# [exec and fork()](http://stackoverflow.com/questions/1653340/exec-and-fork)

The use of fork and exec exemplifies the spirit of UNIX in that it provides a very simple way to start new processes.

The fork call basically makes a duplicate of the current process, identical in *almost* every way (not everything is copied over, for example, resource limits in some implementations but the idea is to create as close a copy as possible).

The new process (child) gets a different process ID (PID) and has the the PID of the old process (parent) as its parent PID (PPID). Because the two processes are now running exactly the same code, they can tell which is which by the return code of fork - the child gets 0, the parent gets the PID of the child. This is all, of course, assuming the fork call works - if not, no child is created and the parent gets an error code.

The exec call is a way to basically replace the entire current process with a new program. It loads the program into the current process space and runs it from the entry point.

So, fork and exec are often used in sequence to get a new program running as a child of a current process. Shells typically do this whenever you try to run a program like find - the shell forks, then the child loads the find program into memory, setting up all command line arguments, standard I/O and so forth.

But they're not required to be used together. It's perfectly acceptable for a program to fork itself without execing if, for example, the program contains both parent and child code (you need to be careful what you do, each implementation may have restrictions). This was used quite a lot (and still is) for daemons which simply listen on a TCP port and fork a copy of themselves to process a specific request while the parent goes back to listening.

Similarly, programs that know they're finished and just want to run another program don't need to fork,exec and then wait for the child. They can just load the child directly into their process space.

Some UNIX implementations have an optimized fork which uses what they call copy-on-write. This is a trick to delay the copying of the process space in fork until the program attempts to change something in that space. This is useful for those programs using only fork and not exec in that they don't have to copy an entire process space.

If the exec *is* called following fork (and this is what happens mostly), that causes a write to the process space and it is then copied for the child process.

Note that there is a whole family of exec calls (execl, execle, execve and so on) but exec in context here means any of them.

The following diagram illustrates the typical fork/exec operation where the bash shell is used to list a directory with the ls command:

+--------+

| pid=7 |

| ppid=4 |

| bash |

+--------+

|

| calls fork

V

+--------+ +--------+

| pid=7 | forks | pid=22 |

| ppid=4 | ----------> | ppid=7 |

| bash | | bash |

+--------+ +--------+

| |

| waits for pid 22 | calls exec to run ls

| V

| +--------+

| | pid=22 |

| | ppid=7 |

| | ls |

V +--------+

+--------+ |

| pid=7 | | exits

| ppid=4 | <---------------+

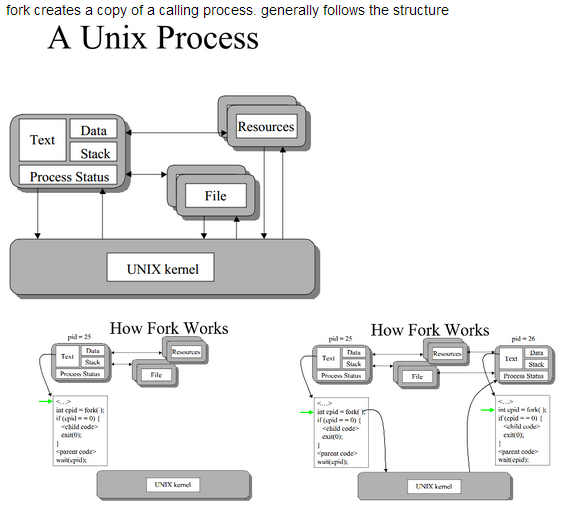
| bash |

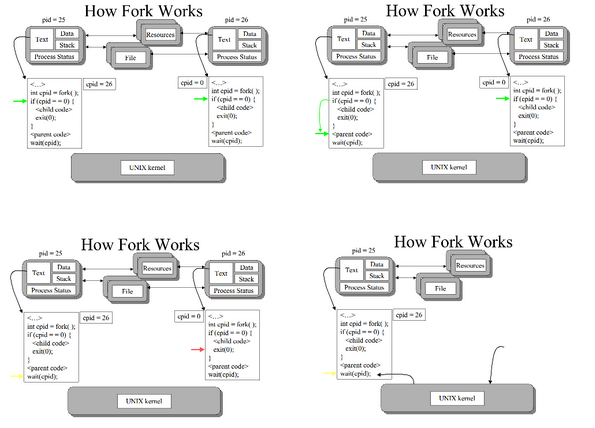
+--------+

|

| continues

V

****

****

int cpid = fork( );

if (cpid = = 0)

