# LECTURE NOTES

# ON

# REAL TIME OPERATING SYSTEMS (CS852PE)

*MODULE –I*

# Introduction to UNIX/LINUX

Introduction to Linux

Linux is a Unix-like computer operating system assembled under the model of free and open source software development and distribution. The defining component of Linux is the Linux kernel, an operating system kernel first released 5 October 1991 by Linus Torvalds.

Linux was originally developed as a free operating system for Intel x86-based personal computers. It has since been ported to more computer hardware platforms than any other operating system. It is a leading operating system on servers and other big iron systems such as mainframe computers and supercomputers more than 90% of today's 500 fastest supercomputers run some variant of Linux, including the 10 fastest. Linux also runs on embedded systems (devices where the operating system is typically built into the firmware and highly tailored to the system) such as mobile phones, tablet computers, network routers, televisions and video game consoles; the Android system in wide use on mobile devices is built on the Linux kernel.

A distribution oriented toward desktop use will typically include the X Window System and an accompanying desktop environment such as GNOME or KDE Plasma. Some such distributions may include a less resource intensive desktop such as LXDE or Xfce for use on older or less powerful computers. A distribution intended to run as a server may omit all graphical environments from the standard install and instead include other software such as the Apache HTTP Server and an SSH server such as OpenSSH. Because Linux is freely redistributable, anyone may create a distribution for any intended use. Applications commonly used with desktop Linux systems include the Mozilla Firefox web browser, the LibreOffice office application suite, and the GIMP image editor.Since the main supporting user space system tools and libraries originated in the GNU Project, initiated in 1983 by Richard Stallman, the Free Software Foundation prefers the name *GNU/Linux*.

History of Unix

The Unix operating system was conceived and implemented in 1969 at AT&T's Bell Laboratories in the United States by Ken Thompson, Dennis Ritchie, Douglas McIlroy, and Joe Ossanna. It was first released in 1971 and was initially entirely written in assembly language, a common practice at the time. Later, in a key pioneering approach in 1973, Unix was re-written in the programming language C by Dennis Ritchie (with exceptions to the kernel and I/O). The availability of an operating system written in a high-level language allowed easier portability to different computer platforms.

Today, Linux systems are used in every domain, from embedded systems to supercomputers, and have secured a place in server installations often using the popular LAMP application stack. Use of Linux distributions in home and enterprise desktops has been growing. They have also gained popularity with various local and national governments. The federal government of Brazil is well known for its support for Linux. News of the Russian

military creating its own Linux distribution has also surfaced, and has come to fruition as the G.H.ost Project. The Indian state of Kerala has gone to the extent of mandating that all state high schools run Linux on their computers.

Design

A Linux-based system is a modular Unix-like operating system. It derives much of its basic design from principles established in Unix during the 1970s and 1980s. Such a system uses a monolithic kernel, the Linux kernel, which handles process control, networking, and peripheral and file system access. Device drivers are either integrated directly with the kernel or added as modules loaded while the system is running.

Separate projects that interface with the kernel provide much of the system's higher- level functionality. The GNU userland is an important part of most Linux-based systems, providing the most common implementation of the C library, a popular shell, and many of the common Unix tools which carry out many basic operating system tasks. The graphical user interface (or GUI) used by most Linux systems is built on top of an implementation of the X Window System.

Programming on Linux

Most Linux distributions support dozens of programming languages. The original development tools used for building both Linux applications and operating system programs are found within the GNU toolchain, which includes the GNU Compiler Collection (GCC) and the GNU build system. Amongst others, GCC provides compilers for Ada, C, C++, Java, and Fortran. First released in 2003, the Low Level Virtual Machine project provides an alternative open-source compiler for many languages. Proprietary compilers for Linux include the Intel C++ Compiler, Sun Studio, and IBM XL C/C++ Compiler. BASIC in the form of Visual Basic is supported in such forms as Gambas, FreeBASIC, and XBasic.

Most distributions also include support for PHP, Perl, Ruby, Python and other dynamic languages. While not as common, Linux also supports C# (via Mono), Vala, and Scheme. A number of Java Virtual Machines and development kits run on Linux, including the original Sun Microsystems JVM (HotSpot), and IBM's J2SE RE, as well as many open- source projects like Kaffe and JikesRVM.

**Linux Advantages**

**Low cost:** You don‘t need to spend time and money to obtain licenses since Linux and much of its software come with the GNU General Public License. You can start to work immediately without worrying that your software may stop working anytime because the free trial version expires. Additionally, there are large repositories from which you can freely download high quality software for almost any task you can think of.

**Stability:** Linux doesn‘t need to be rebooted periodically to maintain performance levels. It doesn‘t freeze up or slow down over time due to memory leaks and such. Continuous up-times of hundreds of days (up to a year or more) are not uncommon.

**Performance:** Linux provides persistent high performance on workstations and on networks. It can handle unusually large numbers of users simultaneously, and can make old computers sufficiently responsive to be useful again.

**Network friendliness:** Linux was developed by a group of programmers over the Internet and has therefore strong support for network functionality; client and server systems can be easily set up on any computer running Linux. It can perform tasks such as network backups faster and more reliably than alternative systems.

**Flexibility:** Linux can be used for high performance server applications, desktop applications, and embedded systems. You can save disk space by only installing the components needed for a particular use. You can restrict the use of specific computers by installing for example only selected office applications instead of the whole suite.

**Compatibility:** It runs all common Unix software packages and can process all common file formats.

**Choice:** The large number of Linux distributions gives you a choice. Each distribution is developed and supported by a different organization. You can pick the one you like best; the core functionalities are the same; most software runs on most distributions.

**Fast and easy installation:** Most Linux distributions come with user-friendly installation and setup programs. Popular Linux distributions come with tools that make installation of additional software very user friendly as well.

**Full use of hard disk:** Linux continues work well even when the hard disk is almost full. 10.**Multitasking:** Linux is designed to do many things at the same time; e.g., a large printing job in the background won‘t slow down your other work.

**Security:** Linux is one of the most secure operating systems. ―Walls‖ and flexible file access permission systems prevent access by unwanted visitors or viruses. Linux users have to option to select and safely download software, free of charge, from online repositories containing thousands of high quality packages. No purchase transactions requiring credit card numbers or other sensitive personal information are necessary.

**Open Source:** If you develop software that requires knowledge or modification of the operating system code, Linux‘s source code is at your fingertips. Most Linux applications are Open Source as well.

**The difference between Linux and UNIX operating systems?**

UNIX is copyrighted name only big companies are allowed to use the UNIX copyright and name, so IBM AIX and Sun Solaris and HP-UX all are UNIX operating systems. The Open Group holds the UNIX trademark in trust for the industry, and manages the UNIX trademark licensing program.

Most UNIX systems are commercial in nature.

**Linux is a UNIX Clone**

But if you consider Portable Operating System Interface (POSIX) standards then Linux can be considered as UNIX. To quote from Official Linux kernel README file:

Linux is a Unix clone written from scratch by Linus Torvalds with assistance from a loosely- knit team of hackers across the Net. It aims towards POSIX compliance.

However, "Open Group" do not approve of the construction "Unix-like", and consider it misuse of their UNIX trademark.

**Linux Is Just a Kernel**

Linux is just a kernel. All Linux distributions includes GUI system + GNU utilities (such as cp, mv, ls,date, bash etc) + installation & management tools + GNU c/c++ Compilers + Editors (vi) + and various applications (such as OpenOffice, Firefox). However, most UNIX operating systems are considered as a complete operating system as everything come from a single source or vendor. As I said earlier Linux is just a kernel and Linux distribution makes it complete usable operating systems by adding various applications. Most UNIX operating systems comes with A-Z programs such as editor, compilers etc. For example HP-UX or Solaris comes with A-Z programs.

**License and cost**

Linux is Free (as in beer [freedom]). You can download it from the Internet or redistribute it under GNU licenses. You will see the best community support for Linux. Most UNIX like operating systems are not free

(but this is changing fast, for example OpenSolaris UNIX). However, some Linux distributions such as Redhat / Novell provides additional Linux support, consultancy, bug fixing, and training for additional fees.

**User-Friendly**

Linux is considered as most user friendly UNIX like operating systems. It makes it easy to install sound card, flash players, and other desktop goodies. However, Apple OS X is most popular UNIX operating system for desktop usage.

**Security Firewall Software**

Linux comes with open source netfilter/iptables based firewall tool to protect your server and desktop from the crackers and hackers. UNIX operating systems comes with its own firewall product (for example Solaris UNIX comes with ipfilter based firewall) or you need to purchase a 3rd party software such as Checkpoint UNIX firewall.

**Backup and Recovery Software**

UNIX and Linux comes with different set of tools for backing up data to tape and other backup media. However, both of them share some common tools such as tar, dump/restore, and cpio etc.

**File Systems**

Linux by default supports and use ext3 or ext4 file systems.
UNIX comes with various file systems such as jfs, gpfs (AIX), jfs, gpfs (HP-UX), jfs, gpfs (Solaris).

**System Administration Tools**

UNIX comes with its own tools such as SAM on HP-UX.
Suse Linux comes with Yast
Redhat Linux comes with its own gui tools called redhat-config-*.
However, editing text config file and typing commands are most popular options for sys admin work under UNIX and Linux.

**System Startup Scripts**

Almost every version of UNIX and Linux comes with system initialization script but they are located in different directories:
HP-UX - /sbin/init.d
AIX - /etc/rc.d/init.d
Linux - /etc/init.d

**End User Perspective**

The differences are not that big for the average end user. They will use the same shell (e.g. bash or ksh) and other development tools such as Perl or Eclipse development tool.

**System Administrator Perspective**

Again, the differences are not that big for the system administrator. However, you may notice various differences while performing the following operations:
Software installation procedure
Hardware device names
Various admin commands or utilities
Software RAID devices and mirroring
Logical volume management
Package management
Patch management

**UNIX Operating System Names**

A few popular names:
HP-UX
IBM AIX
Sun Solairs
Mac OS X
**IRIX**

**Linux Distribution (Operating System) Names**

A few popular names:
Redhat Enterprise Linux

Fedora Linux
Debian Linux
Suse Enterprise Linux
Ubuntu Linux

**Common Things Between Linux & UNIX**

Both share many common applications such as:
GUI, file, and windows managers (KDE, Gnome)
Shells (ksh, csh, bash)
Various office applications such as OpenOffice.org
Development tools (perl, php, python, GNU c/c++ compilers)
Posix interface

## FILE HANDLING UTILITIES:

**cat Command:**

cat linux command concatenates files and print it on the standard output.

SYNTAX:
The Syntax is
cat [OPTIONS] [FILE]...

OPTIONS:

| | |
|---|---|
| -A | Show all. |
| -b | Omits line numbers for blank space in the output. |
| -e | A $ character will be printed at the end of each line prior to a new line. |
| -E | Displays a $ (dollar sign) at the end of each line. |
| -n | Line numbers for all the output lines. |
| -s | If the output has multiple empty lines it replaces it with one empty line. |
| -T | Displays the tab characters in the output. |
| -v | Non-printing characters (with the exception of tabs, new-lines and form-feeds) are printed visibly. |

rm Command:
rm linux command is used to remove/delete the file from the directory.

SYNTAX:
The Syntax is
s rm [options..] [file | directory]

OPTIONS:

| | |
|---|---|
| -f | Remove all files in a directory without prompting the user. |
| -i | Interactive. With this option, rm prompts for confirmation before removing any files. |

| | |
|---|---|
| -r (or) -R | Recursively remove directories and subdirectories in the argument list. The directory will be emptied of files and removed. The user is normally prompted for removal of any write-protected files which the directory contains. |

cd Command:
cd command is used to change the directory.

SYNTAX:
The Syntax is

cd [directory | ~ | ./ | ../ | - ]

OPTIONS:

| | |
|---|---|
| -L | Use the physical directory structure. |
| -P | Forces symbolic links. |

**EXAMPLE:**

cd linux-command
This command will take you to the sub-directory(linux-command) from its parent directory.
cd **..**
This will change to the parent-directory from the current working directory/sub- directory.
cd ~
This command will move to the user's home directory which is "/home/username".
cp Command:
cp command copy files from one location to another. If the destination is an existing file, then the file is overwritten; if the destination is an existing directory, the file is copied into the directory (the directory is not overwritten).

SYNTAX:
The Syntax is
cp [OPTIONS]... SOURCE DEST
cp [OPTIONS]... SOURCE... DIRECTORY
cp [OPTIONS]... --target-directory=DIRECTORY SOURCE...

OPTIONS:

| | |
|---|---|
| -a | same as -dpR. |
| --backup[=CONTROL] | make a backup of each existing destination file |
| -b | like --backup but does not accept an argument. |
| -f | if an existing destination file cannot be opened, remove it and try again. |
| -p | same as --preserve=mode,ownership,timestamps. |
| --preserve[=ATTR_LIST] | preserve the specified attributes (default: mode,ownership,timestamps) and security contexts, if possible additional attributes: links, all. |

| | |
|---|---|
| --no-preserve=ATTR_LIST | don't preserve the specified attribute. |
| --parents | append source path to DIRECTORY. |

Copy two files: cp file1 file2
The above cp command copies the content of file1.php to file2.php.

