffle memory contents to place all free memory together in one large block Compaction is possible only if relocation is dynamic, and is done at execution time

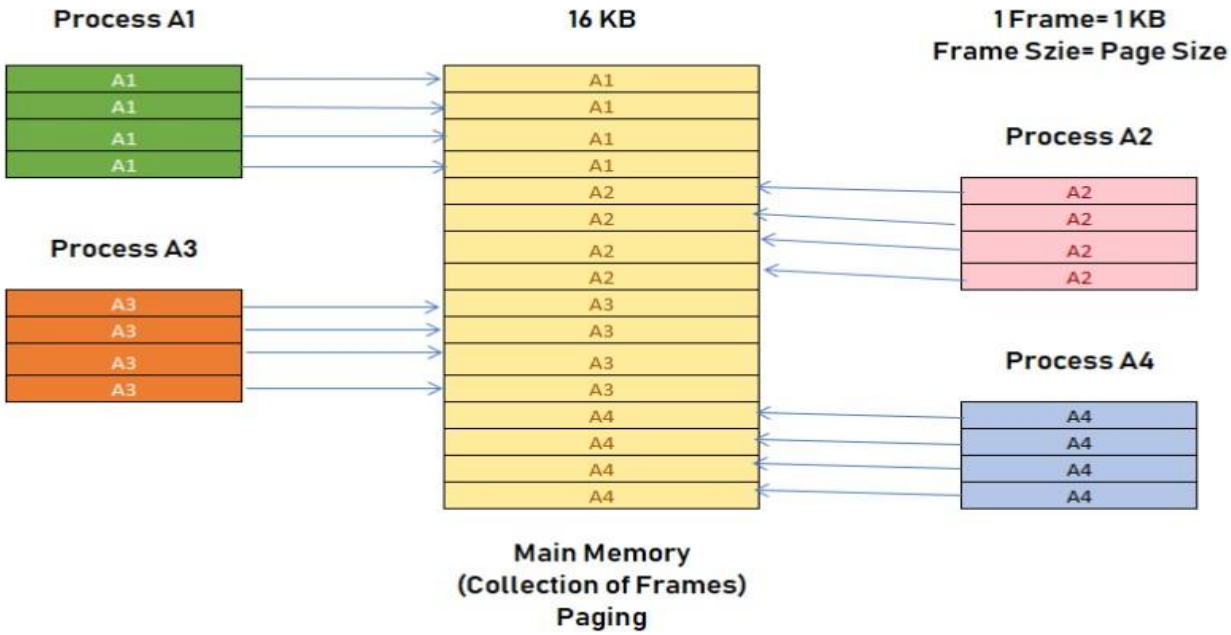
Paging

Paging is a storage mechanism that allows OS to retrieve processes from the secondary storage into the main memory in the form of pages. In the Paging method, the main memory is divided into small fixed- size blocks of physical memory, which is called frames. The size of a frame should be kept the same as that of a page to have maximum utilization of the main memory and to avoid external fragmentation.

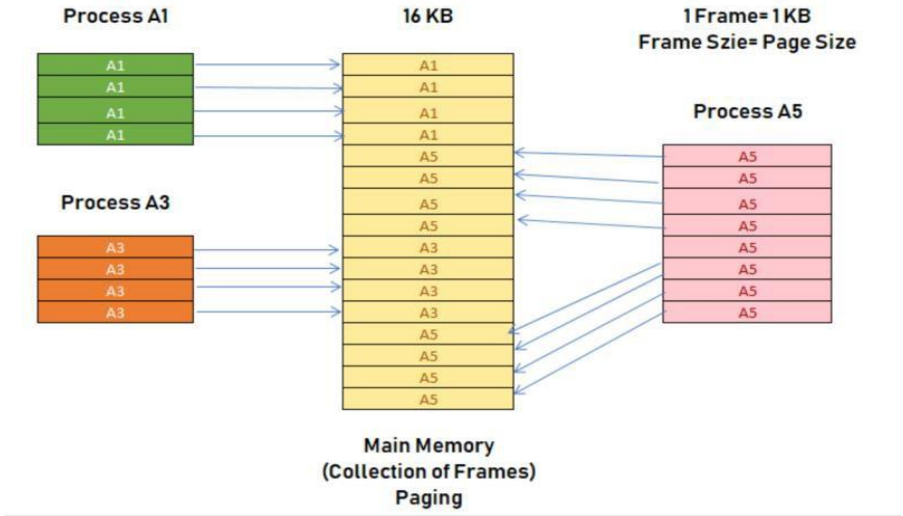
- Paging permits the physical address space of a process to be non-contiguous.
- Paging solves the problem of fitting memory chunks of varying sizes onto the backing store and this problem is suffered by many memory management schemes.
- Paging helps to avoid external fragmentation and the need for compaction.
- The mapping from virtual to physical address is done by the memory management unit (MMU) which is a hardware device and this mapping is known as the paging technique
- The Physical Address Space is conceptually divided into several fixed-size blocks, called frames.
- The Logical Address Space is also split into fixed-size blocks, called pages.
- Page Size = Frame Size.