{

//child code

exit(0);

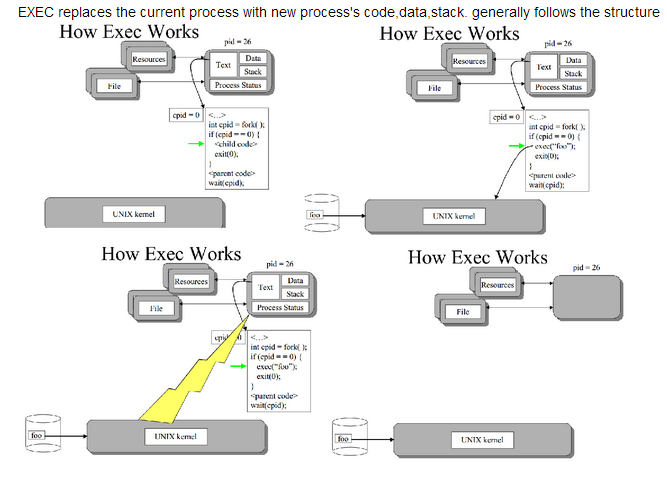
}

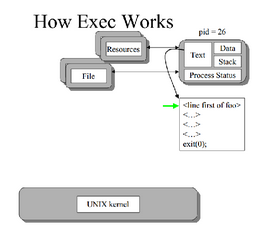
//parent code

wait(cpid);

// end

(for child process text(code),data,stack is same as calling process) child process executes code in if block.

****

****

cpid = fork( );

if (cpid = = 0)

{

//child code

exec(foo);

exit(0);

}

//parent code

wait(cpid);

// end

(after exec call unix kernel clears the child process text,data,stack and fills with foo process related text/data) thus child process is with different code (foo's code {not same as parent})

**Dup()**

The dup system call duplicates an existing file descriptor, returning a new one that refers to the same underlying I/O object.

Dup allows shells to implement commands like this:

ls existing-file non-existing-file > tmp1 2>&1

The 2>&1 tells the shell to give the command a file descriptor 2 that is a duplicate of descriptor 1.  
Now the error if we tried to do ls of non-existing file and the legal output of ls on existing file show up in temp1 file.

The following example code runs the program wc with standard input connected to the read end of a pipe.

int p[2];

char \*argv[2];

argv[0] = "wc";

argv[1] = 0;

pipe(p);

if(fork() == 0) {

close(STDIN); //CHILD CLOSING stdin

dup(p[STDIN]); // copies the fd of read end of pipe into its fd i.e 0 (STDIN)

close(p[STDIN]);

close(p[STDOUT]);

exec("/bin/wc", argv);

} else {

write(p[STDOUT], "hello world\n", 12);

close(p[STDIN]);

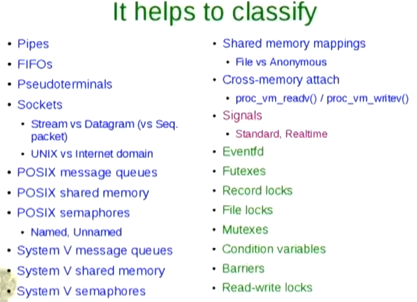
close(p[STDOUT]);

}

The child dups the read end onto file descriptor 0, closes the file de scriptors in p, and execs wc. When wc reads from its standard input, it reads from the pipe.  
This is how pipes are implemented using dup, well that one use of dup now you use pipe to build something else, that's the beauty of system calls,you build one thing after another using tools which are already there , these tool were inturn built using something else so on .. At the end system calls are the most basic tools you get in kernel

Both make a new file descriptor corresponding to an existing open file description. Most properties between the old and new fd (like position) are shared; the only property I can think of that's not shared is the close-on-exec flag. The difference between dup and dup2 is that dup assigns the lowest available file descriptor number, while dup2 lets you choose the file descriptor number that will be assigned and atomically closes and replaces it if it's already taken.