To backup the copied file: cp -b file1.php file2.php
Backup of file1.php will be created with '~' symbol as file2.php~.
Copy folder and subfolders: cp -R scripts scripts1
The above cp command copy the folder and subfolders from scripts to scripts1.
ls Command:
ls command lists the files and directories under current working directory.

SYNTAX:
The Syntax is
ls [OPTIONS]... [FILE]

ln Command:
ln command is used to create link to a file (or) directory. It helps to provide soft link for desired files. Inode will be different for source and destination.

SYNTAX:
The Syntax is
ln [options] existingfile(or directory)name newfile(or directory)name

OPTIONS:

| | |
|---|---|
| -f | Link files without questioning the user, even if the mode of target forbids writing. This is the default if the standard input is not a terminal. |
| -n | Does not overwrite existing files. |
| -s | Used to create soft links. |

**EXAMPLE:**

ln -s file1.txt file2.txt
Creates a symbolic link to 'file1.txt' with the name of 'file2.txt'. Here inode for 'file1.txt' and 'file2.txt' will be different.
ln -s nimi nimi1
Creates a symbolic link to 'nimi' with the name of 'nimi1'.
mkdir COMMAND:
mkdir command is used to create one or more directories.

SYNTAX:
The Syntax is
mkdir [options] directories

OPTIONS:

| | |
|---|---|
| -m | Set the access mode for the new directories. |
| -p | Create intervening parent directories if they don't exist. |
| -v | Print help message for each directory created. |

rmdir Command:
rmdir command is used to delete/remove a directory and its subdirectories.

SYNTAX:
The Syntax is
rmdir [options..] Directory

OPTIONS:

| | |
|---|---|
| -p | Allow users to remove the directory dirname and its parent directories which become empty. |

**EXAMPLE:**

To delete/remove a directory rmdir tmp
rmdir command will remove/delete the directory tmp if the directory is empty.
To delete a directory tree: rm -ir tmp
This command recursively removes the contents of all subdirectories of the tmp directory, prompting you regarding the removal of each file, and then removes the tmp directory itself.
mv Command:
mv command which is short for move. It is used to move/rename file from one directory to another. mv command is different from cp command as it completely removes the file from the source and moves to the directory specified, where cp command just copies the content from one file to another.

SYNTAX:
The Syntax is
mv [-f] [-i] oldname newname

OPTIONS:

| | |
|---|---|
| -f | This will not prompt before overwriting (equivalent to --reply=yes). mv -f will move the file(s) without prompting even if it is writing over an existing target. |
| -i | Prompts before overwriting another file. |

mv file1.txt file2.txt
This command renames file1.txt as file2.txt
To move a directory mv hscripts  tmp
In the above line mv command moves all the files, directories and sub-directories from hscripts folder/directory to tmp directory if the tmp directory already exists. If there is no tmp directory it rename's the hscripts directory as tmp directory.
To Move multiple files/More files into another directory mv file1.txt tmp/file2.txt newdir
This command moves the files file1.txt from the current directory and file2.txt from the tmp folder/directory to newdir.
diff Command:
diff command is used to find differences between two files.

SYNTAX:
The Syntax is
diff [options..] from-file to-file

OPTIONS:

| | |
|---|---|
| -a | Treat all files as text and compare them line-by-line. |
| -b | Ignore changes in amount of white space. |
| -c | Use the context output format. |
| -e | Make output that is a valid ed script. |
| -H | Use heuristics to speed handling of large files that have numerous scattered small changes. |
| -i | Ignore changes in case; consider upper- and lower-case letters equivalent. |
| -n | Prints in RCS-format, like -f except that each command specifies the number of lines affected. |
| -q | Output RCS-format diffs; like -f except that each command specifies the number of lines affected. |
| -r | When comparing directories, recursively compare any subdirectories found. |
| -s | Report when two files are the same. |
| -w | Ignore white space when comparing lines. |
| -y | Use the side by side output format. |

chown Command:
chown command is used to change the owner / user of the file or directory. This is an admin command, root user only can change the owner of a file or directory.

SYNTAX:
The Syntax is
chown [options] newowner filename/directoryname

OPTIONS:

| | |
|---|---|
| -R | Change the permission on files that are in the subdirectories of the directory that you are currently in. |
| -c | Change the permission for each file. |
| -f | Prevents chown from displaying error messages when it is unable to change the ownership of a file. |

**EXAMPLE:**

chown hiox test.txt
The owner of the 'test.txt' file is root, Change to new user hiox.
chown -R hiox test
The owner of the 'test' directory is root, With -R option the files and subdirectories user also gets changed.
chown -c hiox calc.txt
Here Change The Owner For The Specific 'Calc.Txt' File Only.

chmod Command:
chmod command allows you to alter / Change access rights to files and directories.

File Permission is given for users,group and others as,

| | Read | Write | Execute |
|---|---|---|---|
| **User** | | | |
| **Group** | | | |
| **Others** | | | |

**Permission** 000
**Symbolic Mode** ---

**SYNTAX:**
The Syntax is
chmod [options] [MODE] FileName

**File Permission**

| # | File Permission |
|---|---|
| 0 | none |
| 1 | execute only |
| 2 | write only |
| 3 | write and execute |
| 4 | read only |
| 5 | read and execute |
| 6 | read and write |
| 7 | set all permissions |

**OPTIONS:**

| | |
|---|---|
| -c | Displays names of only those files whose permissions are being changed |
| -f | Suppress most error messages |
| -R | Change files and directories recursively |
| -v | Output version information and exit. |

**EXAMPLE:**

To view your files with what permission they are: ls -alt
This command is used to view your files with what permission they are.
To make a file readable and writable by the group and others. chmod 066 file1.txt

To allow everyone to read, write, and execute the file chmod 777 file1.txt
with friend ship | with friend ship

chgrp Command:
chgrp command is used to change the group of the file or directory. This is an admin command. Root user only can change the group of the file or directory.

SYNTAX:
The Syntax is
chgrp [options] newgroup filename/directoryname

OPTIONS:

| | |
|---|---|
| -R | Change the permission on files that are in the subdirectories of the directory that you are currently in. |
| -c | Change the permission for each file. |
| -f | Force. Do not report errors. |

Hioxindia.com <
EXAMPLE:

chgrp hiox test.txt
The group of 'test.txt' file is root, Change to newgroup hiox.
chgrp -R hiox test
The group of 'test' directory is root. With -R, the files and its subdirectories also changes to newgroup hiox.
chgrp -c hiox calc.txt
They above command is used to change the group for the specific file('calc.txt') only.

PROCESS UTILITIES:
**ps COMMAND:**
ps command is used to report the process status. ps is the short name for Process Status.

SYNTAX:
The Syntax is ps [options]

OPTIONS:

| | |
|---|---|
| -a | List information about all processes most frequently requested: all those except process group leaders and processes not associated with a terminal.. |
| -A or e | List information for all processes. |
| -d | List information about all processes except session leaders. |
| -e | List information about every process now running. |
| -f | Generates a full listing. |
| -j | Print session ID and process group ID. |
| -l | Generate a long listing. |

kill COMMAND:

kill command is used to kill the background process.

SYNTAX:
The Syntax is
kill [-s] [-l] %pid

OPTIONS:

| | |
|---|---|
| -s | Specify the signal to send. The signal may be given as a signal name or number. |
| -l | Write all values of signal supported by the implementation, if no operand is given. |
| -pid | Process id or job id. |
| -9 | Force to kill a process. |

| | |
|---|---|
| | special built-in utilities, the results are undefined. |
| argument | Any string to be supplied as an argument when invoking command. |

Examples
**nice +13 pico myfile.txt** - runs the pico command on myfile.txt with an increment of +13. About at
Schedules a command to be ran at a particular time, such as a print job late at night.
Syntax

| | |
|---|---|
| at | executes commands at a specified time. |
| atq | lists the user's pending jobs, unless the user is the superuser; in that case, everybody's jobs are listed. The format of the output lines (one for each job) is: Job number, date, hour, job class. |
| atrm | deletes jobs, identified by their job number. |
| batch | executes commands when system load levels permit; in other words, when the load average drops below 1.5, or the value specified in the invocation of atrun. |

*at [-c | -k | -s] [-f filename] [-q queuename] [-m] -t time [date] [-l] [-r]*

| | |
|---|---|
| -c | C shell. csh(1) is used to execute the at-job. |
| -k | Korn shell. ksh(1) is used to execute the at-job. |
| -s | Bourne shell. sh(1) is used to execute the at-job. |
| -f filename | Specifies the file that contains the command to run. |
| -m | Sends mail once the command has been run. |

-t time Specifies at what time you want the command to be ran. Format hh:mm. am / pm indication can also follow the time otherwise a 24-hour clock is used. A timezone name of GMT, UCT or ZULU (case insensitive) can follow to specify that the time is in Coordinated Universal Time. Other timezones can be specified using the TZ environment variable. The below quick times can also be entered:

midnight - Indicates the time 12:00 am (00:00). noon - Indicates the time 12:00 pm.
now - Indicates the current day and time. Invoking at - now will submit submit an at-job for potentially immediate execution.

date Specifies the date you wish it to be ran on. Format month, date, year. The following quick days can also be entered:

today - Indicates the current day.
tomorrow - Indicates the day following the current day.

-l Lists the commands that have been set to run.

-r Cancels the command that you have set in the past.

Examples
**at -m 01:35 < atjob** = Run the commands listed in the 'atjob' file at 1:35AM, in addition all output that is generated from job mail to the user running the task. When this command has been successfully enter you should receive a prompt similar to the below example.
commands will be executed using /bin/csh job 1072250520.a at Wed Dec 24 00:22:00 2003
**at -l** = This command will list each of the scheduled jobs as seen below. 1072250520.a Wed Dec 24 00:22:00 2003
**at -r 1072250520.a** = Deletes the job just created. or
**atrm 23** = Deletes job 23.
If you wish to create a job that is repeated you could modify the file that executes the commands with another command that recreates the job or better yet use the crontab command.

FILTERS:
**more COMMAND:**
more command is used to display text in the terminal screen. It allows only backward movement.

SYNTAX:
The Syntax is
more [options] filename

OPTIONS:

-c Clear screen before displaying.
-e Exit immediately after writing the last line of the last file in the argument list.

| | |
|---|---|
| -n | Specify how many lines are printed in the screen for a given file. |
| +n | Starts up the file from the given number. |

**EXAMPLE:**

more -c index.php
Clears the screen before printing the file .
more -3 index.php
Prints first three lines of the given file. Press **Enter** to display the file line by line.

head COMMAND:
head command is used to display the first ten lines of a file, and also specifies how many lines to display.

SYNTAX:
The Syntax is
head [options] filename

OPTIONS:

| | |
|---|---|
| -n | To specify how many lines you want to display. |
| -n number | The number option-argument must be a decimal integer whose sign affects the location in the file, measured in lines. |
| -c number | The number option-argument must be a decimal integer whose sign affects the location in the file, measured in bytes. |

**EXAMPLE:**

head index.php
This command prints the first 10 lines of 'index.php'.
head -5 index.php
The head command displays the first 5 lines of 'index.php'.
head -c 5 index.php
The above command displays the first 5 characters of 'index.php'.
tail COMMAND:
tail command is used to display the last or bottom part of the file. By default it displays last 10 lines of a file.

SYNTAX:
The Syntax is
tail [options] filename

OPTIONS:

| | |
|---|---|
| -l | To specify the units of lines. |
| -b | To specify the units of blocks. |
| -n | To specify how many lines you want to display. |
| -c number | The number option-argument must be a decimal integer whose sign affects the location in the file, measured in bytes. |

| | |
|---|---|
| -n number | The number option-argument must be a decimal integer whose sign affects the location in the file, measured in lines. |

**EXAMPLE:**

tail index.php
It displays the last 10 lines of 'index.php'.
tail -2 index.php
It displays the last 2 lines of 'index.php'.
tail -n 5 index.php
It displays the last 5 lines of 'index.php'.
tail -c 5 index.php
It displays the last 5 characters of 'index.php'.
cut COMMAND:
cut command is used to cut out selected fields of each line of a file. The cut command uses delimiters to determine where to split fields.

SYNTAX:
The Syntax is cut [options]

OPTIONS:

| | |
|---|---|
| -c | Specifies character positions. |
| -b | Specifies byte positions. |
| -d flags | Specifies the delimiters and fields. |

**EXAMPLE:**
cut -c1-3 text.txt
Output:
Thi
Cut the first three letters from the above line.
cut -d, -f1,2 text.txt
Output:
This is, an example program
The above command is used to split the fields using delimiter and cut the first two fields.
paste COMMAND:
paste command is used to paste the content from one file to another file. It is also used to set column format for each line.

SYNTAX:
The Syntax is paste [options]

OPTIONS:

| | |
|---|---|
| -s | Paste one file at a time instead of in parallel. |
| -d | Reuse characters from LIST instead of TABs . |

EXAMPLE:

paste test.txt>test1.txt
Paste the content from 'test.txt' file to 'test1.txt' file.
ls | paste - - - -

List all files and directories in four columns for each line.
sort COMMAND:
sort command is used to sort the lines in a text file.

