OPERATING SYSTEMS

PROCESS MANAGEMENT COMMANDS

Moving a process into the foreground (fg):

- You can use the fg command to resume the execution of a suspended job in the foreground or move a background job into the foreground.

fg [%job_id]

- Where job_id is the job ID (not process ID) of the suspended or background process.
- If %job_id is omitted, the current job is assumed.

PROCESS MANAGEMENT COMMANDS

Moving a process into the background (bg):

- You can use the bg command to put the current or a suspended process into the background

bg [%job_id]

- If %job_id is omitted, the current job is assumed.

PROCESS MANAGEMENT COMMANDS

Displaying the status of jobs (background and suspended processes)

You can use the jobs command to display the status of suspended and background processes.

PROCESS MANAGEMENT COMMANDS

Suspending a process:

- You can suspend a foreground process by pressing <Ctrl Z>, which sends a STOP/SUSPEND signal to the process.
- The shell displays a message saying that the job has been suspended and displays its prompt.
- You can then manipulate the state of this job, put it in the background with the bg command, run some other commands, and then eventually bring the job back into the foreground with the fg command.

PROCESS MANAGEMENT COMMANDS

Terminating a process:

- You can terminate a foreground process by pressing <Ctrl C>.
- Recall that this keypress sends the SIGINT signal to the process and the default action is the termination of the process.
- If your foreground process intercepts SIGINT and ignores it, you cannot terminate it with <Ctrl C>.

PROCESS MANAGEMENT COMMANDS

Killing a process:

- You can also terminate a process with the kill command. When executed, this command sends a signal to the process whose process ID is specified in the command line.
- Here is the syntax of the command.

kill [-signal] PID

PROCESS MANAGEMENT COMMANDS

Killing a process:

- Here "signal" is the signal number and PID is the process ID of the process to whom the specified signal is to be sent.
- For example, kill –2 1234 command sends the signal number 2 (which is also called SIGINT) to the process with ID 1234.
- The default action for a signal is the termination of the process identified in the command line.
- When executed without a signal number, the command sends the SIGTERM signal to the process.
- A process that has been coded to intercept and ignore a signal, can be terminated by sending it the "sure kill" signal, SIGKILL, whose signal number is 9, as in kill –9 1234.
- You can display all of the signals supported by your system, along with their numbers, by using the kill –l command.

MULTI-THREADING

9

WHY WE NEED MULTI-THREADING

- There are two main issues with processes:

1. The fork() system call is expensive (it requires memory to memory copy of the executable image of the calling process and allocation of kernel resources to the child process).
2. An inter-process communication channel (IPC) is required to pass information between a parent process and its children processes.

- These problems can be overcome by using threads.

MULTI-THREADING

- A thread sometimes called a lightweight process (LWP), is a basic unit of CPU utilization and executes within the address space of the process that creates it.
- A traditional (heavyweight) process has a single thread of control.
- If a process has multiple threads of control, it can do more than one task at a time.