Example

For example, if the main memory size is 16 KB and Frame size is 1 KB. Here, the main memory will be divided into the collection of 16 frames of 1 KB each.
There are 4 separate processes in the system that is A1, A2, A3, and A4 of 4 KB each. Here, all the processes are divided into pages of 1 KB each so that operating system can store one page in one frame.



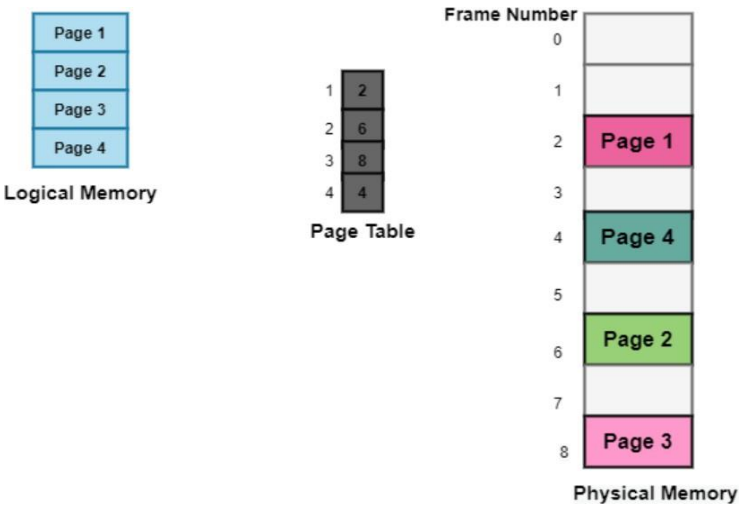
Let's say A2 and A4 are moved to the waiting state after some time and the process A5 of size 8 pages (8KB) are waiting in the ready queue.



Here, eight non-contiguous frames which is available in the memory, and paging offers the flexibility of storing the process at the different places.

Page table

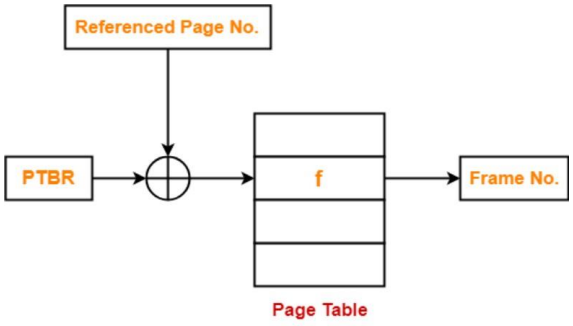
Page table mainly provides the corresponding frame number (base address of the frame) where that page is stored in the main memory i.e. it maps the page number referenced by the CPU to the frame number where that page is stored.



Some of the characteristics of the Page Table are as follows:

- It is stored in the main memory.
- Generally, the Number of entries in the page table = the Number of Pages in which the process is divided.
- Page Table Base Register (PTBR) provides the base address of the page table
- Each process has its own independent page table.

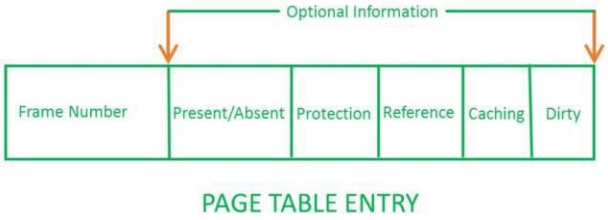
Working



Obtaining Frame Number Using Page Table

- Page Table Base Register (PTBR) provides the base address of the page table.
- The base address of the page table is added with the page number referenced by the CPU.
- It gives the entry of the page table containing the frame number where the referenced page is stored

This is how a PTE looks like:



PAGE TABLE ENTRY

This PTE will contain information like frame number (The address of main memory where we want to refer), and some other useful bits (e.g., valid/invalid bit, dirty bit, protection bit etc.)
Reasons for not having faster access in case of larger memory size:

1. To find the frame number
2. To go to the address specified by frame number

To overcome this problem a high-speed cache is set up for page table entries called a Translation Lookaside Buffer (TLB)

Translation Lookaside Buffer

Translation Lookaside Buffer (TLB) is nothing but a special cache used to keep track of recently used transactions. TLB contains page table entries that have been most recently used

Given a virtual address, the processor examines the TLB if a page table entry is present (TLB hit), the frame number is retrieved and the real address is formed. If a page table entry is not found in the TLB (TLB miss)

Translation Lookaside Buffer (TLB) consists of two columns-

1. Page Number
2. Frame Number

Page Number	Frame Number

Translation Lookaside Buffer

Dirty Bit

A dirty bit is a bit in memory switched on when an update is made to a page by computer hardware. When the dirty bit is switched on, the page is modified and can be replaced in memory. If it is off, no replacement is necessary since no updates have been made.