SYNTAX:
The Syntax is
sort [options] filename

OPTIONS:

| | |
|---|---|
| -r | Sorts in reverse order. |
| -u | If line is duplicated display only once. |
| -o filename | Sends sorted output to a file. |

**EXAMPLE:**

sort test.txt
Sorts the 'test.txt'file and prints result in the screen.
sort -r test.txt
Sorts the 'test.txt' file in reverse order and prints result in the screen.
About uniq
Report or filter out repeated lines in a file.
Syntax
*uniq [-c | -d | -u ] [ -f fields ] [ -s char ] [-n] [+m] [input_file [ output_file ] ]*

| | |
|---|---|
| -c | Precede each output line with a count of the number of times the line occurred in the input. |
| -d | Suppress the writing of lines that are not repeated in the input. |
| -u | Suppress the writing of lines that are repeated in the input. |
| -f fields | Ignore the first fields fields on each input line when doing comparisons, where fields is a positive decimal integer. A field is the maximal string matched by the basic regular expression:<br>[[:blank:]]*[^[:blank:]]*<br>If fields specifies more fields than appear on an input line, a null string will be used for comparison. |
| -s char | Ignore the first chars characters when doing comparisons, where chars is a positive decimal integer. If specified in conjunction with the -f option, the first chars characters after the first fields fields will be ignored. If chars specifies more characters than remain on an input line, a null string will be used for comparison. |
| -n | Equivalent to -f fields with fields set to n. |
| +m | Equivalent to -s chars with chars set to m. |

| | |
|---|---|
| input_file | A path name of the input file. If input_file is not specified, or if the input_file is -, the standard input will be used. |
| output_file | A path name of the output file. If output_file is not specified, the standard output will be used. The results are unspecified if the file named by output_file is the file named by input_file. |

Examples

**uniq myfile1.txt > myfile2.txt** - Removes duplicate lines in the first file1.txt and outputs the results to the second file.

About tr

Translate characters.

Syntax

*tr [-c] [-d] [-s] [string1] [string2]*

| | |
|---|---|
| -c | Complement the set of characters specified by string1. |
| -d | Delete all occurrences of input characters that are specified by string1. |
| -s | Replace instances of repeated characters with a single character. |
| string1 | First string or character to be changed. |
| string2 | Second string or character to change the string1. |

Examples

**echo "12345678 9247" | tr 123456789 computerh** - this example takes an echo response of '12345678 9247' and pipes it through the tr replacing the appropriate numbers with the letters. In this example it would return *computer hope*.

**tr -cd '\11\12\40-\176' < myfile1 > myfile2** - this example would take the file myfile1 and strip all non printable characters and take that results to myfile2.

**General Commands:**

**date COMMAND:**

date command prints the date and time.

SYNTAX:

The Syntax is

date [options] [+format] [date]

OPTIONS:

| | |
|---|---|
| -a | Slowly adjust the time by sss.fff seconds (fff represents fractions of a second). This adjustment can be positive or negative.Only system admin/ super user can adjust the time. |
| -s date-string | Sets the time and date to the value specfied in the datestring. The datestr may contain the month names, timezones, 'am', 'pm', etc. |
| -u | Display (or set) the date in Greenwich Mean Time (GMT-universal time). |

**Format:**

| | |
|---|---|
| %a | Abbreviated weekday(Tue). |
| %A | Full weekday(Tuesday). |
| %b | Abbreviated month name(Jan). |
| %B | Full month name(January). |
| %c | Country-specific date and time format.. |
| %D | Date in the format %m/%d/%y. |
| %j | Julian day of year (001-366). |
| %n | Insert a new line. |
| %p | String to indicate a.m. or p.m. |
| %T | Time in the format %H:%M:%S. |
| %t | Tab space. |
| %V | Week number in year (01-52); start week on Monday. |

**EXAMPLE:**

date command date
The above command will print Wed Jul 23 10:52:34 IST 2008
To use tab space:
date +"Date is %D %t Time is %T"
The above command will removespace and print as Date is 07/23/08 Time is 10:52:34
To know the week number of the year, date -V
The above command will print 30
To set the date,
date -s "10/08/2008 11:37:23"

The above command will print Wed Oct 08 11:37:23 IST 2008
who COMMAND:
who command can list the names of users currently logged in, their terminal, the time they have been logged in, and the name of the host from which they have logged in.

SYNTAX:
The Syntax is
who [options] [file]

OPTIONS:

| | |
|---|---|
| am i | Print the username of the invoking user, The 'am' and 'i' must be space separated. |
| -b | Prints time of last system boot. |
| -d | print dead processes. |
| -H | Print column headings above the output. |

| | |
|---|---|
| -i | Include idle time as HOURS:MINUTES. An idle time of . indicates activity within the last minute. |
| -m | Same as who am i. |
| -q | Prints only the usernames and the user count/total no of users logged in. |
| -T,-w | Include user's message status in the output. |

**EXAMPLE:**

who -uH
Output:
NAME     LINE TIME IDLE PID COMMENT
hiox     ttyp3   Jul 10 11:08 . 4578
This sample output was produced at 11 a.m. The "." indiacates activity within the last minute.
who am i
who am i command prints the user name.
echo COMMAND:
echo command prints the given input string to standard output.

SYNTAX:
The Syntax is
echo [options..] [string]

OPTIONS:

| | |
|---|---|
| -n | do not output the trailing newline |
| -e | enable interpretation of the backslash-escaped characters listed below |
| -E | disable interpretation of those sequences in STRINGs |

Without -E, the following sequences are recognized and interpolated:

| | |
|---|---|
| \NNN | the character whose ASCII code is NNN (octal) |
| \a | alert (BEL) |
| \\ | backslash |
| \b | backspace |
| \c | suppress trailing newline |
| \f | form feed |
| \n | new line |
| \r | carriage return |
| \t | horizontal tab |
| \v | vertical tab |

EXAMPLE:

echo command
echo "hscripts Hiox India"
The above command will print as hscripts Hiox India

To use backspace:
echo -e "hscripts \bHiox \bIndia"
The above command will remove space and print as hscriptsHioxIndia
To use tab space in echo command echo -e "hscripts\tHiox\tIndia"
The above command will print as hscripts Hiox India
passwd COMMAND:
passwd command is used to change your password.

SYNTAX:
The Syntax is passwd [options]

OPTIONS:

| | |
|---|---|
| -a | Show password attributes for all entries. |
| -l | Locks password entry for name. |
| -d | Deletes password for name. The login name will not be prompted for password. |
| -f | Force the user to change password at the next login by expiring the password for name. |

EXAMPLE:

1. passwd
Entering just passwd would allow you to change the password. After entering passwd you will receive the following three prompts:
Current Password:
New Password:
Confirm New Password:
Each of these prompts must be entered correctly for the password to be successfully changed.
pwd COMMAND:
pwd - Print Working Directory. pwd command prints the full filename of the current working directory.

SYNTAX:
The Syntax is pwd [options]

OPTIONS:

| | |
|---|---|
| -P | The pathname printed will not contain symbolic links. |
| -L | The pathname printed may contain symbolic links. |

**EXAMPLE:**

1. Displays the current working directory. pwd
If you are working in home directory then, pwd command displays the current working directory as /home.
cal COMMAND:
cal command is used to display the calendar.

SYNTAX:
The Syntax is
cal [options] [month] [year]

OPTIONS:

| | |
|---|---|
| -1 | Displays single month as output. |
| -3 | Displays prev/current/next month output. |
| -s | Displays sunday as the first day of the week. |
| -m | Displays Monday as the first day of the week. |
| -j | Displays Julian dates (days one-based, numbered from January 1). |
| -y | Displays a calendar for the current year. |

EXAMPLE:

cal
Output:
September 2008
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
cal command displays the current month calendar.
2. cal -3 5 2008
Output:
April 2008 May 2008 June 2008
Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa 1 2 3 4 5 1 2 3 1 2 3 4 5 6 7
6 7 8 9 10 11 12 4 5 6 7 8 9 10 8 9 10 11 12 13 14
13 14 15 16 17 18 19 11 12 13 14 15 16 17 15 16 17 18 19 20 21
20 21 22 23 24 25 26 18 19 20 21 22 23 24 22 23 24 25 26 27 28
27 28 29 30 25 26 27 28 29 30 31 29 30
Here the cal command displays the calendar of April, May and June month of year 2008.
login Command
Signs into a new system.
Syntax
*login [ -p ] [ -d device ] [-h hostname | terminal | -r hostname ] [ name [ environ ] ]*

| | |
|---|---|
| -p | Used to pass environment variables to the login shell. |
| -d device | login accepts a device option, device. device is taken to be the path name of the TTY port login is to operate on. The use of the device option can be expected to improve login performance, since login will not need to call ttyname. The -d option is available only to users whose UID and effective UID are root. Any other attempt to use -d will cause login to quietly exit. |
| -h hostname \| terminal | Used by in.telnetd to pass information about the remote host and terminal type. |
| -r hostname | Used by in.rlogind to pass information about the remote host. |

Examples
**login computerhope.com** - Would attempt to login to the computerhope domain.
uname command

Print name of current system.

Syntax
uname [-a] [-i] [-m] [-n] [-p] [-r] [-s] [-v] [-X] [-S systemname]

| | |
|---|---|
| -a | Print basic information currently available from the system. |
| -i | Print the name of the hardware implementation (platform). |
| -m | Print the machine hardware name (class). Use of this option is discouraged; use uname -p instead. |
| -n | Print the nodename (the nodename is the name by which the system is known to a communications network). |
| -p | Print the current host's ISA or processor type. |
| -r | Print the operating system release level. |
| -s | Print the name of the operating system. This is the default. |
| -v | Print the operating system version. |
| -X | Print expanded system information, one information element per line, as expected by SCO Unix. The displayed information includes:<br>system name, node, release, version, machine, and number of CPUs.<br>BusType, Serial, and Users (set to "unknown" in Solaris)<br>OEM# and Origin# (set to 0 and 1, respectively) |
| -S systemname | The nodename may be changed by specifying a system name argument. The system name argument is restricted to SYS_NMLN characters. SYS_NMLN is an implementation specific value defined in <sys/utsname.h>. Only the super-user is allowed this capability. |

Examples
uname -arv
List the basic system information, OS release, and OS version as shown below. SunOS hope 5.7 Generic_106541-08 sun4m sparc SUNW,SPARCstation-10 **uname -p**
Display the Linux platform.

**<u>Disk utilities</u>**

**df - summarize disk block and file usage**

***df*** is used to report the number of disk blocks and inodes used and free for each file system. The output format and valid options are very specific to the OS and program version in use.
Syntax
***df*** [options] [resource]

Common Options

**-l** local file systems only (SVR4) **-k** report in kilobytes (SVR4) **df**

Filesystem kbytes used avail capacity Mounted on

/dev/sd0a 20895 19224 0 102% /

/dev/sd0h 319055 131293 155857 46% /usr

/dev/sd1g 637726 348809 225145 61% /usr/local

du - report disk space in use

***du*** reports the amount of disk space in use for the files or directories you specify.

Syntax
***du*** [options] [directory or file]

## NETWORKING COMMANDS

| Command/Syntax | What it will do |
|---|---|
| *finger* [options] *user[@hostname]* | report information about users on local and remote machines |
| *ftp* [options] *host* | transfer file(s) using file transfer protocol |
| *rcp* [options] *hostname* | remotely copy files from this machine to another machine |
| *rlogin* [options] *hostname* | login remotely to another machine |
| *rsh* [options] *hostname* | remote shell to run on another machine |
| *telnet* [host [port]] | communicate with another host using telnet protocol |

---

**TELNET** and **FTP** are Application Level Internet protocols. The TELNET and FTP protocol specifications have been implemented by many different sources, including The National Center for Supercomputer Applications (NCSA), and many other public domain and shareware sources ***rlogin*** is a remote login service that was at one time exclusive to Berkeley

BSD UNIX.

Essentially, it offers the same functionality as ***telnet***, except that it passes to the remote computer information about the user's login environment. Machines can be configured to allow connections from trusted hosts without prompting for the users' passwordsA more secure version of this protocol is the Secure SHell, **SSH**, software written by Tatu Ylonen and available via ftp://ftp.net.ohio-state.edu/pub/security/ssh.

their commands—rsh (remotshell), rcp (remote copy), and rlogin (remote login)—were prevalent in the past, but because they

offer little security, they're generally discouraged in today's environments. rsh and rlogin are similar in functionality to telnet, and rcp is similar to ftp.