MULTI-THREADING

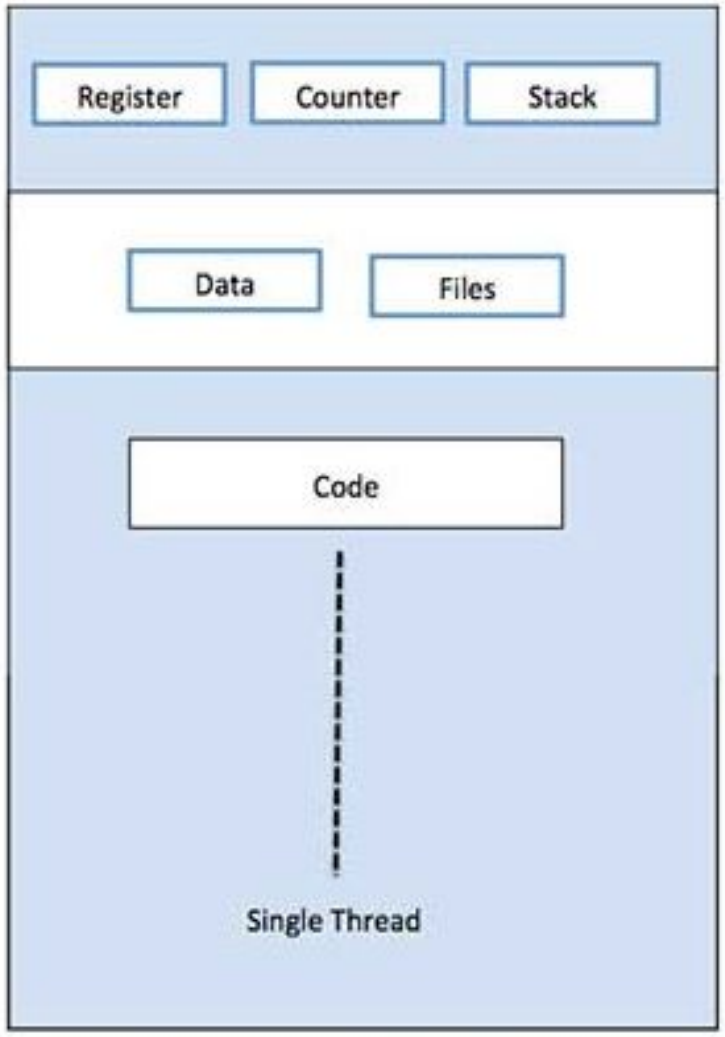
Threads have their own:

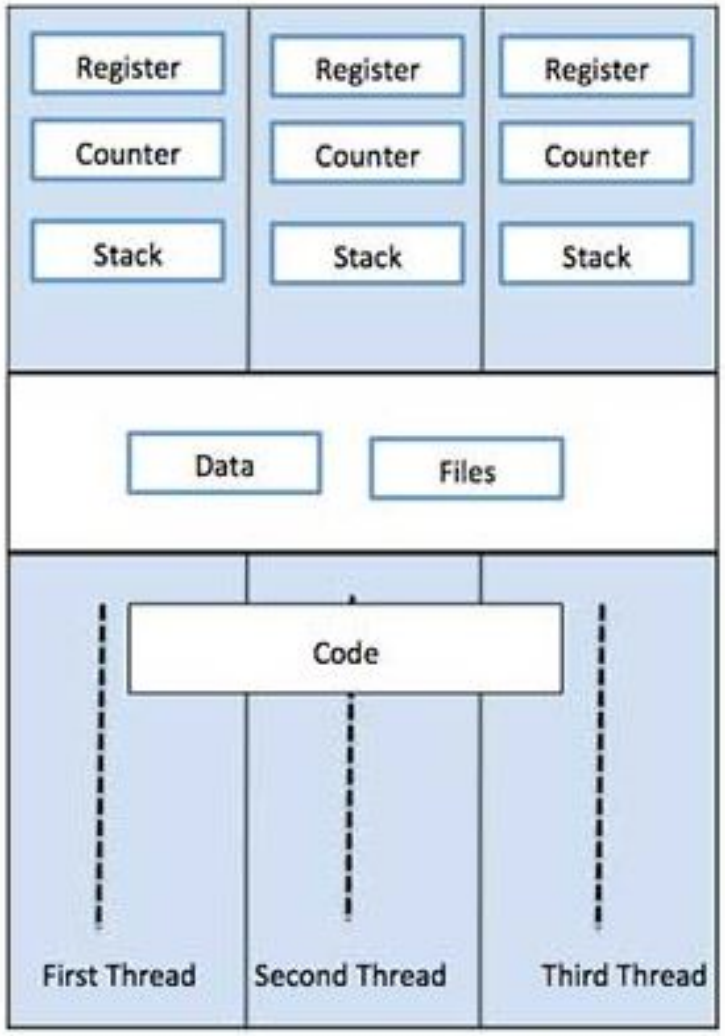
- Thread ID
- CPU context (PC, SP, register set, etc.)
- Stack
- Priority
- errno

MULTI-THREADING

Threads shares:

- Code and data
- Open files (through the PPFDT)
- Current working directory
- User and group IDs
- Signal setups and handlers
- PCB

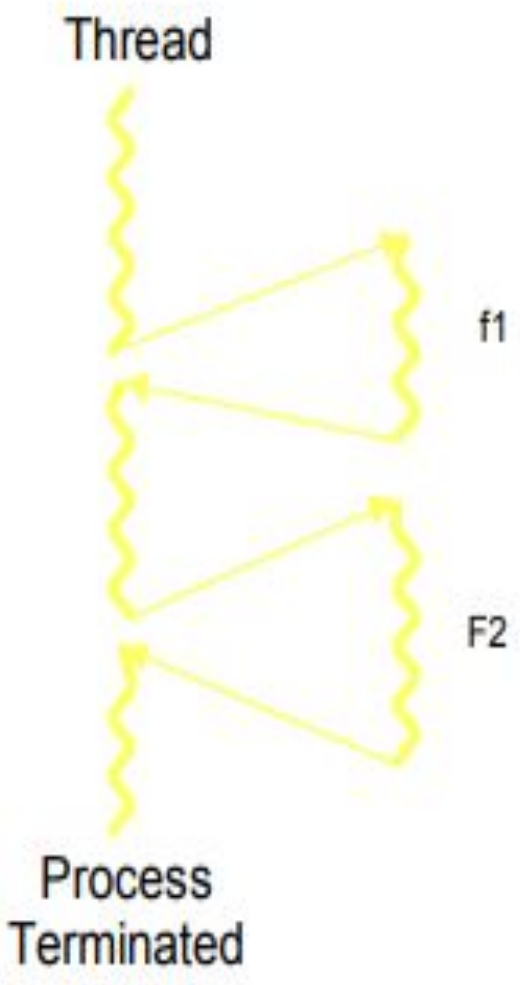




https://www.tutorialspoint.com/operating_system/os_multi_threading.htm

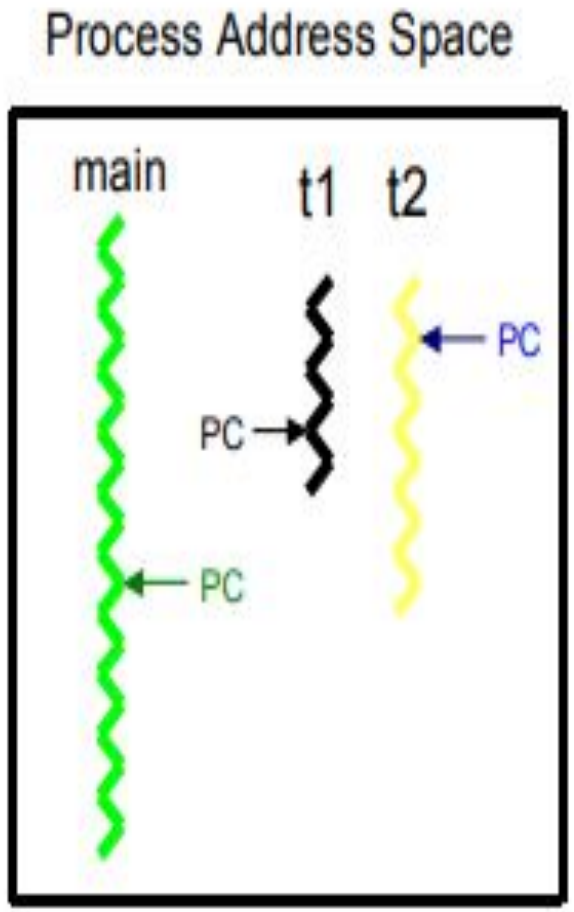
Single-Threaded

```
main()
{

     f1(

     f2(…);
     …
}
f1(…)
{  …  }
f2(…)
{  …  }
```



Multi-Threaded

```
main()
{
      ...
      thread(t1,f1);
      ...
      thread(t2,f2);
      ...
}
f1(...)
{ ... }
f2(...)
{ ... }
```



MULTI-THREADING (ADVANTAGES)

Responsiveness—Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user.