**IPC on Linux**

****

****

Communication: Things are about exchanging data.

Synchronization: Which is about co-ordinating activity of processes.

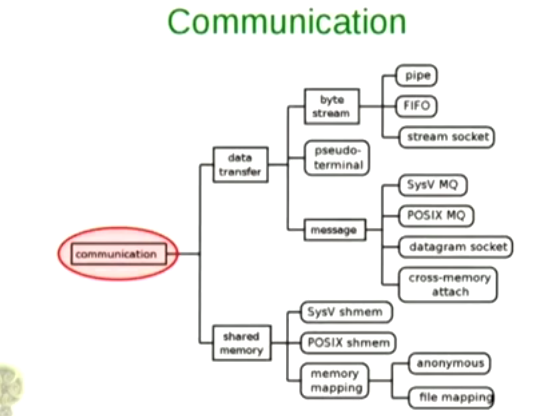
Signals: Don’t fall into either of above 2category, But can be both.

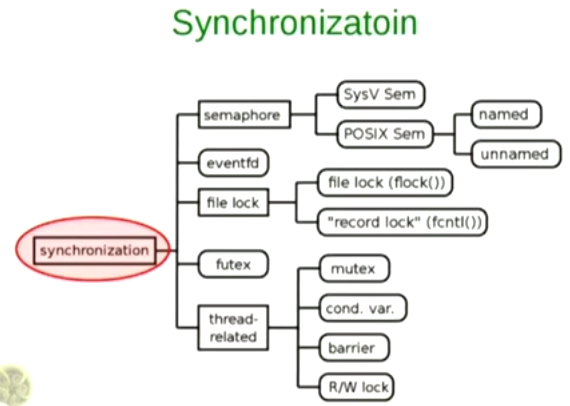
Ofcourse,

you can use communication methods for synchronization of processes

and

if we are tricky, you can use synchronization methods for communication of data.

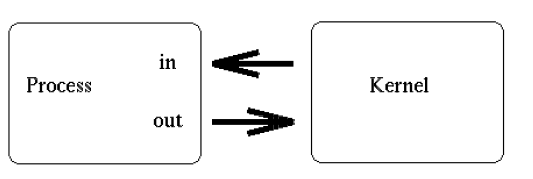
****

****

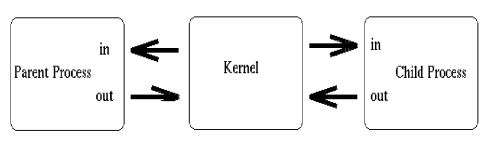
**Q) Is errno thread-safe?**

**A)** Yes, it is thread safe. On Linux, the global errno variable is thread-specific. POSIX requires that errno be threadsafe.

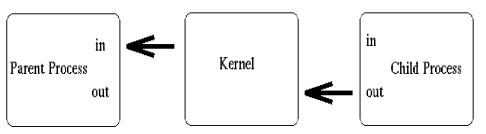
**Unnamed Pipe**

****

**After fork()**

****

**After closing unwanted descriptors**

****

[**Pipe vs. Temporary File**](http://stackoverflow.com/questions/6977561/pipe-vs-temporary-file)

One big difference is that with the pipe, processes A and B can be running concurrently, so that B gets to work on the output from A before A has finished producing it. Further, the size of the pipe is limited, so A won't be able to produce vastly more data than B has consumed; it will be made to wait for B to catch up.

If the volume of data is big, then writing to the temporary file involves disk activity, even if only for creating and then destroying the file. The data might well stay in the in-memory buffer pools - so no disk I/O there - even for surprisingly large files. Writing to the pipe 'never' involves writing to disk.

# [Why do I need to close fds when reading and writing to the pipe?](http://stackoverflow.com/questions/976015/why-do-i-need-to-close-fds-when-reading-and-writing-to-the-pipe)

Your pipe is a unidirections stream - with a file descriptor for each end. It is not neccessary to close() either end of the pipe to allow data to pass along it.