***telnet*** [options] [ remote_host [ port_number ]
] ***tn3270*** [options] [ remote_host [port_number ] ] ***ftp*** [options] [ remote_host ]

Common Options ftp telnet Action

**-d** set debugging mode on

**-d** same as above (SVR4only) **-i** turn off interactive prompting

**-n** don‘t attempt auto-login on connection **-v** verbose mode on

**-l** user connect with username, **user**, on the remote host (SVR4 only) **-8** 8-bit data path (SVR4 only)

telnet solaris or

telnet 192.168.1

Few of the useful commands are listed below −

| Command | Description |
|---|---|
| put filename | Upload filename from local machine to remote machine. |
| get filename | Download filename from remote machine to local machine. |
| mput file list | Upload more than one files from local machine to remote machine. |
| mget file list | Download more than one files from remote machine to local machine. |
| prompt off | Turns prompt off, by default you would be prompted to upload or download movies using mput or mget commands. |
| prompt on | Turns prompt on. |
| Dir | List all the files available in the current directory of remote machine. |
| cd dirname | Change directory to dirname on remote machine. |
| lcd dirname | Change directory to dirname on local machine. |
| Quit | Logout from the current login. |

# TEXT PROCESSING COMMANDS

sort

File sort utility, often used as a filter in a pipe. This command sorts a *text stream* or file forwards or backwards, or according to various keys or character positions. Using the -m option, it merges presorted input files. The *info page* lists its many capabilities and options.

tsort

*Topological sort*, reading in pairs of whitespace-separated strings and sorting according to input patterns. The original purpose of **tsort** was to sort a list of dependencies for an obsolete version of the *ld* linker in an «ancient» version of UNIX.

The results of a *tsort* will usually differ markedly from those of the standard **sort**
command, above.

uniq

This filter removes duplicate lines from a sorted file. It is often seen in a pipe coupled with sort.

cat list-1 list-2 list-3 | sort | uniq > final.list # Concatenates the list files,
# sorts them,
# removes duplicate lines,
# and finally writes the result to an output file.
The useful -c option prefixes each line of the input file with its number of occurrences. bash$ **cat testfile**
This line occurs only once. This line occurs twice.
This line occurs twice.
This line occurs three times. This line occurs three times. This line occurs three times.
bash$ **uniq -c testfile**
1 This line occurs only once. 2 This line occurs twice.
3 This line occurs three times.

bash$ sort testfile | uniq -c | sort -nr
3 This line occurs three times. 2 This line occurs twice.
1 This line occurs only once.

The **sort INPUTFILE | uniq -c | sort -nr** command string produces a *frequency of occurrence* listing on the INPUTFILE file (the -nr options to **sort** cause a reverse numerical sort). This template finds use in analysis of log files and dictionary lists, and wherever the lexical structure of a document needs to be examined.

15.12. Word Frequency Analysis

&wf;
bash$ **cat testfile**
This line occurs only once. This line occurs twice.
This line occurs twice.

This line occurs three times. This line occurs three times. This line occurs three times.

bash$ **./wf.sh testfile**
6 this
6 occurs
6 line
3 times
3 three
2 twice
1 only
1 once

expand, unexpand

The **expand** filter converts tabs to spaces. It is often used in a pipe.

The **unexpand** filter converts spaces to tabs. This reverses the effect of **expand**.
cut

A tool for extracting fields from files. It is similar to the **print $N** command set in awk, but more limited. It may be simpler to use *cut* in a script than *awk*. Particularly important are the -d (delimiter) and -f (field specifier) options.

Using **cut** to obtain a listing of the mounted filesystems:

cut -d ' ' -f1,2 /etc/mtab

Using **cut** to list the OS and kernel version:

uname -a | cut -d" " -f1,3,11,12

Using **cut** to extract message headers from an e-mail folder:

bash$ grep '^Subject:' read-messages | cut -c10-80
Re: Linux suitable for mission-critical apps? MAKE MILLIONS WORKING AT HOME!!!
Spam complaint
Re: Spam complaint Using **cut** to parse a file:
# List all the users in /etc/passwd.

FILENAME=/etc/passwd

for user in $(cut -d: -f1 $FILENAME) do
echo $user done

# Thanks, Oleg Philon for suggesting this.

cut -d ' ' -f2,3 filename is equivalent to awk -F'[ ]' '{ print $2, $3 }' filename Замечание

It is even possible to specify a linefeed as a delimiter. The trick is to actually embed a linefeed (**RETURN**) in the command sequence.

bash$ **cut -d'**
' -f3,7,19 testfile
This is line 3 of testfile. This is line 7 of testfile. This is line 19 of testfile.

paste

Tool for merging together different files into a single, multi-column file. In combination with cut, useful for creating system log files.

join

Consider this a special-purpose cousin of **paste**. This powerful utility allows merging two files in a meaningful fashion, which essentially creates a simple version of a relational database.

The **join** command operates on exactly two files, but pastes together only those lines with a common tagged field (usually a numerical label), and writes the result to stdout. The files to be joined should be sorted according to the tagged field for the matchups to work properly.

File: 1.data

100 Shoes
200 Laces
300 Socks File: 2.data

100 $40.00
200 $1.00
300 $2.00
bash$ **join 1.data 2.data**
File: 1.data 2.data

100 Shoes $40.00
200 Laces $1.00
300 Socks $2.00

The tagged field appears only once in the output.
head

lists the beginning of a file to stdout. The default is 10 lines, but a different number can be specified. The command has a number of interesting options.

**FILE API FUNCTIONS**

under this concept we discuss the functions or methods of regular file. On each function we discuss the usage, syntax, arguments, argument values and return types
**Open()**:used to open a regular file.

Syntax: int open(const char *pathname, int oflag,/* mode_t mode*/); #include <sys/types.h>
#include <sys/stat.h> #include <fcntl.h>
int open(const char *pathname, int oflag,/* mode_t mode*/);

**open -oflag**

•O_RDONLY open for reading only

•O_WRONLY open for writing only

•O_RDWR open for reading and writing

•O_APPEND append on each write –not atomic when using NFS

•O_CREAT create file if it does not exist

•O_TRUNC truncate size to 0

•O_EXCL error if create and file exists

•O_SYNC Any writes on the resulting file descriptor will block the calling process until the data has been physically written to the underlying hardware .
**open -mode**

•Specifies the permissions to use if a new file is created.

•This mode only applies to future accesses of the newly created file.

**User:** S_IRWXU, S_IRUSR, S_IWUSR, S_IXUSR **Group:** S_IRWXG,S_IRGRP,S_IWGRP,S_IXGRP
**Other:** S_IRWXO, S_IROTH,S_IWOTH,S_IXOTH
•mode must be specified when O_CREAT is in the flags.

**Identifying errors**

•How can we tell if the call failed?

–the system call returns a negative number

•How can we tell what was the error?

–Using errno – a global variable set by the system call if an error has occurred.

–Defined in errno.h

–Use str error to get a string describing the problem

–Use p error to print a string describing the problem #include <errno.h>
int fd;

fd = open( FILE_NAME, O_RDONLY, 0644 );

if( fd < 0 ) {

printf( "Error opening file: %s\n", strerror( errno ) ); return -1;
}

open –possible errno values

•EEXIST–O_CREAT and O_EXCL were specified and the file exists.

•ENAMETOOLONG -A component of a pathname exceeded {NAME_MAX} characters, or an entire path name exceeded {PATH_MAX} characters.
•ENOENT -O_CREAT is not set and the named file does not exist.

•ENOTDIR -A component of the path prefix is not a directory.

•EROFS -The named file resides on a read-only file system, and write access was requested.

•ENOSPC -O_CREAT is specified, the file does not exist, and there is no space left on the file system containing the directory.
•EMFILE -The process has already reached its limit for open file descriptors.

**create()**:used to create a regular file.

Syntax: int creat(const char *pathname, mode_t mode) #include <sys/types.h>
#include <sys/stat.h> #include <fcntl.h>
Int creat(const char *pathname, mode_tmode)

Equivalent to: open(pathname, O_WRONLY|O_CREAT|O_TRUNC, mode)

**lseek:**used to position the cursor at specified location.

Syntax: off_t lseek(int fd, off_t offset, int whence); #include <sys/types.h>
#include <unistd.h>

off_t lseek(int fd, off_t offset, int whence);

fd

–The file descriptor.

–It must be an open file descriptor.

offset

–Repositions the offset of the file descriptor fd to the argument offset according to the directive whence.
•Return value

–The offset in the file after the seek

–If negative, errno is set.

**lseek–whence**

•SEEK_SET -The offset is set to offset bytes from the beginning of the file.

•SEEK_CUR -The offset is set to its current location plus offset bytes.

–currpos= lseek(fd, 0, SEEK_CUR)

•SEEK_END -The offset is set to the size of the file plus offset bytes.

–If we use SEEK_END and then write to the file, it extends the file size in kernel and the gap is filled with zeros.

**lseek: Examples**

•Move to byte #16

–newpos= lseek( fd, 16, SEEK_SET );

•Move forward 4 bytes

–newpos= lseek( fd, 4, SEEK_CUR );

•Move to 8 bytes from the end

–newpos= lseek( fd, -8, SEEK_END );

•Move backward3 bytes

–lseek(fd, -3, SEEK_CUR);

**lseek-errno**

lseek() will fail and the file pointer will remain unchanged if:

–EBADF - fd is not an open file descriptor.

–ESPIPE - fd is associated with a pipe, socket, or FIFO.

–EINVAL - Whence is not a proper value.

**Read()**:used to read a block of data from regular file. Syntax: ssize_t read(int fd,void *buff,size_t nbytes) #include <unistd.h>
ssize_tread(int fd,void *buff,size_t nbytes)

•Attempts to read nbytes of data from the object referenced by the descriptor fd into the buffer pointed to by buff.
If successful, the number of bytes actually read is returned.

If we are at end-of-file, zero is returned.

•Otherwise, -1 is returned and the global variable errno is set to indicate the error.read -errno

•EBADF -fd is not a valid file descriptor or it is not open for reading.

•EIO -An I/O error occurred while reading from the file system.

•EINTR The call was interrupted by a signal before any data was read

•EAGAIN-The file was marked for non-blocking I/O, and no data was ready to be read.

**write()**:used to write a block of data to the regular file. Syntax: ssize_t write(int fd, const void *buff, size_t nbytes) #include <unistd.h>
ssize_twrite(int fd, const void *buff, size_t nbytes)

•Attempts to write nbytes of data to the object referenced by the descriptor fd from the buffer pointed to by buff.

•Upon successful completion, the number of bytes actually written is returned.

–The number can be smaller than nbytes, even zero

•Otherwise -1 is returned and errno is set.

•―A successful return from write() does not make any guarantee that data has been committed to disk.‖ write -errno

•EBADF - fd is not a valid descriptor or it is not open for writing.

•EPIPE -An attempt is made to write to a pipe that is not open for reading by any process.

•EFBIG -An attempt was made to write a file that exceeds the maximum file size.

•EINVAL -fdis attached to an object which is unsuitable for writing (such as keyboards).

•ENOSPC -There is no free space remaining on the file system containing the file.

•EDQUOT -The user's quota of disk blocks on the file system containing the file has been exhausted.

•EIO -An I/O error occurred while writing to the file system.

•EAGAIN -The file was marked for non-blocking I/O, and no data could be written immediately.

Processes Concepts:

A process is more than just a program. Especially in a multi-user, multi-tasking operating system such as Linux there is much more to consider. Each program has a set of data that it uses to do what it needs. Often, this data is not part of the program. For example, if you are using a text editor, the file you are editing is not part of the program on disk, but is part of the process in memory. If someone else were to be using the same editor, both of you would be using the same program. However, each of you would have a different process in memory. See the figure below to see how this looks graphically.

**Kernel support for Process:**

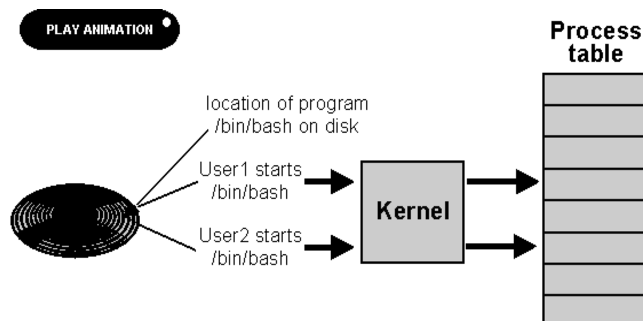
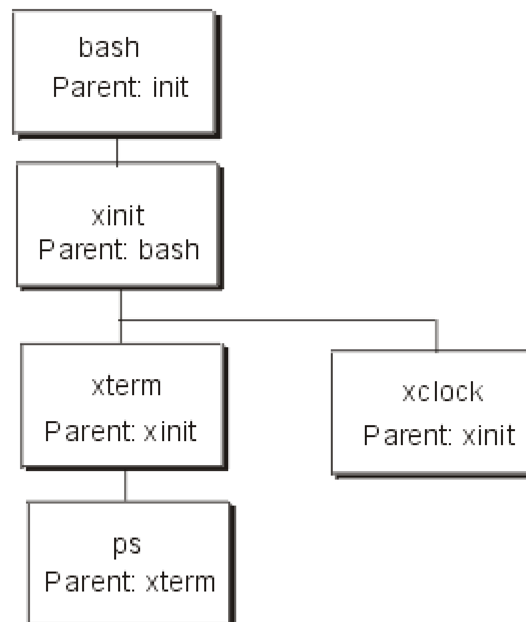


Image - Reading programs from the hard disk to create processes. (**interactive**)

Under Linux many different users can be on the system at the same time. In other words, they have processes that are in memory all at the same time. The system needs to keep track of what user is running what process, which terminal the process is running on, and what other resources the process has (such as open files). All of this is part of the process.