> The dirty bit is set to "1" by the hardware whenever the page is modified. (written into).
> When we select a victim by using a page replacement algorithm, we examine its dirty bit. If it is set, that means the page has been modified since it was swapped in. In this case we have to write that page into the backing store.
>
> However if the dirty bit is reset, that means the page has not been modified since it was swapped in, so we don't have to write it into the backing store. The copy in the backing store is valid.

Ques of CCEE

Dirty bit is used to indicate which of the following?

A. A page fault has occurred
B. A page has corrupted data
C. A page has been modified after being loaded into cache
D. An illegal access of page

CPU throttling

Adjusting the clock speed of the CPU. Also called "dynamic frequency scaling," CPU throttling is commonly used to automatically slow down the computer when possible to use less energy and conserve battery, especially in laptops. CPU throttling can also be adjusted manually to make the system quieter, because the fan can then run slower. Processor throttling is also known as "automatic underclocking". Automatic overclocking (boosting) is also technically a form of dynamic frequency scaling, but it's relatively new and usually not discussed with throttling.
Dynamic frequency scaling reduces the number of instructions a processor can issue in a given amount of time, thus reducing performance. Hence, it is generally used when the workload is not CPU-bound.

Re-entrant Function

A computer program or routine is described as re-entrant if it can be safely called again before its previous invocation has been completed (i.e. it can be safely executed concurrently).

Conditions for a re-entrant function or code

- It may not use global and static data
- It should not call another non-re-entrant function but it can call any re-entrant function.
- It should not modify its own code

```
// A non-re-entrant example

// [The function depends on

global variable i] int i;

// Both fun1() and fun2() are not re-entrant

//   fun1()   is   NOT   re-entrant

because it uses global variable i int

fun1()

    {
    return i * 5;
    }

// fun2() is NOT re-entrant because it calls a non-re-entrant

fun2()

{

return

fun1()

* 5;

}

// Both fun1 () and fun2 () are re-entrant

    int fun1(int i)
```

```
{
return i * 5;
}
int fun2(int i)
{
return fun1 (i) * 5;
}
```

Inter Process Communication

Most modern computer systems use the notion of a process that lets us execute multiple tasks at any time. And, as multiple processes execute at the same time, often they need to communicate with each other for various reasons.

Processes executing concurrently in a computer system are of two types –

- independent processes
- cooperating processes

A process is independent if it cannot affect or be affected by any other process executing in the computer system (does not share data with any other process)

A process is cooperating if it can affect or be affected by other processes executing in the computer system (shares data with other processes is a cooperating process)

The cooperating processes need to communicate with each other to exchange data and information. Inter-process communication is the mechanism of communicating between processes.

Modes of IPC

There are two modes through which processes can communicate with each other –

- shared memory
- message passing

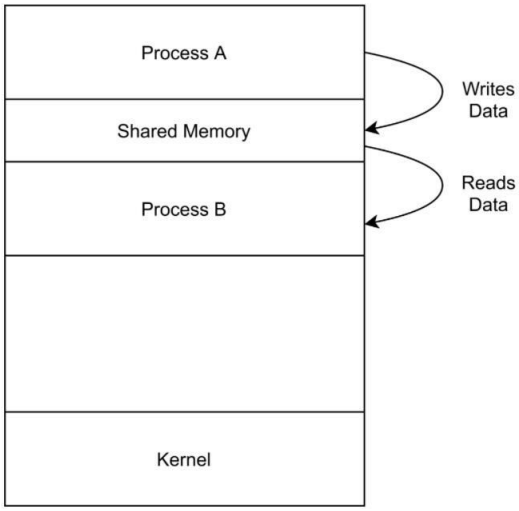
As the name suggests, the shared memory region shares a shared memory between the processes.

On the other hand, the message passing lets processes exchange information through messages.

Shared Memory

Intercrossed communication through the shared memory model requires communicating processes to establish a shared memory region. In general, the

process that wants to communicate creates the shared memory region in its own address space. Other processes that wish to communicate to this process need to attach their address space to this shared memory segment:



What did we learn?

- One process will create an area in RAM which the other process can access
- Both processes can access shared memory like a regular working memory
- Fast
- Limitation: Error prone. Needs synchronization between processes.

Shared Memory Functions Used for IPC

int shmget (key, size, flags)

- Create a shared memory segment;
- Returns ID of segment : shmid
- key : unique identifier of the shared memory segment
- size : size of the shared memory (rounded up to the PAGE_SIZE)

int shmat (shmid, addr, flags)

- Attach shmid shared memory to address space of the calling process
- addr : pointer to the shared memory address space

int shmdt (shmid)

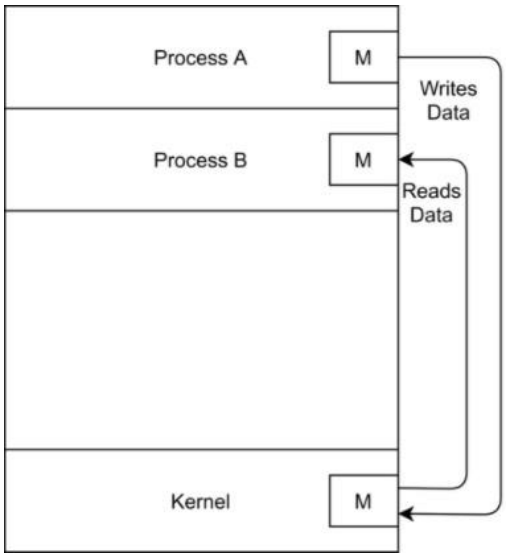
- Detach shared memory

Message Passing

In this mode, processes interact with each other through messages with assistance from the underlying operating system i.e. via kernel.