**if your pipe spans processes** (i.e. is created before a fork() and then the parent and child use it to communicate) you can have one write and and one read end. Then it is good practice to close the unwanted ends of the pipe. This will

* make sure that when the writing end closes the pipe it is seen by the read end. As an example, say the child is the write side, and it dies. If the parent write side has not been closed, then the parent will not get "eof" (zero length read()) from the pipe - because the pipe has a open write-end.
* make it clear which process is doing the writing and which process is doing the reading on the pipe.

**if your pipe spans threads** (within the same process), then do not close the unwanted ends of the pipe. This is because the file descriptor is held by the process, and closing it for one thread will close it for all threads, and therefore the pipe will become unusable.

There is nothing stopping you having one process writing continuously to the pipe and the other process reading. If this is a problem you are having then give us more details to help you out.

# Pipe the Easy Way!

The *popen*() function shall execute the command specified by the string *command*. It shall create a pipe between the calling program and the executed command, and shall return a pointer to a stream that can be used to either read from or write to the pipe.

The environment of the executed command shall be as if a child process were created within the *popen*() call using the [*fork*()](http://pubs.opengroup.org/onlinepubs/009696699/functions/fork.html) function, and the child invoked the *[sh](http://pubs.opengroup.org/onlinepubs/009696699/utilities/sh.html)* utility using the call:

execl(*shell path*, "sh", "-c", *command*, (char \*)0);

# If a calling process A does

pipe\_fp = popen("sort", "w")) == NULL)

# then scenario is:

# 

# Named Pipe

# [What are the advantages of using named pipe over unnamed pipe?](http://unix.stackexchange.com/questions/69057/what-are-the-advantages-of-using-named-pipe-over-unnamed-pipe)

Named pipes (fifo) have ~~four~~ three advantages I can think of:

* you don't have to start the reading/writing processes at the same time
* you can have multiple readers/writers which do not need common ancestry
* as a file you can control ownership and permissions
* ~~they are bi-directional, unnamed pipes may be unidirectional~~ \*

\*) Think of a standard shell|pipeline which is unidirectional, several shells (ksh,zsh, andbash) also offer[coprocesses](http://www.ict.griffith.edu.au/anthony/info/shell/co-processes.hints)which allow bi-directional communication. POSIX treats pipes as half-duplex (i.e. each side can only read or write), thepipe()system call returns two file handles and you may be required to treat one as read-only and the other as write-only. Some (BSD) systems support read and write simultaneously (not forbidden by POSIX), on others you would need two pipes, one for each direction. Check yourpipe(),popen()and possiblypopen2()man pages. The undirectionality may not be dependent on whether the pipe is named or not, though on Linux 2.6 it is dependent.

(Updated, thanks to feedback from Stephane Chazelas)

So one immediately obvious task you cannot achieve with an unnamed pipe is a conventional client/server application.

The last (stricken) point above about unidirectional pipes is relevant on Linux, POSIX (see [popen()](http://pubs.opengroup.org/onlinepubs/007904975/functions/popen.html)) says that a pipe need only be readable or writeable, on [Linux they are unidirectional](http://unix.stackexchange.com/questions/53641/how-to-make-bidirectional-pipe-between-two-programs). SeeUnderstanding The Linux Kernel (3rd Ed. O'Reilly) for Linux-specific details (p787). Other OS's offer bidirectional (unnamed) pipes.

As an example, Nagios uses a fifo for its [command file](http://nagios.sourceforge.net/docs/3_0/extcommands.html). Various external processes (CGI scripts, external checks, NRPE etc) write commands/updates to this fifo and these are processed by the persistent Nagios process.

Named pipes have features not unlike TCP connections, but there are important differences. Because a fifo has a persistent filesystem name you can write to it even when there is no reader, admittedly the writes will block (without async or non-blocking I/O), though you won't loose data if the receiver isn't started (or is being restarted).

# Difference between named & unnamed pipe

A named pipe works much like a regular pipe, but does have some noticeable differences.

* Named pipes exist as a device special file in the file system.
* Processes of different ancestry can share data through a named pipe.
* When all I/O is done by sharing processes, the named pipe remains in the file system for later use.