With the exception of the init process (PID 1) every process is the child of another process.

Another example we see in the next figure. When you login, you normally have a single process, which is your login shell(bash). If you start the X Windowing System, your shell starts another process, xinit. At this point, both your shell and xinit are running, but the shell is waiting for xinit to complete. Once X starts, you may want a terminal in which you can enter commands, so you start xterm.




## Process API

### Fork():

The fork() system call will spawn a new child process which is an identical process to the parent except that has a new system process ID. The process is copied in memory from the parent and a new process structure is assigned by the kernel. The return value of the function is which discriminates the two threads of execution. A zero is returned by the fork function in the child's process.

### exit() vs _exit():

The C library function exit() calls the kernel system call _exit() internally. The kernel system call _exit() will cause the kernel to close descriptors, free memory, and perform the kernel terminating process clean- up. The

C library function exit() call will flush I/O buffers and perform aditional clean-up before calling _exit() internally. The function exit(*status*) causes the executable to return "status" as the return code for main(). When exit(*status*) is called by a child process, it allows the parent process to examine the terminating status of the child (if it terminates first). Without this call (or a call from main() to return()) and specifying the status argument, the process will not return a value.

| #include <stdlib.h> | #include <unistd.h> |
|---|---|

| void exit(int *status*); | void _exit(int *status*); |
|---|---|

**vfork():**

The Vfork() function is the same as fork() except that it does not make a copy of the address space. The memory is shared reducing the overhead of spawning a new process with a unique copy of all the memory. This is typically used when using fork() to exec() a process and terminate. The vfork() function also executes the child process first and resumes the parent process when the child terminates.

**wait():** Blocks calling process until the child process terminates. If child process has already teminated, the wait() call returns immediately. if the calling process has multiple child processes, the function returns when one returns.

**waitpid():** Options available to block calling process for a particular child process not the first one.

**Kill():**

This is the real reason to set up a process group. One may kill all the processes in the process group without having to keep track of how many processes have been forked and all of their process id's.

**execl() and execlp():**

The function call "execl()" initiates a new program in the same environment in which it is operating. An executable (with fully qualified path. i.e. /bin/ls) and arguments are passed to the function. Note that "arg0" is the command/file name to execute.

```
int execl(const char *path, const char *arg0, const char *arg1, const char
*arg2, ... const char *argn, (char *) 0);
```

Where all function arguments are null terminated strings. The list of arguments is terminated by NULL.

The routine execlp() will perform the same purpose except that it will use environment variable PATH to determine which executable to process. Thus a fully qualified path name would not have to be used. The first argument to the function could instead be "ls". The function execlp() can also take the fully qualified name as it also resolves explicitly.

## execv() and execvp():

This is the same as execl() except that the arguments are passed as null terminated array of pointers to char. The first element "argv[0]" is the command name.

```
int execv(const char *path, char *const argv[]);
```

The routine execvp() will perform the same purpose except that it will use environment variable PATH to determine which executable to process. Thus a fully qualified path name would not have to be used. The first argument to the function could instead be "ls". The function execvp() can also take the fully qualified name as it also resolves explicitly.

## execve():

The function call "execve()" executes a process in an environment which it assigns.

Set the environment variables:

```
char *env[] = { "USER=user1", "PATH=/usr/bin:/bin:/opt/bin", (char *) 0 };
```

# UNIT-II

# INTRODUCTION TO REAL – TIME OPERATING SYSTEMS

- **Introduction**
  - A more complex software architecture is needed to handle multiple tasks, coordination, communication, and interrupt handling – an RTOS architecture
  - **Distinction:**
    - Desktop OS – OS is in control at all times and runs applications, OS runs in different address space
    - RTOS – OS and embedded software are integrated, ES starts and activates the OS – both run in the same address space (RTOS is less protected)
    - RTOS includes only service routines needed by the ES application
    - RTOS vendors: VsWorks (we got it!), VTRX, Nucleus, LynxOS, uC/OS
    - Most conform to POSIX (IEEE standard for OS interfaces)
    - Desirable RTOS properties: use less memory, application programming interface, debugging tools, support for variety of microprocessors, already-debugged network drivers

**What Is an O.S?**

- A piece of software
- It provides tools to manage (for embedded systems)
  - Processes, (or tasks)
  - Memory space

**What Is an Operating System?**

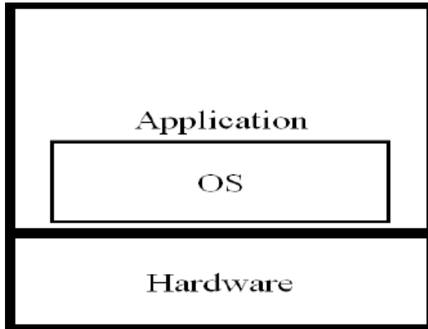
- What? It is a program (software) that acts as an intermediary between a user of a computer and the computer hardware.
- Why? Make the use of a computer CONVENIENT and EFFICIENT.

**What Is an Operating System?*For an Embedded System***

- Provides software tools for a convenient and prioritized control of tasks.

- ☐ Provides tools for task (process) synchronization.
- ☐ Provides a simple memory management system

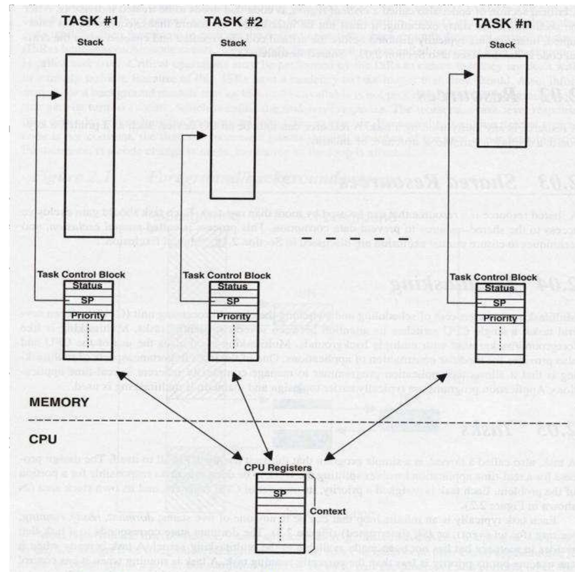
**Abstract View of A System (*Embedded System*):**




**Process/Task Concept:**

- ☐ Process is a program in execution; process execution must progress in sequential fashion
- ☐ A process includes:
  - ■ program counter
  - ■ stack
  - ■ data section

**Multitasking:**



**Process/Task Concept:**

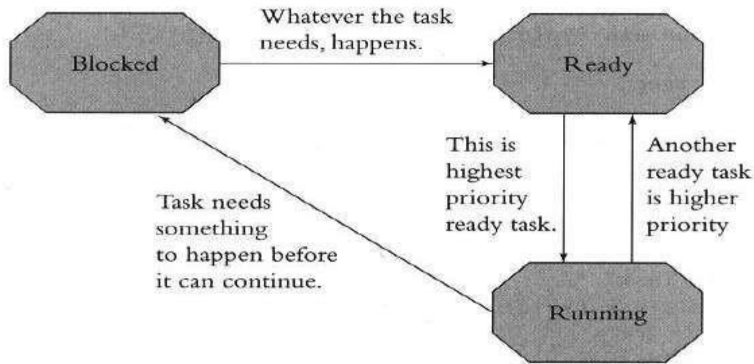
- Task States:
  - Running: Instructions are being executed
  - Ready: The process is waiting to be assigned to a process
  - Blocked: The process is waiting for some event to occur
  - terminated: The process has finished execution
  - new: The process is being created

**Task states:**

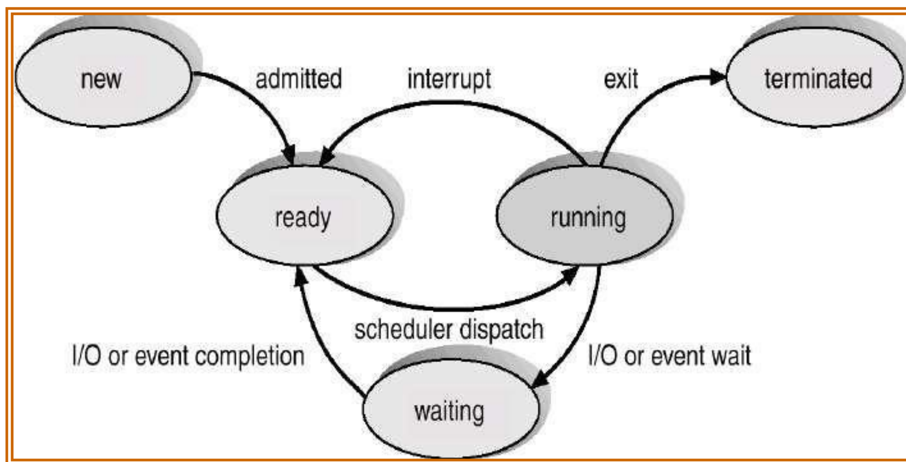
**Tasks and Task States:**

- A task – a simple subroutine
- ES application makes calls to the RTOS functions to start tasks, passing to the OS, start address, stack pointers, etc. of the tasks
- Task States:
  - Running
  - Ready (possibly: suspended, pended)
  - Blocked (possibly: waiting, dormant, delayed)
  - [Exit]
  - Scheduler – schedules/shuffles tasks between Running and Ready states
  - Blocking is self-blocking by tasks, and moved to Running state via other tasks' interrupt signaling (when block-factor is removed/satisfied)
- When a task is unblocked with a higher priority over the _running' task, the scheduler _switches' context immediately (for all pre-emptive RTOSs)

Figure 6.1 Task States




**Task State Transitions:**




Tasks – 1:

- Issue – Scheduler/Task signal exchange for block-unblock of tasks via function calls
- Issue – All tasks are blocked and scheduler idles forever (not desirable!)
- Issue – Two or more tasks with same priority levels in Ready state (time-slice, FIFO)
- Example: scheduler switches from processor-hog vLevelsTask to vButtonTask (on user interruption by pressing a push-button), controlled by the main() which initializes the RTOS, sets priority levels, and starts the RTOS

Figure 6.2 Uses for Tasks

```
/* "Button Task" */
void vButtonTask (void)    /* High priority */
{
   while (TRUE)
   {
      !! Block until user pushes a button
      !! Quick: respond to the user
   }
}

/* "Levels Task" */
void vLevelsTask (void)    /* Low priority */
{
   while (TRUE)
   {
      !! Read levels of floats in tank
      !! Calculate average float level
```

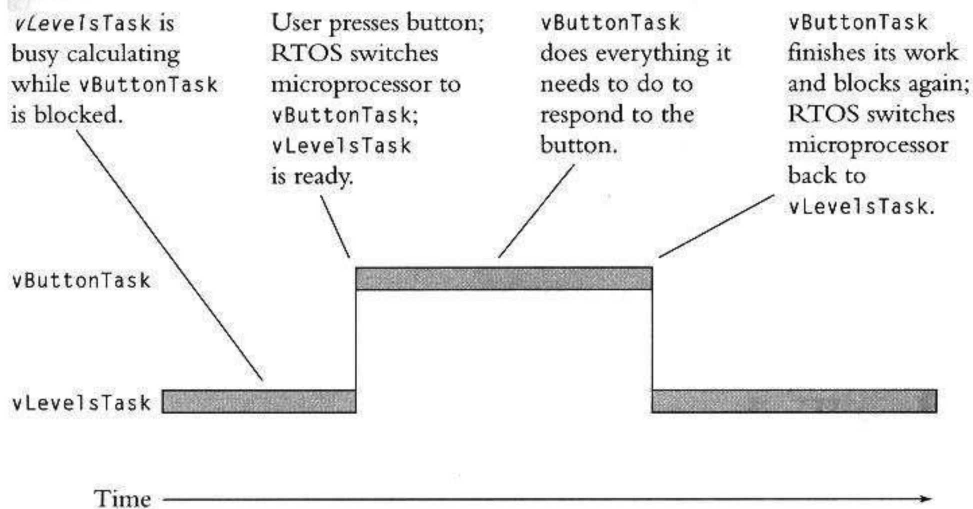
*(continued)*

Figure 6.2 *(continued)*

```
      !! Do some interminable calculation
      !! Do more interminable calculation
      !! Do yet more interminable calculation

      !! Figure out which tank to do next
   }
}
```

Figure 6.3 Microprocessor Responds to a Button under an RTOS




**Tasks and Data:**

- Each tasks has its won context - not shared, private registers, stack, etc.
- In addition, several tasks share common data (via global data declaration; use of _extern‘ in one task to point to another task that declares the shared data
- Shared data caused the _shared-data problem‘ without solutions discussed in Chp4 or use of _Reentrancy‘ characterization of functions