Here two processes A, and B are communicating with each other through message

passing. Process Asends a message M to the operating system (kernel). This message is then read by process B.



Other modes of IPC

- signals
- FIFOS (named pipes)
- Pipes

Signals

Signals allow for asynchronous communication between processes, but of a very limited nature. A signal communicates that an event has occurred, but nothing else. There's no opportunity for sending additional data concerning the event

Pipes

Pipes – direct the out stream of one process to feed the input of another process.

IPC is a very common mechanism in Linux and Pipe maybe one of the most widely used IPC methods. When you type cat file | grep bar, you create a pipe to connect stdout of cat to stdin of grep

A pipe, as its name states, can be understood as a channel with two ends.

Pipe is actually implemented using a piece of kernel memory. The system call pipe always create a pipe and two associated file descriptions, fd[0] for reading from the pipe and fd[1] for writing to the pipe.
Synatx in C for pipe system call – pipe()

int pipe(int fds[2]);

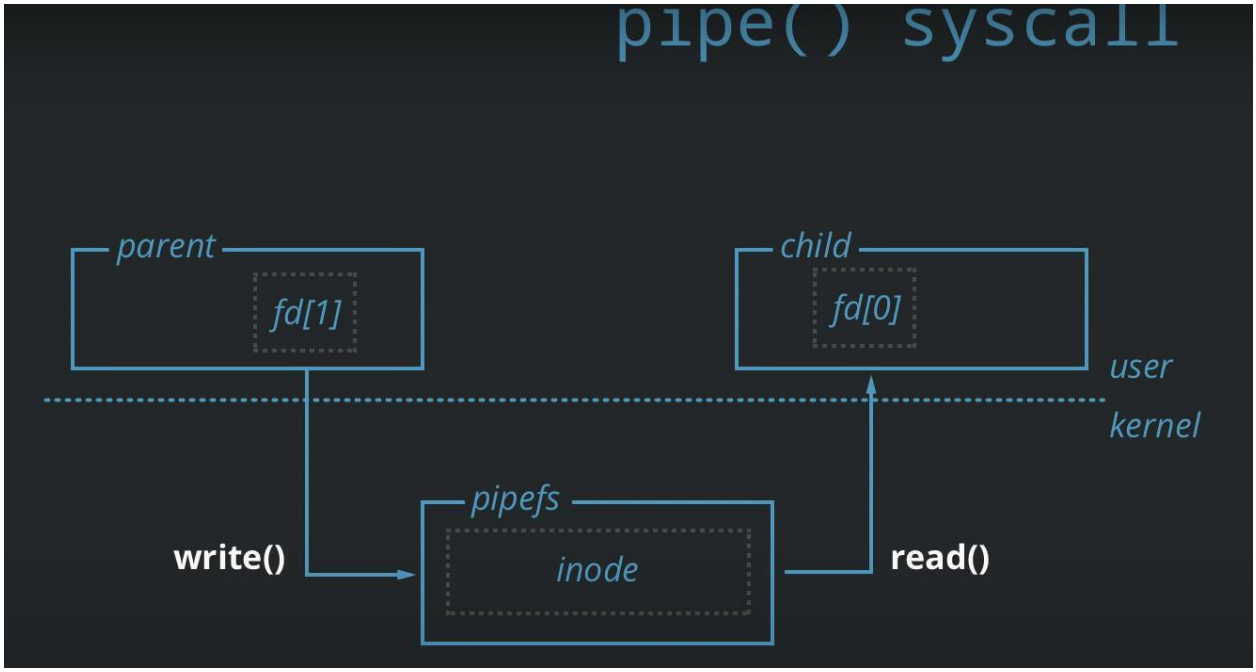
Parameters:

- fd[0] will be the fd(file descriptor) for the read end of pipe. So, fd[0] is argument of read() system call
- fd[1] will be the fd(file descriptor) for the write end of pipe. So, fd[1] is argument of write() system call
- These file descriptors are used for read and write operations
- Returns: 0 on Success
- -1 on error – to know the cause of error or failure – we can use perror() system call

After the pipe system call executes, the array pipefd [2] contains two file descriptors, pipefd [0] is for reading from the pipe and pipefd [1] is for writing to the pipe.

