This code is a Linux kernel module that lists all tasks in the system using Depth-First Search (DFS) traversal of the task hierarchy.
It defines two functions: tasks_lister_dfs_init and tasks_lister_dfs_exit, which are used to initialize and exit the module, respectively.

The dfs function is called from tasks_lister_dfs_init with the init_task as the argument. This is the first task in the system and is the parent of all other tasks. The dfs function then iterates over all the child tasks of init_task using the list_for_each macro, and for each child task,
it prints its process ID, process name, and state using the printk function. It then recursively calls itself
with the child task as the argument to traverse the entire task hierarchy.

The module uses the printk function to output the list of tasks to the kernel log buffer. When the module is loaded, the tasks_lister_dfs_init function is called, and when the module is unloaded,
the tasks_lister_dfs_exit function is called.

Makefile script for building a Linux kernel module named dfs.o.

The obj-m += dfs.o line specifies that the dfs.o file is the output object file that will be generated by the Makefile.

The KERNELDIR variable is set to the path of the kernel build directory, which is usually /lib/modules/$(shell uname -r)/build. The $(shell uname -r) command is used to get the current kernel release version.

The PWD variable is set to the current working directory.

The all target is used to build the module by invoking the make command with the -C option to change to the kernel build directory, and the M option to specify the current directory as the location of the module source code.

The clean target is used to clean up the build artifacts by invoking the make command with the same options as the all target, but with the clean parameter.

To use this Makefile, navigate to the directory containing the Makefile and run make to build the module or make clean to clean up the build artifacts.

This is a kernel module written in C language that lists the tasks using DFS (Depth First Search) traversal. The module starts from the init task, which is the ancestor of all tasks in the Linux kernel, and traverses the process tree using DFS. For each task visited during the traversal, the module prints its process ID, process name, and process state using the printk() function.

The DFS traversal is implemented using a recursive function named dfs(). The function takes a pointer to a task_struct structure, which represents a task in the kernel, as its parameter. The function first iterates over the children of the task using the list_for_each() macro and a pointer to the children list of the task. For each child task, the function prints its process ID, process name, and process state and then calls itself recursively with the child task as its argument.

The module has two functions - tasks_lister_dfs_init() and tasks_lister_dfs_exit() - that are used to initialize and remove the module, respectively. The module also contains some metadata such as the license, description, and author information.

Overall, this module can be used to traverse the process tree in the Linux kernel and obtain information about each task in a depth-first manner. This can be useful for debugging purposes or for obtaining information about the system's processes.

tasks_struct

task_struct is a data structure in the Linux kernel that represents a task, also known as a process. It contains all the information about the task, such as its process ID (PID), state, priority, memory usage, file descriptors, signal handlers, and more.

The task_struct structure is defined in the sched.h header file and is used extensively throughout the kernel for managing tasks. Every running process in the system has a corresponding task_struct structure that is dynamically allocated when the process is created.

The task_struct structure is a large and complex data structure, consisting of many fields that are used by various parts of the kernel. It is also heavily accessed and modified by the scheduler, which is responsible for deciding which tasks should run and for how long.

Overall, task_struct is a fundamental data structure in the Linux kernel that is essential for managing and scheduling tasks in the system.

list_head

list_head is a data structure in the Linux kernel that represents a doubly linked list node. It contains two pointers, next and prev, that point to the next and previous nodes in the list, respectively.

list_head is used extensively throughout the kernel for implementing linked lists of various types of data structures, including tasks (task_struct), files (file), sockets (socket), and more.

One of the main advantages of using list_head is that it provides a uniform way to manage linked lists that is both efficient and easy to use. The list_for_each() macro, for example, can be used to iterate over a list of nodes, while the list_add() and list_del() functions can be used to insert and remove nodes from a list, respectively.

In addition to list_head, there are other data structures in the kernel that are based on linked lists, such as hlist_head (for hash lists) and rcu_head (for read-copy-update synchronization).

list_for_each()

list_for_each() is a macro in the Linux kernel that is used to iterate over a linked list. It takes two arguments: a pointer to a list_head structure that represents the head of the list, and a pointer to a variable that will hold the current element in the list.

In the code snippet you provided, list_for_each() is used to iterate over the list of child tasks of a given task. The first argument is a pointer to the children field of the task_struct structure, which is a list_head structure that represents the head of the list of child tasks. The second argument is a pointer to a variable called list, which will hold the current element in the list.

During each iteration, the list_entry() macro is used to extract the current child task from the list variable. The list_entry() macro takes three arguments: a pointer to the current element in the list, the type of the data structure that contains the list_head structure (in this case, struct task_struct), and the name of the list_head structure within the data structure (in this case, sibling).