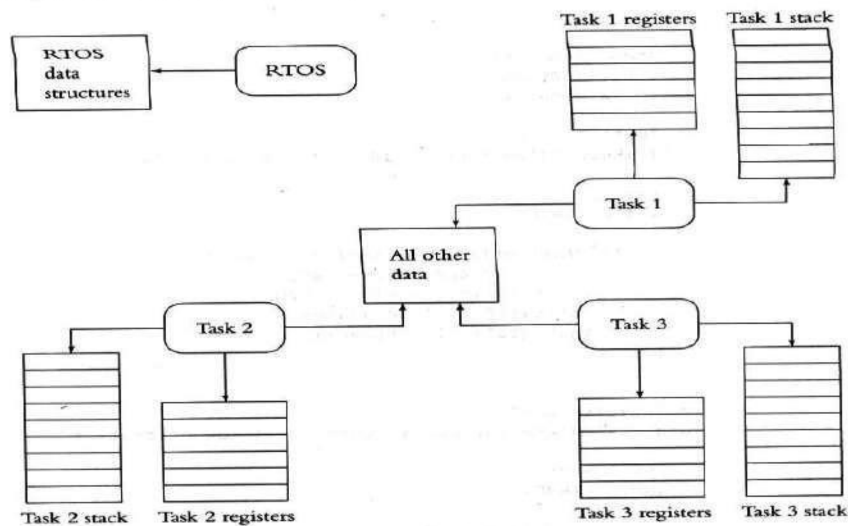



Figure 6.5 Data in an RTOS-Based Real-Time System

**Semaphores and Shared Data** – A new tool for atomicity

- Semaphore – a variable/lock/flag used to control access to shared resource(to avoid shared-data problems in RTOS)
- Protection at the start is via primitive function, called ***take***, indexed by the semaphore
- Protection at the end is via a primitive function, called ***release***, also indexed similarly
- Simple semaphores – Binary semaphores are often adequate for shared data problems in RTOS

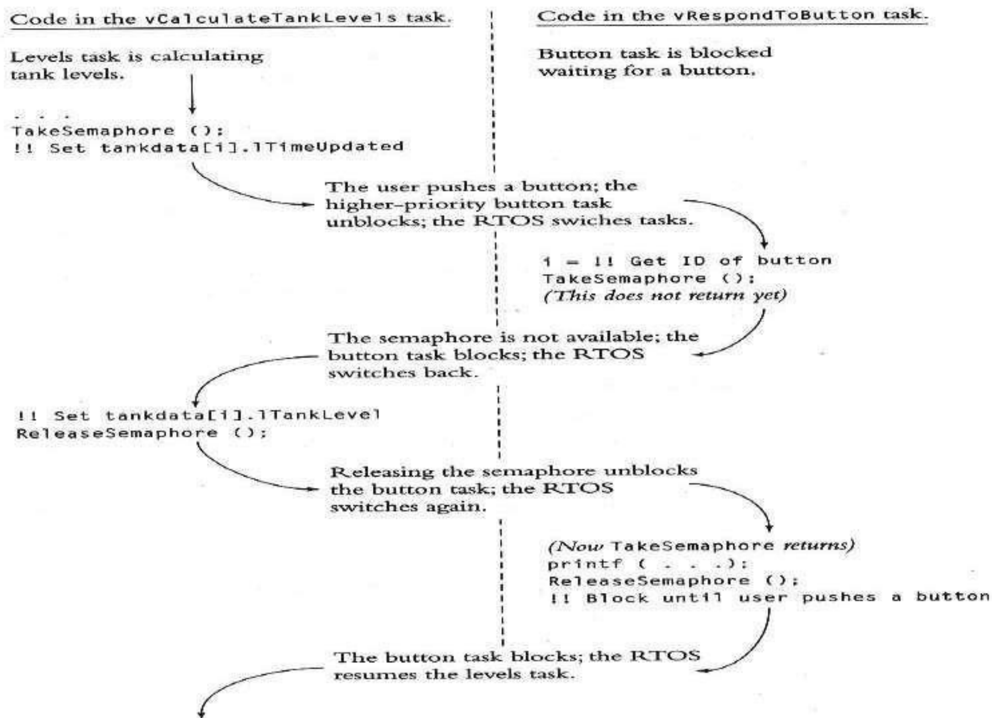


Figure 6.13 Execution Flow with Semaphores

**Semaphores and Shared Data – 1:**

- RTOS Semaphores & Initializing Semaphores
- Using binary semaphores to solve the _tank monitoring‘ problem
- The nuclear reactor system: The issue of initializing the semaphore variable in a dedicated task (not in a _competing‘ task) before initializing the OS – timing of tasks and priority overrides, which can undermine the effect of the semaphores
- Solution: Call OSSemInit() before OSInit()

Figure 6.14 Semaphores Protect Data in the Nuclear Reactor

```
#define TASK_PRIORITY_READ 11
#define TASK_PRIORITY_CONTROL 12
#define STK_SIZE 1024
static unsigned int ReadStk [STK_SIZE];
static unsigned int ControlStk [STK_SIZE];

static int iTemperatures[2];
OS_EVENT *p_semTemp;

void main (void)
{
   /* Initialize (but do not start) the RTOS */
   OSInit ();

   /* Tell the RTOS about our tasks */
   OSTaskCreate (vReadTemperatureTask,  NULLP,
      (void *)&ReadStk[STK_SIZE], TASK_PRIORITY_READ);
   OSTaskCreate (vControlTask,  NULLP,
      (void *)&ControlStk[STK_SIZE], TASK_PRIORITY_CONTROL);

   /* Start the RTOS.  (This function never returns.) */
   OSStart ();
}

void vReadTemperatureTask (void)
{
   while (TRUE)
   {
      OSTimeDly (5); /* Delay about 1/4 second */

      OSSemPend (p_semTemp, WAIT_FOREVER);
      !! read in iTemperatures[0];
      !! read in iTemperatures[1];
      OSSemPost (p_semTemp);
   }
}

void vControlTask (void)
{
   p_semTemp = OSSemInit (1);
   while (TRUE)
   {
      OSSemPend (p_semTemp, WAIT_FOREVER);
      if (iTemperatures[0] != iTemperatures[1])
         !! Set off howling alarm;
      OSSemPost (p_semTemp);

      !! Do other useful work
   }
}
```

**Semaphores and Shared Data – 2**

- Reentrancy, Semaphores, Multiple Semaphores, Device Signaling,
- a reentrant function, protecting a shared data, cErrors, in critical section
- Each shared data (resource/device) requires a separate semaphore for individual protection, allowing multiple tasks and data/resources/devices to be shared exclusively, while allowing efficient implementation and response time
- example of a printer device signaled by a report-buffering task, via semaphore signaling, on each print of lines constituting the formatted and buffered report

Figure 6.15 Semaphores Make a Function Reentrant

```
void Task1 (void)
{
   :
   vCountErrors (9);
   :
}

void Task2 (void)
{
   :
   vCountErrors (11);
   :
}

static int cErrors;
static NU_SEMAPHORE semErrors;

void vCountErrors (int cNewErrors)
{
   NU_Obtain_Semaphore (&semErrors, NU_SUSPEND);
   cErrors += cNewErrors;
   NU_Release_Semaphore (&semErrors);
}
```

Figure 6.16 Using a Semaphore as a Signaling Device

```
/* Place to construct report. */
static char a_chPrint[10][21];

/* Count of lines in report. */
static int iLinesTotal;

/* Count of lines printed so far. */
static int iLinesPrinted;

/* Semaphore to wait for report to finish. */
static OS_EVENT *semPrinter;
```

```
void vPrinterTask(void)
{
   BYTE byError;   /* Place for an error return. */
   Int wMsg;

   /* Initialize the semaphore as already taken. */
   semPrinter = OSSemInit(0);

   while (TRUE)
   {
      /* Wait for a message telling what report to format. */
      wMsg = (int) OSQPend (QPrinterTask, WAIT_FOREVER, &byError);

      !! Format the report into a_chPrint
      iLinesTotal = !! count of lines in the report

      /* Print the first line of the report */
      iLinesPrinted = 0;
      vHardwarePrinterOutputLine (a_chPrint[iLinesPrinted++]);

      /* Wait for print job to finish. */
      OSSemPend (semPrinter, WAIT_FOREVER, &byError);
   }
}
```

*(continued)*

Figure 6.16 *(continued)*

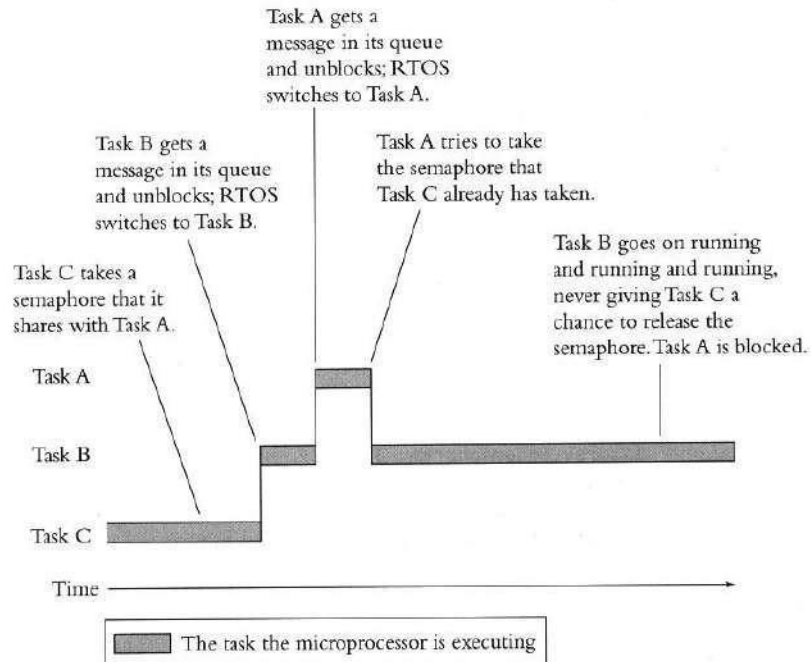
```
void  vPrinterInterrupt (void)
{
   if (iLinesPrinted == iLinesTotal)
      /* The report is done.  Release the semaphore. */
      OSSemPost (semPrinter);

   else
      /* Print the next line. */
      vHardwarePrinterOutputLine (a_chPrint[iLinesPrinted++]);
}
```

**semaphores and Shared Data – 3:**

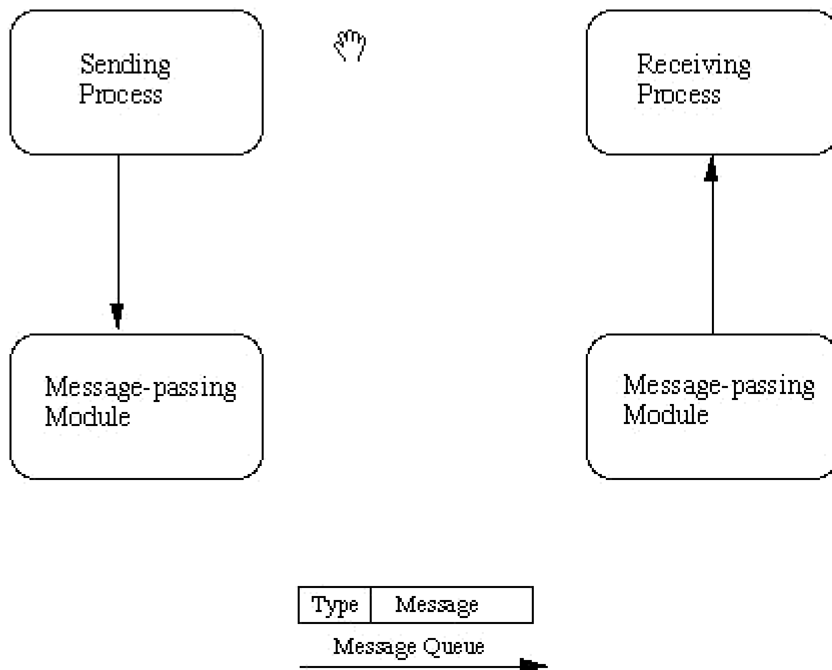
- Semaphore Problems – _Messing up‘ with semaphores
  - The initial values of semaphores – when not set properly or at the wrong place
  - The _symmetry‘ of takes and releases – must match or correspond – each _take‘ must have a corresponding _release‘ somewhere in the ES application
  - _Taking‘ the wrong semaphore unintentionally (issue with multiple semaphores)
  - Holding a semaphore for too long can cause _waiting‘ tasks‘ deadline to be missed
  - Priorities could be _inverted‘ and usually solved by _priority inheritance/promotion‘

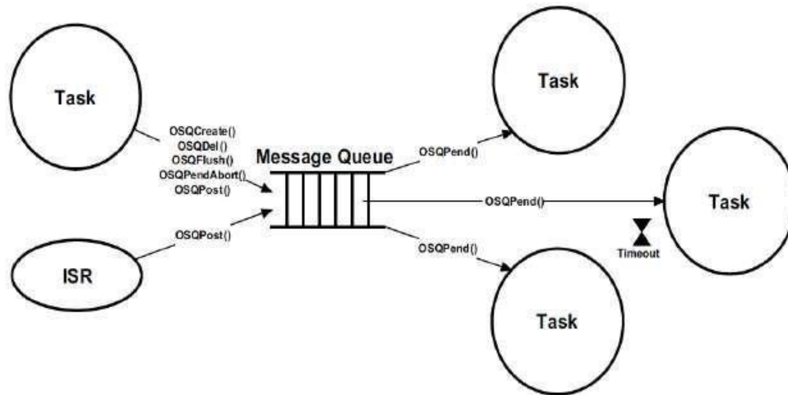
Figure 6.17 Priority Inversion




**message queue :**

Two (or more) processes can exchange information via access to a common system message queue. The *sending* process places via some (OS) message-passing module a message onto a queue which can be read by another process (Figure )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 (subject to other permissions -- see below).