## Creating a FIFO

There are several ways of creating a named pipe. The first two can be done directly from the shell.

mknod MYFIFO p

mkfifo a=rw MYFIFO

The above two commands perform identical operations, with one exception. The mkfifo command provides a hook for altering the permissions on the FIFO file directly after creation. With mknod, a quick call to the chmod command will be necessary.

FIFO files can be quickly identified in a physical file system by the ``p'' indicator seen here in a long directory listing:

$ ls -l MYFIFO

prw-r--r-- 1 root root 0 Dec 14 22:15 MYFIFO|

Also notice the vertical bar (``pipe sign'') located directly after the file name. Another great reason to run Linux, eh?

To create a FIFO in C, we can make use of the mknod() system call:

LIBRARY FUNCTION: mknod();

PROTOTYPE: int mknod( char \*pathname, mode\_t mode, dev\_t dev);

RETURNS: 0 on success,

-1 on error: errno = EFAULT (pathname invalid)

EACCES (permission denied)

ENAMETOOLONG (pathname too long)

ENOENT (invalid pathname)

ENOTDIR (invalid pathname)

(see man page for mknod for others)

NOTES: Creates a filesystem node (file, device file, or FIFO)

I will leave a more detailed discussion of mknod() to the man page, but let's consider a simple example of FIFO creation from C:

mknod("/tmp/MYFIFO", S\_IFIFO|0666, 0);

In this case, the file ``/tmp/MYFIFO'' is created as a FIFO file. The requested permissions are ``0666'', although they are affected by the umask setting as follows:

final\_umask = requested\_permissions & ~original\_umask

A common trick is to use the umask() system call to temporarily zap the umask value:

umask(0);

mknod("/tmp/MYFIFO", S\_IFIFO|0666, 0);

In addition, the third argument to mknod() is ignored unless we are creating a device file. In that instance, it should specify the major and minor numbers of the device file.

## FIFO Operations

I/O operations on a FIFO are essentially the same as for normal pipes, with once major exception. An ``open'' system call or library function should be used to physically open up a channel to the pipe. With half-duplex pipes, this is unnecessary, since the pipe resides in the kernel and not on a physical filesystem. In our examples, we will treat the pipe as a stream, opening it up with fopen(), and closing it with fclose().

Consider a simple server process:

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Excerpt from "Linux Programmer's Guide - Chapter 6"

(C)opyright 1994-1995, Scott Burkett

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

MODULE: fifoserver.c

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <stdio.h>

#include <stdlib.h>

#include <sys/stat.h>

#include <unistd.h>

#include <linux/stat.h>

#define FIFO\_FILE "MYFIFO"

int main(void)

{

FILE \*fp;

char readbuf[80];

/\* Create the FIFO if it does not exist \*/

umask(0);

mknod(FIFO\_FILE, S\_IFIFO|0666, 0);

while(1)

{

fp = fopen(FIFO\_FILE, "r");

fgets(readbuf, 80, fp);

printf("Received string: %s\n", readbuf);

fclose(fp);

}

return(0);

}

Since a FIFO blocks by default, run the server in the background after you compile it:

$ fifoserver&

We will discuss a FIFO's blocking action in a moment. First, consider the following simple client frontend to our server:

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Excerpt from "Linux Programmer's Guide - Chapter 6"

(C)opyright 1994-1995, Scott Burkett

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

MODULE: fifoclient.c

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <stdio.h>

#include <stdlib.h>

#define FIFO\_FILE "MYFIFO"

int main(int argc, char \*argv[])

{

FILE \*fp;

if ( argc != 2 ) {

printf("USAGE: fifoclient [string]\n");

exit(1);

}

if((fp = fopen(FIFO\_FILE, "w")) == NULL) {

perror("fopen");

exit(1);

}

fputs(argv[1], fp);

fclose(fp);

return(0);

}

## Blocking Actions on a FIFO

Normally, blocking occurs on a FIFO. In other words, if the FIFO is opened for reading, the process will "block" until some other process opens it for writing. This action works vice-versa as well. If this behavior is undesirable, the O\_NONBLOCK flag can be used in an open() call to disable the default blocking action.