A *pipe* is a mechanism for interprocess communication; data written to the pipe by one process can be read by another process. The data is handled in a first-in, first-out (FIFO) order. The pipe has no name; it is created for one use and both ends must be inherited from the single process which created the pipe.

pipe() syscall to shows parent writes and child reads



FIFO Named Pipe

A *FIFO special file* is similar to a pipe, but instead of being an anonymous, temporary connection, a FIFO has a name or names like any other file. Processes open the FIFO by name in order to communicate through it.
A FIFO special file is similar to a pipe, except that it is created in a different way. Instead of being an anonymous communications channel, a FIFO special file is entered

into the file system by calling mkfifo.
The mkfifo function is declared in the header file

sys/stat.h. Function: int mkfifo (const char

*filename, mode_t mode)

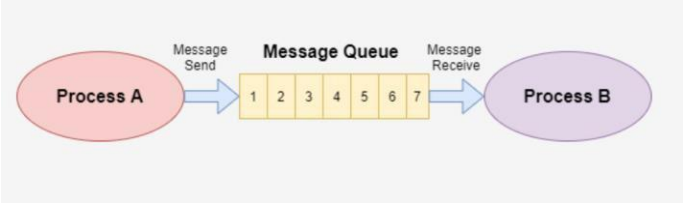
The mkfifo function makes a FIFO special file with name filename. The mode argument is used to set the
file's permissions
The normal, successful return value from mkfifo is 0. In the case of an error, -1 is returned.
In addition to the usual file name errors, the following error conditions are defined for this function:
EEXIST
The named file already exists.
ENOSPC
The directory or file system cannot be extended.
EROFS
The directory that would contain the file resides on a read-only file system.

```
[root@ljhamb edac_os]# mkfifo myfile
[root@ljhamb edac_os]# cal > myfile
^Z
[1]+  Stopped                 cal > myfile
[root@ljhamb edac_os]# bg 1
[1]+ cal > myfile &
[root@ljhamb edac_os]# cat myfile
    October 2021
Su Mo Tu We Th Fr Sa
                1  2
 3  4  5  6  7  8  9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
31
[1]+  Done                    cal > myfile
[root@ljhamb edac_os]# ls -l | grep myfile
prw-r--r--. 1 root root    0 Oct  2 20:20 myfile
[root@ljhamb edac_os]#
```

Message queues

Message queues are one of the inter-process communication mechanism which allows processes to exchange data in the form of messages between two processes. It allows processes to communicate asynchronously by sending messages to each other where the messages are stored in a queue, waiting to be processed, and are deleted after being processed.
IPC using message queue involves the following steps:

- Defining a Message Structure
- Creating the Message Queue
- Sending the Message to the Process A through the Message Queue
- Receiving the Message from the Process A through the Message Queue



Each message is given an identification or "type" so that processes can select the appropriate message
Process must share a common "key" in order to gain access to the queue

in the first place. System calls used for message queues

- ftok(): used to generate a unique key
- msgget() – creating message using a queue
- msgsnd() - sending a message to a queue
- msgrcv() – fetching a message from a queue
- msgctl() - controlling a queue.

// Program for Message Queue (Writer Process) -- client.c

```
#include <stdio.h>

#include <sys/ipc.h>

#include <sys/msg.h>

// message queue structure

struct mesgbuffer {

long mesg type;
```

```
char mesg text [100];
};
Message;
Int main() {
key tkey;
int mesgid;
// generate unique key
key = ftok(“somefile”,65);
// create a message queue and return identifier
msgid = msgget(key, 0666 | IPC_CREAT);
message.mesg type =1;
printf (“Insert message: “);
gets (message.mesg text);
// send message
msgsnd(msgid, &message, sizeof(message), 0);
// display the message
printf(“Message Sent to server: %s\n”, message.mesg text);
return 0; }
//Program for Message Queue (Reader Process) – server.c
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/mesg.h>
//Structure for message queue
struct mesg buffer {
long mesg type;
char mesg text[100];
}
message;
```

```
int main(){
key tkey;
int msgid;
// generate unique key
key = ftok("somefile",65);
// create a message queue and return identifier
msgid = mesget(key, 0666 | IPC_CREAT);
printf ("Waiting for a message from client….\n");
// receive message
msgrcv(msgid, &message, sizeof(message),1,0);
// display the message
printf ("Message received from client: %s\n", message.mesg text);
// to destroy the message queue
msgctl (msgid, IPC_RMID, NULL);
return 0;
}
```

Now task is to compile and run ./client and enter message to be sent to server and then run ./server and observe the output.

Semaphores

To avoid a series of issues caused by multiple processes (or programs) accessing a shared resource at the same time, we need a method that can be authorized by generating and using a token, so that only one execution thread can access the critical section of the code at any given time. The critical section is a code segment where the shared variables can be accessed, and the atomic action is required in this section.

Binary Semaphore (0 or 1) aka Mutexes

A semaphore is implemented as an integer variable with atomic increment and decrement operations; so long as the value is not negative the thread will continue, it will block otherwise.

- The increment operation is called V, or signal or wake-up or up
- the decrement is called P, or wait or sleep or down

wait(): decreases the counter by one; if the counter is negative, then it puts the thread on a queue and blocks.

signal(): increments the counter; wakes up one waiting

process.