**Basic Message Passing** IPC messaging lets processes send and receive messages, and queue messages for processing in an arbitrary order. Unlike the file byte-stream data flow of pipes, each IPC message has an explicit length. Messages can be assigned a specific type. Because of this, a server process can direct message traffic between clients on its queue by using the client process PID as the message type. For single-message transactions, multiple server processes can work in parallel on transactions sent to a shared message queue.

Before a process can send or receive a message, the queue must be initialized (through the `msgget` function see below) Operations to send and receive messages are performed by the `msgsnd()` and `msgrcv()` functions, respectively.

When a message is sent, its text is copied to the message queue. The `msgsnd()` and `msgrcv()` functions can be performed as either blocking or non-blocking operations. Non-blocking operations allow for asynchronous message transfer -- the process is not suspended as a result of sending or receiving a message. In blocking or synchronous message passing the sending process cannot continue until the message has been transferred or has even been acknowledged by a receiver. IPC signal and other mechanisms can be employed to implement such transfer. A blocked message operation remains suspended until one of the following three conditions occurs:

- The call succeeds.
- The process receives a signal.
- The queue is removed.

**Initialising the Message Queue :**

The `msgget()` function initializes a new message queue:

```
int msgget(key_t key, int msgflg)
```

It can also return the message queue ID (`msqid`) of the queue corresponding to the key argument. The value passed as the `msgflg` argument must be an octal integer with settings for the queue's permissions and control flags.

The following code illustrates the `msgget()` function.

```
#include <sys/ipc.h>;
#include <sys/msg.h>;

...


key_t key; /* key to be passed to msgget() */
int msgflg /* msgflg to be passed to msgget() */
int msqid; /* return value from msgget() */

...
key = ...
msgflg = ...

if ((msqid = msgget(key, msgflg)) == &ndash;1)
  {
    perror("msgget: msgget failed");
    exit(1);
   } else
    (void) fprintf(stderr, &ldquo;msgget succeeded");
```

**Mailbox:**

- Mailbox (for message) is an IPC through a message-block at an OS that can be used only by a single destined task.
- 
- A task on an OS function call puts (means post and also send) into the mailbox nly a pointer to a mailbox message
- Mailbox message may also include a header to identify the message-type specification.

**Mailbox IPC features:**

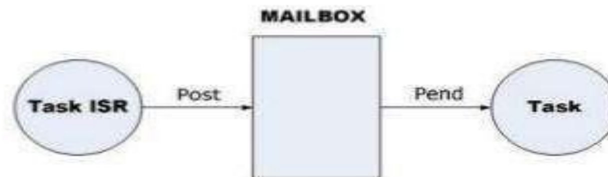
•OS provides for inserting and deleting message into the mailbox message- pointer. Deleting eans message-pointer pointing to Null.

•Each mailbox for a message need initialization (creation) before using the functions in the scheduler for the message queue and message pointer pointing to null

ESC Embedded Systems Committee

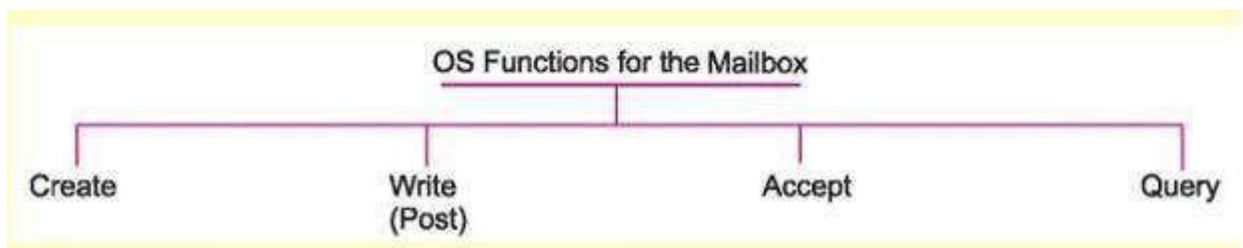
# Intertask Communication

- Mailboxes
  - Any task can send a message to a mailbox and any task can receive a message from a mailbox




Copyright © 2012 Embedded Systems Committee

**Mailbox Related Functions at the OS:**




**Pipe Function:**

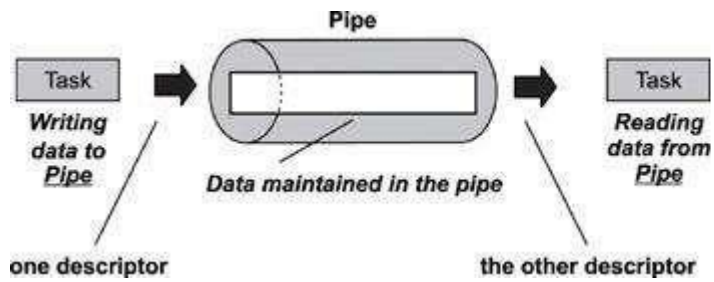
**Pipe**

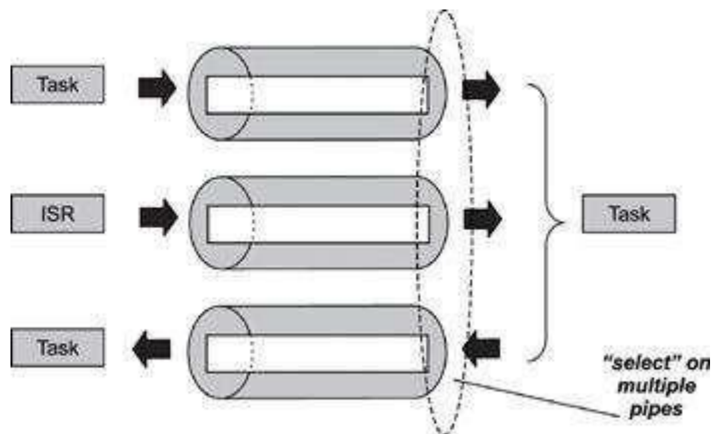
- Pipe is a device used for the interprocess communication
- Pipe has the functions create, connect and delete and functions similar to a device driver

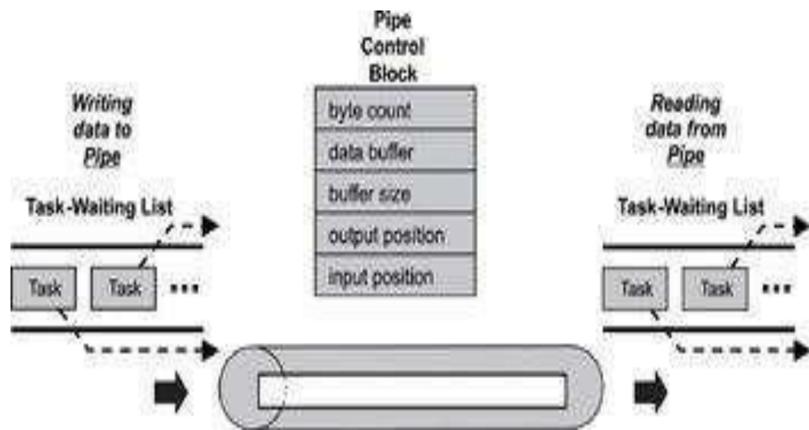
**Writing and reading a Pipe:**

• A message-pipe─ a device for inserting (writing) and deleting (reading) from that between two given inter-connected tasks or two sets of tasks.

• Writing and reading from a pipe is like using a C command *fwrite with a file name* to write into a named file, and C command *fread with a file name* to read into a named




Pipe function calls:

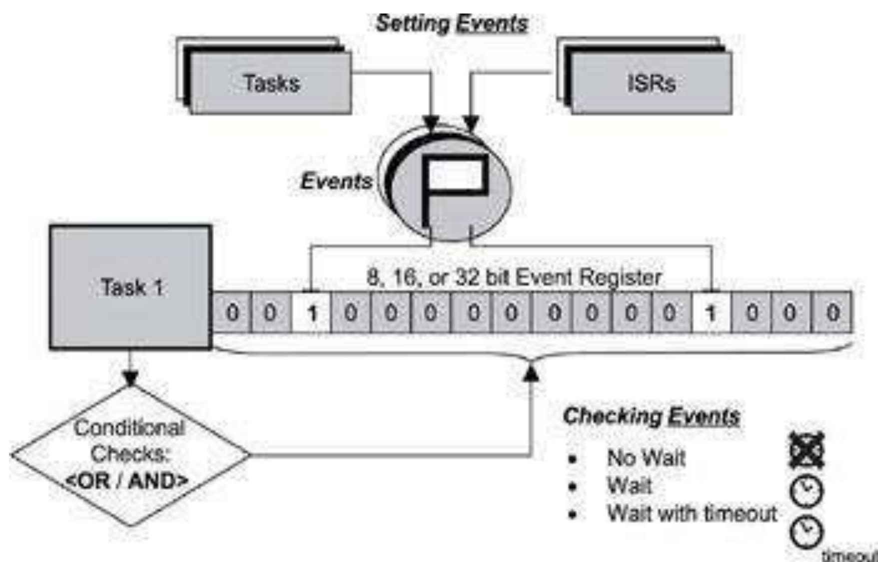
- Create a pipe
- Open pipe
- Close pipe
- Read from the pipe
- Write to the pipe

**Event Functions:**

- Wait for only one event (semaphore or mailboxmessage posting event)
- Event related OS functions can wait for number of events before initiating an action or wait for any of the predefined set of events
- Events for wait can be from different tasks or the ISRs

## Event functions at OS:

Some OSes support and some don't support event functions for a group of event




**Event registers function calls:**

- Create an event register
- Delete an event register
- Query an event register
- Set an event register
- Clear an event register

- Each bit I an event register can be used to obtain the states of an event .

- A task can have an event register and other tasks can set/clear the bits in the event register

**Signal:**

- one way for messaging is to use an OS function signal ( ).
- Provided in Unix, Linux and several RTOSes.
- Unix and Linux OSes use signals profusely and have thirty-one different types of signals for the various events.
- A signal is the software equivalent of the flag at a register that sets on a hardware interrupt. Unless masked by a signal mask, the signal allows the execution of the Signal handling function and allows the handler to run just as a hardware interrupt allows the execution of an ISR
- Signal provides the shortest communication.

**Signal management fuction calls:**

- Install a signal handler
- Remove an installed signal handler
- Send a signal to another task
- Block a signal from being delivered
- Unblock a blocked signal
- Ignore a signal

**Timers:**

- Real time clock ─ system clock, on each tick SysClkIntr interrupts
- Based on each SysClkIntr interrupts─ there are number of OS timer functions
- Timer are used to message the elasped time ofevents for instance , the kernel has to keep track of different times

The following functions calls are provided to manage the timer

- Get time
- Set time
- Time delay( in system clock)
- Time delay( in sec.)
- Reset timer

**Memory management:**

**Memory allocation:**

- Memory allocation When a process is created, the memory manager allocates the memory addresses (blocks) to it by mapping the process address space.
- Threads of a process share the memory space of the process

**Memory Managing Strategy for a system**

- Fixed

- blocks allocation
- Dynamic
- blocks Allocation
- Dynamic Page
- Allocation
- Dynamic Data memory Allocation

**Interrupt service routine (ISR):**

- Interrupt is a hardware signal that informs the cpu that an important event has occurred when interrupt occured, cpu saves its content and jumps to the ISR
- In RTOS
    - Interrupt latency
    - Interrupt response
    - Interrupt recovery

**Mutex:**

Mutex standards for mutual exclusion ,mutex is the general mechanism used for both rsource synchronization as well as task synchronization

**It has following mechanisms**

- Disabling the scheduler
- Disabling the interrupts
- By test and set operations
- Using semaphore

**UNIT-III**
**OBJECTS, SERVICES AND I/O**

*pipes* are kernel objects that provide unstructured data exchange and facilitate synchronization among tasks. In a traditional implementation, a pipe is a unidirectional data exchange facility, as shown in Figure Two descriptors, one for each end of the pipe (one end for reading and one for writing), are returned when the pipe is created. Data is written via one descriptor and read via the other. The data remains in the pipe as an unstructured byte stream. Data is read from the pipe in FIFO order.

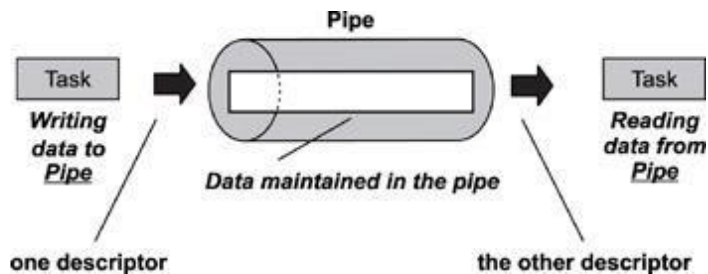


Figure 8.1: A common pipe-unidirectional.