Overall, this code iterates over the list of child tasks of a given task, and for each child task, it extracts its task_struct structure using list_entry() and performs some operation on it (in this case, printing its process ID, name, and state).

printk()

printk() is a function in the Linux kernel that is used to print messages to the kernel log buffer. It takes a format string and a variable number of arguments, similar to the printf() function in standard C.

In the code snippet you provided, printk() is used to print information about each child task of a given task. The format string is "pid: %d | pname: %s | state: %ld\n", which specifies the format of the output message. The %d format specifier is used to print the process ID of the child task, %s is used to print its name, and %ld is used to print its state.

printk(KERN_INFO "pid: %d | pname: %s | state: %ld\n", task_next->pid, task_next->comm, task_next->state);

The task_next pointer is a pointer to the current child task in the list, which was obtained using list_entry() in the previous line of code. The -> operator is used to access the fields of the task_struct structure, such as pid, comm (which stores the process name), and state.

Overall, this code prints a message to the kernel log buffer for each child task of a given task, containing information about the process ID, name, and state of the child task.

KERN Info

KERN_INFO is a logging level in the Linux kernel that is used to indicate the severity of a log message. It is one of several logging levels defined in the kernel, including KERN_DEBUG, KERN_WARNING, KERN_ERR, and others.

The KERN_INFO level is typically used for informational messages that do not indicate an error or warning condition. Messages logged at this level are intended to provide useful information to the system administrator or developer, such as the progress of a system operation, the status of a device driver, or other system events.

When a log message is written to the kernel log buffer using printk() or a similar function, the logging level is included as a prefix to the message, along with other information such as the process ID, thread ID, and timestamp.

Overall, KERN_INFO is a logging level in the Linux kernel that is used to indicate an informational message that is not an error or warning condition, but rather provides useful information to the system administrator or developer.

Output:

The output you provided is a log message generated by the Linux kernel, indicating the status of various processes on the system at a particular time. Each line represents a different process, identified by its process ID (PID) and process name (PNAME), along with its current state.

The log message appears to have been generated at 1628.350157 seconds since the system was started, and shows the status of several system-level processes, such as systemd, systemd-journal, and systemd-udevd, as well as various user-level processes like apache2, gdm3, and gnome-shell.

The state of each process is indicated by a number, where 0 indicates that the process is running, and any other number indicates that it is waiting for a particular event or resource. The specific meanings of each state may vary depending on the process, but in general, a process that is not in state 0 is not actively executing code at the moment.

In the log, the "state" refers to the current state of the process. It is a numeric value that indicates the state of the process at the time the log entry was created. The states are defined as follows:

0: Process is running or runnable.
1: Process is sleeping.
2: Process is currently blocked.
4: Process has been stopped.
8: Process has been traced or debugged.
The state of a process can change over time, depending on the actions it is performing or the events that occur in the system. The state can be useful for diagnosing problems or understanding the behavior of a system, as it gives an indication of what the process is currently doing or what state it is in.

In the log message you provided, "state: 1026" refers to the current state of the process with a process ID (PID) of 91 and a process name of "blkcg_punt_bio".

In Linux, the process state is a code that indicates the current condition of a process. The state of a process can be one of several possible values, each corresponding to a different state of the process.

The state code 1026 in this log message means that the process is currently in the "TASK_RUNNING" state. This state means that the process is currently either running or waiting to run on a CPU.

The state code is useful for debugging purposes, as it can help identify what the process is currently doing and whether it is experiencing any issues or delays.

STATE

This is a section of system log output showing the process ID (PID), process name (PNAME), and state of several processes running on the system.

The PID is a unique identifier assigned to each running process, the PNAME is the name of the process, and the state is the current execution state of the process.

For example, “pid: 76 | pname: edac-poller | state: 1026” indicates that the process with PID 76 is named “edac-poller” and its state is 1026. The meaning of the specific state codes can vary depending on the system and context. In this case, a state of 1026 typically indicates that the process is blocked waiting for I/O to complete. The specific state values can vary slightly depending on the operating system, but on many Unix-like systems, the state values for a process can be found by examining the /proc/<pid>/status file. In this case, the state value of 1026 indicates that the process is in the “D” state, which stands for “Uninterruptible Sleep”. This means that the process is waiting for some external event, such as disk I/O or network I/O, and cannot be interrupted by signals or other events until the operation completes.

This is a log message from the Linux kernel's message buffer (dmesg). Each line shows the process ID (pid) and process name (pname) of a running system service along with its current state. The state 1 means that the process is currently running.

In summary, this log message is displaying a list of running system services on the Linux system.

In this context, a state of 1 indicates that the process is currently running, i.e., it is currently executing and has not been suspended or terminated. The state is represented by a number that indicates the current process state. In Linux, the process states are represented by a series of numbers and codes, with each one indicating a different state. Some common states include 0 (process is running and is in the "task running" state), 1 (process is running and is in the "interruptible sleep" state), and 2 (process is running and is in the "uninterruptible sleep" state).