The definition of wait operation is as follows:

```
wait(s)

{

   while (s==0); //no operation

   s=s-1;
```

The definition of signal operation is as follows:

```
signal(s)

{

s=s+1;
```

Note: Initial value of semaphore variable is 1
Example: The critical section of Process P is in between P and V operation.

Process
Code //Initial code of process

P(s); //here if value of s=0, then code will get stuck, else if it is 1(initial value), then it will be decremented to 0, and process will enter the critical section of code
CS; //critical section of the code
V(s); //after execution of critical section, the value 0 will again be incremented and

process exits the critical section of the code and moves to execute the remaining section of code

Remaining Section of the code //noncritical section of code

P1	S=1	Executing noncritical section of the code
P2		Executing noncritical section of the code
P1	S=0	Enters the critical section
P2		Executing noncritical section of the code
P1	S=0	Executing the critical section of the code
P2		Wants to enter critical section but can't because s=0
P1	S=1	Exits the critical section and increments s=1
P2		Enters the critical section and updates the s=0

It shows mutual exclusion. Here we have two processes P1 and P2 and a semaphore s is initialized as 1. Now if suppose P1 enters in its critical section then the value of semaphore s becomes 0. Now if P2 wants to enter its critical section then it will wait until s > 0, this can only happen when P1 finishes its critical section and calls V operation on semaphore s. This way mutual exclusion is achieved. Look at the below image for details which is Binary semaphore.

What did we learn?
Mutually exclusive because at one time only one process can execute the critical section of code

Counting Semaphore

A binary semaphore is restricted to values of zero or one, while a counting semaphore can assume any nonnegative integer value.

A counting semaphore has two components-

- An integer value
- An associated waiting list (usually a queue)

The value of counting semaphore may be positive or negative.

- Positive value indicates the number of processes that can be present in the critical section at the same time.
- Negative value indicates the number of processes that are blocked in the waiting list.
- The waiting list of counting semaphore contains the processes that got blocked when trying to enter the critical section.
- In waiting list, the blocked processes are put to sleep.
- The waiting list is usually implemented using a queue data structure.
- Using a queue as waiting list ensures bounded waiting.

- This is because the process which arrives first in the waiting queue gets the chance to enter the critical section first.
- The wait operation is executed when a process tries to enter the critical section.
- Wait operation decrements the value of counting semaphore by 1.

Then, following two cases are possible-

Counting Semaphore Value >= 0

- If the resulting value of counting semaphore is greater than or equal to 0, process is allowed to enter the critical section.

Counting Semaphore Value < 0

- If the resulting value of counting semaphore is less than 0, process is not allowed to enter the critical section.
- In this case, process is put to sleep in the waiting list.

- The signal operation is executed when a process takes exit from the critical section.
- Signal operation increments the value of counting semaphore by 1.

Then, following two cases are possible-

Counting Semaphore <= 0

- If the resulting value of counting semaphore is less than or equal to 0, it picks the process from blocked state and puts it in ready queue or we can say that - a process is chosen from the waiting list and wake up to execute.

Counting Semaphore > 0

- If the resulting value of counting semaphore is greater than 0, no action is taken.

- By adjusting the value of counting semaphore, the number of processes that can enter the critical section can be adjusted.
- If the value of counting semaphore is initialized with N, then maximum N processes can be present in the critical section at any given time.

What did we learn?

- Consider n units of a particular non-shareable resource are available.
- Then, n processes can use these n units at the same time.
- So, the access to these units is kept in the critical section.
- The value of counting semaphore is initialized with ‘n’.
- When a process enters the critical section, the value of counting semaphore

decrements by 1.

- When a process exits the critical section, the value of counting semaphore increments by 1.

Virtual Memory

Virtual memory is a feature of an operating system that enables a computer to be able to compensate shortages of physical memory by transferring pages of data from random access memory to disk storage. This means that when RAM runs low, virtual memory can move data from it to a space called a paging file. This process allows for RAM to be freed up so that a computer can complete the task. Virtual Memory is a storage scheme that provides user an illusion of having a very big main memory. This is done by treating a part of secondary memory as the main memory.

Example

Let's assume that an OS requires 300 MB of memory to store all the running programs. However, there's currently only 50 MB of available physical memory stored on the RAM. The OS will then set up 250 MB of virtual memory and use a program called the Virtual Memory Manager (VMM) to manage that 250 MB. So, in this case, the VMM will create a file on the hard disk that is 250 MB in size to store extra memory that is required. The OS will now proceed to address memory as it considers 300 MB of real memory stored in the RAM, even if only 50 MB space is available. It is the job of the VMM to manage 300 MB memory even if just 50 MB of real memory space is available. Swapping is the process the OS uses to move data between RAM and virtual memory.

Demand Paging

Demand Paging is a popular method of virtual memory management. In demand paging, the pages of a process which are least used, get stored in the secondary memory.
A page is copied to the main memory when its demand is made or page fault occurs. Hence, a demand paging system is quite similar to a paging system with swapping where processes reside in secondary memory and pages are loaded only on demand, not in advance.