A pipe provides a simple data flow facility so that the reader becomes blocked when the pipe is empty, and the writer becomes blocked when the pipe is full. Typically, a pipe is used to exchange data between a data-producing task and a data-consuming task, as shown in Figure 8.2. It is also permissible to have several writers for the pipe with multiple readers on it.

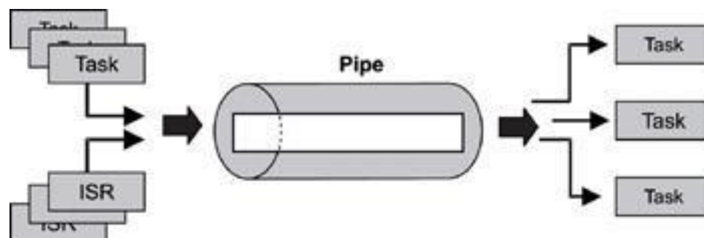


Figure 8.2: Common pipe operation.

Note that a pipe is conceptually similar to a message queue but with significant differences. For example, unlike a message queue, a pipe does not store multiple messages. Instead, the data that it stores is not structured, but consists of a stream of bytes. Also, the data in a pipe cannot be prioritized; the data flow is strictly first-in, first-out FIFO. Finally, as is described below, pipes support the powerful select operation, and message queues do not.

Some kernels provide a special register as part of each task‘s control block, as shown in Figure 8.7. This register, called an *event register,* is an object belonging to a task and consists of a group of binary event flags used to track the occurrence of specific events. Depending on a given kernel‘s implementation of this mechanism, an event register can be 8-, 16-, or 32-bits wide, maybe even more. Each bit in the event register is treated like a binary flag (also called an event flag) and can be either set or cleared.

Through the event register, a task can check for the presence of particular events that can control its execution. An external source, such as another task or an ISR, can set bits in the event register to inform the task that a particular event has occurred.

Applications define the event associated with an event flag. This definition must be agreed upon between the event sender and receiver using the event register.

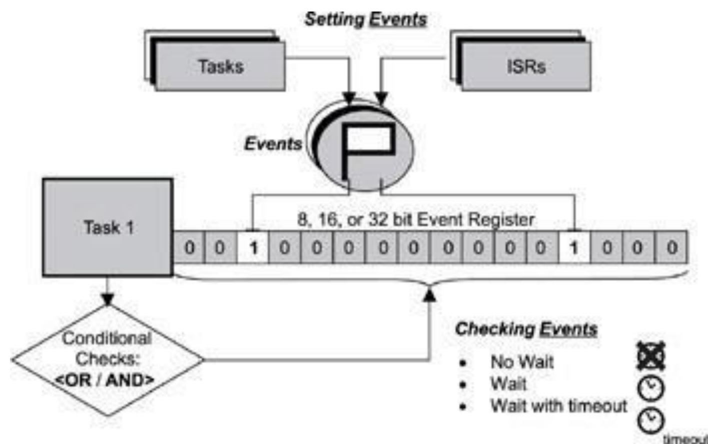



Figure 8.7: Event register.

In this RTOS Revealed, I am going to look at signals, which are the simplest method of inter-task communication supported by Nucleus SE. They provide a very low cost means of passing simple messages between tasks.

Using Signals

Signals are different from all the other types of kernel object in that they are not autonomous – signals are associated with tasks and have no independent existence. If signals are configured for an application, each task has a set of eight signal flags.

Any task can set the signals of another task. Only the owner task can read the signals. The read is destructive – i.e. the signals are cleared by the process of reading. No other task can read or clear a task's signals.

There is a facility in Nucleus RTOS that enables a task to nominate a function that is run when another task sets one or more of its signal flags. This is somewhat analogous to an interrupt service routine. This capability is not supported in Nucleus SE; tasks need to interrogate their signal flags explicitly.

Configuring Signals

As with most aspects of Nucleus SE, the configuration of signals is primarily controlled by **#define** statements in **nuse_config.h**. The key setting is **NUSE_SIGNAL_SUPPORT**, which enables the facility (for all tasks in the application). There is no question of specifying the number of signals – there is simply a set of eight flags for each task in the application.

Setting this enable parameter is the –master enable‖ for signals. This causes a data structure to be defined and sized accordingly, of which more later in this article. It also activates the API enabling settings.

**API Enables**

Every API function (service call) in Nucleus SE has an enabling **#define** symbol in **nuse_config.h**. For signals, these are:

**NUSE_SIGNALS_SEND**
**NUSE_SIGNALS_RECEIVE**

By default, both of these are set to **FALSE**, thus disabling each service call and inhibiting the inclusion of any implementation code. To configure signals for an application, you need to select the API calls that you want to use and set their enabling symbols to **TRUE**.

Here is an extract from the default **nuse_config.h** file:

```
#define NUSE_SIGNAL_SUPPORT FALSE /* Enables support for signals */
#define NUSE_SIGNALS_SEND    FALSE /* Service call enabler */
#define NUSE_SIGNALS_RECEIVE FALSE /* Service call enabler */
```

A compile time error will result if a signals API function is enabled and the signals facility has not been enabled. If your code uses an API call, which has not been enabled, a link time error will result, as no implementation code will have been included in the application. Of course, the enabling of the two API functions is somewhat redundant, as there would be no point in enabling signals support and not having these APIs available. The enables are included for compatibility with other Nucleus SE features.

Signals Service Calls

Nucleus RTOS supports four service calls that appertain to signals, that provide the following functionality:

- Send signals to a specified task. Implemented by **NUSE_Signals_Send()** in Nucleus SE.
- Receive signals. Implemented by **NUSE_Signals_Receive()** in Nucleus SE.
- Register a signal handler. Not implemented in Nucleus SE.
- Enable/disable (control) signals. Not implemented in Nucleus SE.

The implementation of each of these service calls is examined in detail.

Signals Send and Receive Services

The fundamental operations, that can be performed on a task‘s set of signals, are sending data to it (which may be done by any task) and reading data from it (and thus clearing the data, which may only be done by the owner). Nucleus RTOS and Nucleus SE each provide two basic API calls for these operations, that will be discussed here.

Since signals flags are bits, they are best visualized as binary numbers. As standard C does not historically support a representation of binary constants (only octal and hexadecimal), the Nucleus SE distribution includes a useful header file – **nuse_binary.h** – which contains **#define** symbols of the form **b01010101** for all 256 8-bit values. Here is an extract from the **nuse_binary.h** file:

```
#define b00000000 ((U8) 0x00)
#define b00000001 ((U8) 0x01)
#define b00000010 ((U8) 0x02)
#define b00000011 ((U8) 0x03)
#define b00000100 ((U8) 0x04)
#define b00000101 ((U8) 0x05)
```

**Sending Signals**

Any task may send signals to any other task in the application. Sending signals involves setting one or more of the signal flags. This is an OR operation that has no effect upon flags set previously.

***Nucleus RTOS API Call for Sending Signals***

Service call prototype:

**STATUS NU_Send_Signals(NU_TASK *task, UNSIGNED signals);**

Parameters:

**task** – pointer to control block of the task that owns the signal flags to be set

**signals** – the value of the signal flags to be set

Returns:

**NU_SUCCESS** – the call was completed successfully

**NU_INVALID_TASK** – the task pointer is invalid

***Nucleus SE API Call for Sending Signals***

This API call supports the key functionality of the Nucleus RTOS API.

Service call prototype:

**STATUS NUSE_Signals_Send(NUSE_TASK task, U8 signals);**

Parameters:

**task** – the index (ID) of the task that owns the signal flags to be set

**signals** – the value of the signal flags to be set

Returns:

**NUSE_SUCCESS** – the call was completed successfully

**NUSE_INVALID_TASK** – the task index is invalid

***Nucleus SE Implementation of Sending Signals***

Here is the complete code for the **NUSE_Signals_Send()** function:

```
STATUS NUSE_Signals_Send(NUSE_TASK task, U8 signals)
{
   #if NUSE_API_PARAMETER_CHECKING
      if (task >= NUSE_TASK_NUMBER)
      {
         return NUSE_INVALID_TASK;
      }
   #endif
   NUSE_CS_Enter();
   NUSE_Task_Signal_Flags[task] |= signals;
   NUSE_CS_Exit();
   return NUSE_SUCCESS;
```

}

The code is very simple. After any parameter checking, the signal values are ORed into the specified task's signal flags. Task blocking is not relevant to signals.

**Receiving Signals**

A task may only read its own set of signal flags. The process of reading them is destructive; i.e. it also results in the flags being cleared.

***Nucleus RTOS API Call for Receiving Signals***

Service call prototype:

**UNSIGNED NU_Receive_Signals(VOID);**

Parameters: none

Returns: the signals flags value

***Nucleus SE API Call for Receiving Signals***

This API call supports the key functionality of the Nucleus RTOS API.

Service call prototype:

**U8 NUSE_Signals_Receive(void);**

Parameters: none

Returns: the signal flags value

***Nucleus SE Implementation of Receiving Signals***

Here is the complete code for the **NUSE_Signals_Receive()** function:

```
U8 NUSE_Signals_Receive(void)
{
   U8 signals;
   NUSE_CS_Enter();
   signals = NUSE_Task_Signal_Flags[NUSE_Task_Active];
   NUSE_Task_Signal_Flags[NUSE_Task_Active] = 0;
   NUSE_CS_Exit();
   return signals;
}
```

The code is very simple. The flags value is copied, the original value cleared and the copy returned by the API function.

Task blocking is not relevant to signals.

The primary purpose of a real-time operating system (RTOS) is to manage CPU time, distribute it between a number of tasks, and provide services to enable management of other resources such

as peripherals. These functions are achieved in a variety of ways, but in most cases there is an opportunity for tasks to give a –hint‖ that they are able to relinquish the CPU for a while. Blocking system calls are one way to accomplish this. This article reviews how such calls work and when and how they are used.

**API calls**

At the heart of an RTOS is the kernel, which comprises the task scheduler and a number of services available to be called by application programs. Control of the scheduler and access to these services is by means of the kernel‘s application program interface (API). APIs differ from one RTOS to another, although there are some standards, like POSIX, but some characteristics are common to many RTOSes. One of those similarities is the function of blocking and non-blocking calls.

**Non-blocking calls**

A program may make an API call to request a specific resource or service. Such a call may normally return with the required result and/or a pointer to the requested resources. There may also be the possibility for an error return. But what if the call is valid but the resource or service cannot be provided at this time

**Blocking calls**

Instead of a task needing to manage the waiting process, when an API call returns an –unavailable‖ response, an RTOS typically can handle everything. Again, using the example of a Nucleus semaphore:

```
my_status = NU_Obtain_Semaphore(&my_semaphore, NU_SUSPEND);
```

In this case, as the suspend mode is set to NU_SUSPEND, the API call will not return until the semaphore can be obtained. In the meantime, the task is suspended or –blocked‖. When the circumstances change and the resource is available, the task is woken up; i.e. it is made ready to be scheduled according to its priority, which could be immediately.

**Timeout**

Some RTOSes, like Nucleus, offer greater flexibility with a further suspend option. Instead of choosing NU_SUSPEND or NU_NO_SUSPEND, a timeout value may be specified, thus:

```
my_status = NU_Obtain_Semaphore(&my_semaphore, 20);
```

This API call will return when the semaphore is obtained or after 20 clock ticks, whichever occurs first. If the return occurs after the timeout, my_status will have the value NU_TIMEOUT.

**Multiple blocked tasks**

What if several tasks are blocked pending the availability of a resource, which then becomes available? Which task is woken up to have its request satisfied? This depends upon the specific RTOS. Typically tasks are either woken in priority order (i.e. a higher priority task will be woken first) or they are woken in the order in which they suspended ("first in, first out"- FIFO). This may be a kernel configuration choice or may be selectable on a per-object basis.

With Nucleus, RTOS objects are created dynamically and the blocking behavior of each object is determined by a parameter in the creation call, which takes this form:

```
STATUS NU_Create_Semaphore(NU_SEMAPHORE *semaphore, CHAR *name,
UNSIGNED initial_count, OPTION suspend_type);
```

The last parameter determines how blocked tasks are woken up; the options are priority (NU_PRIORITY) or FIFO (NU_FIFO) order.

**To block or not to block**

The option to make blocking API calls simplifies the application code, as the RTOS takes the full responsibility for managing the CPU time utilization, which is, after all, its primary *raison d'être*. In the case of Nucleus RTOS, the API is orthogonal. Calls appertaining to numerous *types of objects – for example, semaphores, mailboxes, memory allocation, queues – all have blocking and non-blocking options.*

***Colin Walls*** has over thirty years of experience in the electronics industry, largely dedicated to embedded software. A frequent presenter at conferences and seminars, he is the author of numerous technical articles and the book *Embedded Software, Second Edition: The Works.* Colin is an embedded software technologist with Mentor Embedded [the Mentor Graphics Embedded Software Division], and is based in the UK. His regular blog is located at *blogs.mentor.com/colinwalls.* He may be reached by email at *colin_walls@mentor.com*.