Resource sharing—By default, threads share the memory and the resources of the process to which they belong. Code sharing allows an application to have several different threads of activity all within the same address space.

MULTI-THREADING (ADVANTAGES)

Economy—Allocating memory and resources for process creation is costly. Alternatively, because threads share resources of the process to which they belong, it is more economical to create and context switch threads.

Utilization of multiprocessor architectures—The benefits of multithreading can be greatly increased in a multiprocessor environment, where each thread may be running in parallel on a different processor. A single-threaded process can only run on one CPU no matter how many are available. Multithreading on multi-CPU machines increases concurrency.

MULTI-THREADING (DISADVANTAGES)

Resource sharing—Whereas resource sharing is one of the major advantages of threads, it is also a disadvantage because proper synchronization is needed between threads for accessing the shared resources (e.g., data and files).

Difficult programming model—It is difficult to write, debug, and maintain multithreaded programs for an average user. This is particularly true when it comes to writing code for synchronized access to shared resources.

TYPES OF PARALLELISM

In general, there are two types of parallelism: data parallelism and task parallelism.

Data parallelism—focuses on distributing subsets of the same data across multiple computing cores and performing the same operation on each core.

Task parallelism—involves distributing not data but tasks (threads) across multiple computing cores. Each thread is performing a unique operation. Different threads may be operating on the same data, or they may be operating on different data.

POSIX THREADS (THE PTHREAD LIBRARY)

- Pthreads refers to the POSIX standard defining an API for thread creation, scheduling, and synchronization.
- This is a specification for thread behavior not an implementation.
- OS designers may implement the specification in any way they wish.
- Generally, libraries implementing the Pthreads specification are restricted to UNIX-based systems such as Solaris 2.

CREATING A THREAD

- You can create a threads by using the pthread_create() call.
- Here is the syntax of this call.

int pthread_create(pthread_t *threadp, const pthread_attr_t *attr, void* (*routine)(void *), arg *arg);

- Here, ‘threadp’ contains thread ID (TID) of the thread created by the call,
- ‘attr’ is used to modify the thread attributes (stack size, stack address, detached, joinable, priority, etc.),
- ‘routine’ is the thread function, and
- ‘arg’ is any argument we want to pass to the thread function.
- The argument does not have to be a simple native type; it can be a ‘struct’ of whatever we want to pass in

PTHREAD CREATE CALL FAILS

- The pthread_create() call fails and returns the corresponding value if any of the following conditions is detected:
- **EAGAIN**—The system-imposed limit on the total number of threads in a process has been exceeded or some system resource has been exceeded (for example, too many LWPs were created).
- **EINVAL—**The value specified by ‘attr’ is invalid.
- **ENOMEM**—Not enough memory was available to create the new thread.
- You can do error handling by including the file <errno.h> and incorporating proper error handling code in your programs.

JOINING A THREAD

- You can have a thread wait for another thread within the same process by using the pthread_join() call.
- Here is the syntax of this call.

int pthread_join(pthread_t aThread, void **statusp);

- Here, ‘aThread’ is the thread ID of the thread to wait for and
- ‘statusp’ gets the return value of pthread_exit() call made in the process for whom wait is being done.
- A thread can only wait for a joinable thread in the same process address space
- A thread cannot wait for a detached thread.
- Multiple threads can join with a thread but only one returns successfully; others return with an error that no thread could be found with the given TID.

TERMINATING A THREAD

- You can terminate a thread explicitly by either returning from the thread function or by using the pthread_exit() call.
- Here is the syntax of the pthread_exit() call.

void pthread_exit(void *valuep);

- Here, ‘valuep’ is a pointer to the value to be returned to the thread which is waiting for this thread to terminate (i.e., the thread which has executed pthread_join() for this thread).
- A thread also terminates when the main thread in the process terminates.
- When a thread terminates with the exit() system call, it terminates the whole process because the purpose of the exit() system call is to terminate a process and not a thread.