What is a Page Fault?

Page fault is more like an error. It mainly occurs when any program tries to access the data or the code that is in the address space of the program, but that data is not currently located in the RAM of the system.
So basically when the page referenced by the CPU is not found in the main memory then the situation is termed as Page Fault.
Whenever any page fault occurs, then the required page has to be fetched from the secondary memory into the main memory.

Page Replacement Algorithm

Page replacement algorithms are the techniques using which an Operating System decides which memory pages to swap out, write to disk when a page of memory needs to be allocated

First In First Out (FIFO)

In this algorithm, the operating system keeps track of all pages in the memory in a queue, the oldest page is in the front of the queue. When a page needs to be replaced page in the front of the queue is selected for removal.
Example: Page Reference string 1, 3, 0, 3, 5, 6, 3 with 3 page frames. Find number of page faults

1	3	0	3	5	6	3
		0	0	0	0	3
	3	3	3	3	6	6
1	1	1	1	5	5	5
Miss	Miss	Miss	Hit	Miss	Miss	Miss

Answer 6

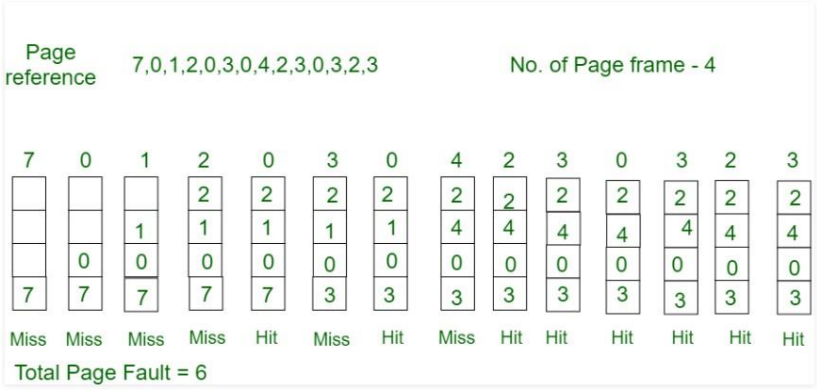
Bélády's anomaly is the name given to the phenomenon where increasing the number of page frames results in an increase in the number of page faults for a given memory access pattern.

Try it out:

Try to find the number of page faults when:
Reference string 3, 2, 1, 0, 3, 2, 4, 3, 2, 1, 0, 4
has 3 slots
Same reference string with 4 slots

Optimal Page replacement

Here pages are replaced which would not be used for the longest duration of time in the future. Try page references 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 3 with 4 page frame

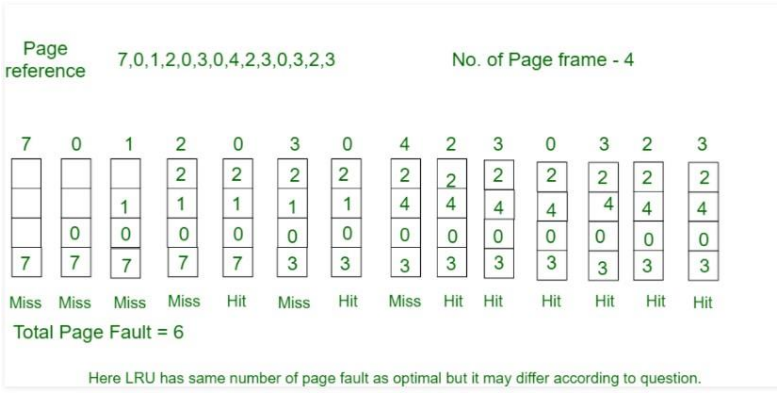


7	0	1	2	0	3	0	4	2	3	0	3	2	3
			2	2	2	2	2	2	2	2	2	2	2
		1	1	1	1	1	4	4	4	4	4	4	4
	0	0	0	0	0	0	0	0	0	0	0	0	0
7	7	7	7	7	3	3	3	3	3	3	3	3	3
M	M	M	M	H	M	H	M	H	H	H	H	H	H

Least Recently Used

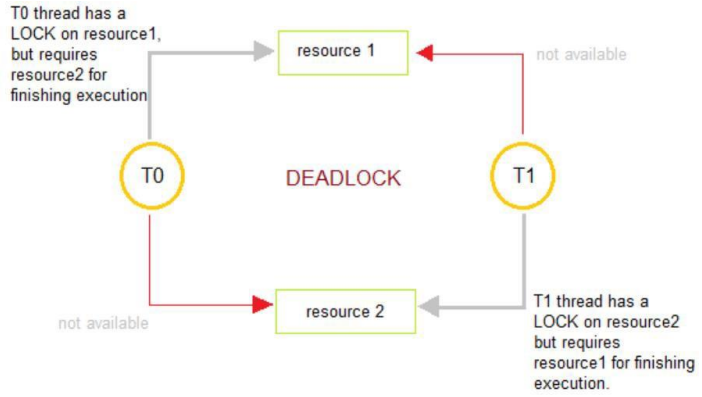
In this algorithm page will be replaced which is least recently used.