In the case with our simple server, we just shoved it into the background, and let it do its blocking there. The alternative would be to jump to another virtual console and run the client end, switching back and forth to see the resulting action.

## The Infamous SIGPIPE Signal

On a last note, pipes must have a reader and a writer. If a process tries to write to a pipe that has no reader, it will be sent the SIGPIPE signal from the kernel. This is imperative when more than two processes are involved in a pipeline.

# --------------------------------------------------------------------

# System V IPC vs POSIX IPC

1. What are the differences between System V IPC and POSIX IPC ?
2. Why do we have two standards ?
3. How to decide which IPC functions to use ?

Both have the same basic tools -- semaphores, shared memory and message queues. They offer a slightly different interface to those tools, but the basic concepts are the same. One notable difference is that POSIX offers some notification features for message queues that Sys V does not. (See mq\_notify().)

Sys V IPC has been around for longer which has a couple of practical implications --

First, POSIX IPC is less widely implemented. I wrote a Python wrapper for POSIX IPC and [its documentation lists what I know about POSIX IPC implementations on various platforms](http://semanchuk.com/philip/posix_ipc/#platforms).

On all of the platforms listed in that documentation, Sys V IPC is completely implemented AFAIK, whereas you can see the POSIX IPC is not.

The second implication of their relative age is that POSIX IPC was designed after Sys V IPC had been used for a while. Therefore, the designers of the POSIX API were able to learn from the strengths and weaknesses of the Sys V API. As a result the POSIX API is simpler and easier to use IMO, and I recommend it over the Sys V API.

I should note that I've never run any performance tests to compare the two. I would think that the older API (Sys V) would have had more time to be performance tuned, but that's just speculation which is of course no substitute for real-world testing.

As to why there are two standards -- POSIX created their standard because they thought it was an improvement on the Sys V standard. But if everyone agreed that POSIX IPC is better, many many many programs still use Sys V IPC and it would take years to port them all to POSIX IPC. In practice, it would not be worth the effort so even if all new code used POSIX IPC as of tomorrow, Sys V IPC would stick around for many years.

We can't tell you which you should use without knowing a lot more about what you intend to do, but the answers you have here should give you enough information to decide on your own.

---------------------------------------------------------------------------------------------------------------------------------------

## General Thread Scheduling and Terminology using pthread library

### Contention Scope

There are two possible contention scopes. PTHREAD\_SCOPE\_SYSTEM and PTHREAD\_SCOPE\_PROCESS. They can be set with pthread\_attr\_setscope(). The scope of a thread can only be specified before the thread is created.

#### PTHREAD\_SCOPE\_SYSTEM

A thread that has a scope of PTHREAD\_SCOPE\_SYSTEM will content with other processes and other PTHREAD\_SCOPE\_SYSTEM threads for the CPU. That is if there is one process P1 with 10 threads with scope PTHREAD\_SCOPE\_SYSTEM and a single threaded process P2, P2 will get one timeslice out of 11 and every thread in P1 will get one timeslice out of 11. I.e. P1 will get 10 time more timeslices than P2.

#### PTHREAD\_SCOPE\_PROCESS

All threads of a process that have a scope of PTHREAD\_SCOPE\_PROCESS will be grouped together and this group of threads contents for the CPU. If there is a process with 4 PTHREAD\_SCOPE\_PROCESS threads and 4 PTHREAD\_SCOPE\_SYSTEM threds, then each of the PTHREAD\_SCOPE\_SYSTEM threads will get a fifth of the CPU and the other 4 PTHREAD\_SCOPE\_PROCESS threads will share the remaing fifth of the CPU. How the PTHREAD\_SCOPE\_PROCESS threads share their fifth of the CPU among themselves is determined by the scheduling policy and the thread's priority.

If there are other processes running, then every PTHREAD\_SCOPE\_SYSTEM and every group of PTHREAD\_SCOPE\_PROCESS threads (i.e. every process with PTHREAD\_SCOPE\_PROCESS threads) will be handled like a seperate process by the system scheduler.

