Process Management and Scheduling

There is a two different type of scheduled tasks methods that we need to consider.

Multitasking and Monotasking

Modern Operation systems have a multitasking algorithms to run multiple tasks on a CPU.

This gives a impression of multitasking.

Other than that planes likely vehicles and some critical devices like medical devices need to real time task run on a CPU.

Process Priorities

Processes can be split real-time processes and non-real-time processes

Hard real-time processes are subject to strict time limits during which certain tasks must be completed.

Linux does not support hard real-time processing, at least not in the vanilla kernel. There are, however, modified versions such as **RTLinux**, **Xenomai**, or **RATI** that offer this feature.

Soft real-time processes are a softer form of hard real-time processes. Although quick results are still required, it is not the end of the world if they are a little late in arriving.

Most processes are normal processes that have no specific time constraints but can still be classified
as more important or less important by assigning priorities to them.

Most processes are normal processes that have no specific time constraints but can still be classified
as more important or less important by assigning priorities to them.

Process Life Cycle

A process has three states

Running — The process is executing at the moment.

Waiting — The process is able to run but is not allowed to because the CPU is allocated to
another process. The scheduler can select the process, if it wants to, at the next task switch.

Sleeping — The process is sleeping and cannot run because it is waiting for an external event.
The scheduler cannot select the process at the next task switch.

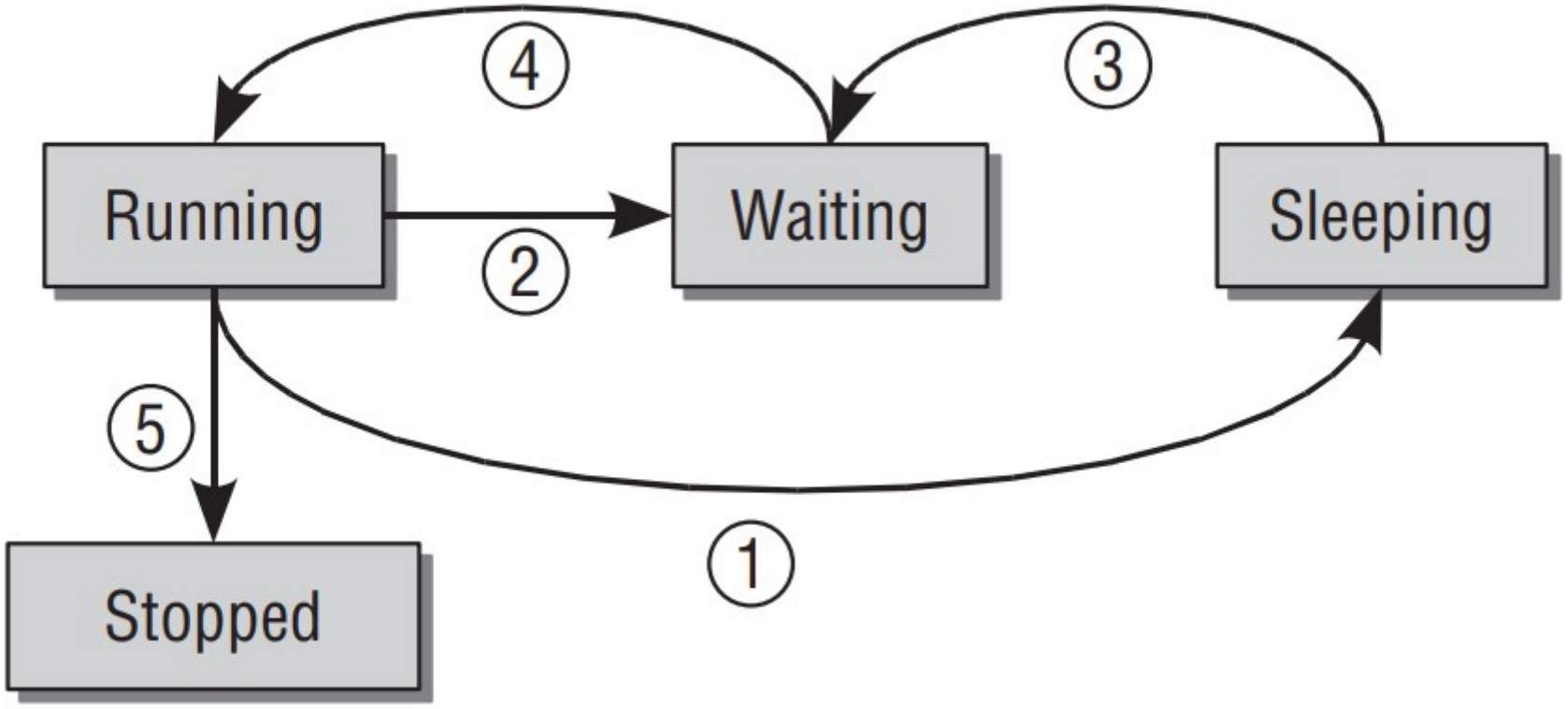


Figure 2-2: Transitions between process states.