Consider the page reference string 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 3 with 4 page frames. Find number of page faults.



Deadlocks

Deadlocks are a set of blocked processes each holding a resource and waiting to acquire a resource held by another process



T0 and T1 are in a deadlock because each of them needs the resource of others to complete their execution but neither of them is willing to give up their resources

Normal mode of Operation utilization of resources by a process is in the following sequence

- Request: Firstly, the process requests the resource.
- Use: The Process can operate on the resource
- Release: The Process releases the resource.

Necessary Conditions that must hold for a deadlock

Mutual Exclusion

According to this condition, at least one resource should be non-shareable (non-shareable resources are those that can be used by one process at a time.)

Hold and wait

According to this condition, a process is holding at least one resource and is waiting for additional resources.

No pre-emption

Resources cannot be taken from the process because resources can be released only voluntarily by the process holding them.

Circular wait

In this condition, the set of processes are waiting for each other in the circular form.

Handling Deadlocks

Ignore

According to this method, it is assumed that deadlock would never occur.

Avoid

This method is used by the operating system in order to check whether the system is in a safe state or in an unsafe state. This method checks every step performed by the operating system. Any process continues its execution until the system is in a safe state. Once the system enters into an unsafe state, the operating system has to take a step back. This concept is more comfortable for single user system because they use their system for simply browsing as well as other simple activities.

Prevent

The main aim of the deadlock prevention method is to violate any one condition among the four; because if any of one condition is violated then the problem of deadlock will never occur.

Starvation vs Deadlock

Starvation	Deadlock
When all the low priority processes got blocked, while the high priority processes execute then this situation is termed as Starvation.	Deadlock is a situation that occurs when one of the processes got blocked.
Starvation is a long waiting but it is not an infinite process.	Deadlock is an infinite process.
It is not necessary that every starvation is a deadlock.	There is starvation in every deadlock.
Starvation is due to uncontrolled priority and resource management.	During deadlock, preemption and circular wait does not occur simultaneously.

Deadlock Simulator Program

```
#include <iostream>

using namespace std;

//Description: This program simulates the execution of 3 processes that will forever be stuck in a deadlock. Each process is looking for a particular person, and each process also can release a person another process is looking for.

//This program is set up in such a way in which none of the processes will be able to ever find the person that another process could release.

int user1 = 0; //Represents user1


int user2 = 0; //Represents user2

int user3 = 0; //Represents user3
```

```
int found_someone = 0; //Used to determine if any of the processes have found the user or not
//Release user1, allowing a process to find him
void release_user1()
{
        user1 = 1; //Set user1 to found
}
//Look for user1
void find_user1()
{
        //If user1 is found: Print that he was caught
        //Else: Ask for user1
          if(user1)
           {
                cout<<"Caught
          }
       else
          {
                cout<<"Where is
          }

}

  //Release user2, allowing a process to find
  void release_user2()
{
          user2 = 1; //Set user2 to
}
  //Look for
  void find_user2()

{
```

```
          //If user2 is found: Print
           //Else: Ask where user2 is
          if(user2)
          {
                    cout<<"Caught
          }
        else
          {
  cout<<"Where is  user2?"<<endl;
          }

 }
 //Release user3, allowing a process to find her
void release_user3()
          user3 = 1;
{

 }
 //Look for user3


 void find_user3()

 {

          //If user3 is found: Print that she was found

          //Else: Ask where in the world she is

          if(user3)

          {
```

```
                cout<<"caught user3"<<endl;
        }
        else
        {
                cout<<"Where is user3?"<<endl;
        }
}

//Process looking for user1
void process1()
{
        find_user1();

        //If user1 is found: Release user2, allowing another process to find him
        if(user1)
        {
```

```
                release_user2();
                found_someone = 1;
        }
}
//Process looking for user2
void process2()
{
        find_user2();

        //If user2 is found: Release user3, allowing another process to find her
        if(user2)
        {
                release_user3();
                found_someone = 1;
        }
```

```
}

//Process looking for user3

void process3()

{

        find_user3();

        //If user3 is found: Release user1, allowing another process to find him

        if(user3)

        {

                release_user1();

                found_someone = 1;

        }

}

int main()

{
```

```
        while(!found_someone)
        {
                process1();
                process2();
                process3();
        }
}
```

Producer Consumer Problem

The problem describes two processes, the producer and the consumer, which share a common, fixed-size buffer used as a queue.

The producer's job is to generate data, put it into the buffer, and start again.

At the same time, the consumer is consuming the data (i.e. removing it from the buffer), one piece at a time.

Problem

To make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

Solution

The producer is to either go to sleep or discard data if the buffer is full. The next time the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again. In the same way, the consumer can go to sleep if it finds the buffer to be empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer.

An inadequate solution could result in a deadlock where both processes are waiting to be awakened.

OPERATING SYSTEM