### Priorities and Scheduling Policy

A PTHREAD\_SCOPE\_PROCESS thread has a priority. Whenever a thread is runnable and no other thread (of this process) has a higher priority the thread will get the CPU. Note that this might lead to starvation of other threads. When two or more runnable threads have the same priority and no other runnable thread has a higher priority, then the scheduling policy will determine which of these highest priority threads to run.

The priority is assigned staticly with pthread\_setschedparam(). The scheduler will not change the priority of a thread.

The scheduling policy can either be SCHED\_FIFO or SCHED\_RR. FIFO is a first come first serve policy. RR is a round robin policy that might preempt threads. But again, the policy only effects threads that have the same priority.

A more extensive description of priorites and policies can be found in [1] and [2]. Note that these documents discuss process scheduling, but the principle is the same.

**Note:** The priority and scheduling policy settings are meaningless when a thread has scope PTHREAD\_SCOPE\_SYSTEM.

### Realtime Process Scheduling

It is also possible to do realtime *process* scheduling. [2] explains how realtime process scheduling works. sched\_setscheduler() is used to set the process scheduling parameters.

### Nice values

The nice value of a process also influences the scheduling behaviour. A process (and the threads therein) with a lower nice value (i.e., higher priority) will get a higher share of the CPU time. Starting a program with nice works as expected. Using the nice() system-call from a threaded program has not been tested (the question is: does a nice() call effect the whole process or the current thread. This may well depend on the pthread imlementation and scope).

---------------------------------------------------------------------------------------------------------------------------------------

**Q) what is the difference between re-entrant function and recursive function?**

A) This concept is language agnostic. The term "re-entrant" means that it is safe to "**re-enter**" the function while it is already executed, typically in a concurrent environment.

In other words, when two tasks can execute the function at the same time without interfering with each other, then the function is re-entrant. A function is not re-entrant when the execution by one task has an impact on the influence of another task. This typically is the case when a global state or data is used. A function that uses only local variables and arguments is typically re-entrant.

There are three main reasons for that to occur: recursion (the function calls itself), multi-threading and interruption. Recursion is normally easier, since it is clear that the function will be re-entered. Multi-threading and interruption are more tricky, as the re-entrance will be asynchronous.

This means that the function cannot use static "global" data, since that data would then be accessed by two (or more) threads in parallel, often breaking horribly. A re-entrant function often has an explicit argument to hold any call-specific state, rather than storing it statically.

strtok() is a classic case of a function in the C standard library that is well-known *not* to be re-entrant.

A function may also be re-entered on the same thread as a result of recursion - either directly or indirectly (ie, function a calls function b which calls function c which calls function a).

Of course if you have protected against re-entrancy on the basis that multiple threads may call it then you are covered for the recursive cases too. That's not true the other way around, however.

==================================================================================

Q) Can pthread\_join() wait on itself?

#include <pthread.h>

int

main(int argc, char \*argv[])

{

void \*res;

int s;

printf("Message from main()\n");

s = pthread\_join(pthread\_self(), &res);

if (s != 0)

printf("pthread\_join(): %d",s);

printf("Thread returned %d\n", (int) res);

exit(0);

}

A)

You cannot join to yourself. The POSIX manpage for pthread\_join specifies that you may get a deadlock error:

[EDEADLK]

A deadlock was detected or the value of thread specifies the calling thread.

And, indeed, a search of that error shows that it's 35, at least on my system:

pax> cd /usr/include

pax> grep EDEADLK \*.h \*/\*.h \*/\*/\*.h

asm-generic/errno.h:#define EDEADLK 35 /\* Resource deadlock would occur \*/

While some deadlocks are subtle and difficult for pthreads to automatically detect, this one is relatively easy, with something like this at the start of the join function:

int pthread\_join(pthread\_t thread, void \*\*value\_ptr) {

if (thread == pthread\_self())

return EDEADLK;

// rest of function

}

=====================================================================================

Q) What pthread\_join(t, NULL) does?

A) Above call, still waits for the thread to finish but, since you provide nowhere for the exit value to be stored, it gets thrown away.

=====================================================================================