If a CPU is allocated by different processes a task waits until the scheduler grants it CPU time. Once this happens, its state changes to “running”

When the scheduler dedices to withdraw CPU resources from the process, the process state changes from “running” to “waiting”

If the process has no wait for an event, its state changes from “running” to “sleeping”

Once program execution termianates, the process state changes from “running” to “stopped”

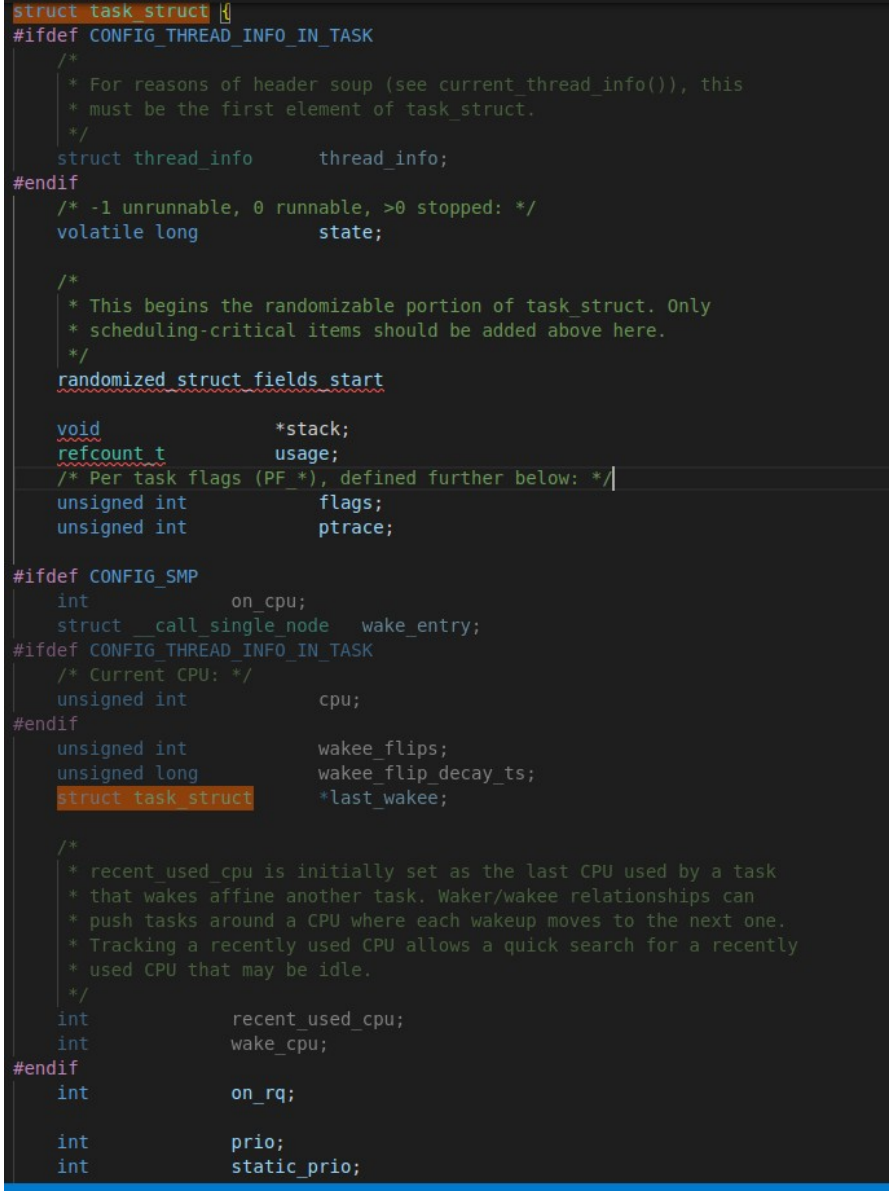
A special process state not listed in the diagram is “zombie” state. Such processes are defunct but are somehow still alive.

Preemtive Multitasking

The structure of Linux process management requires two further process state option- user mode and kernel mode. These reflect the fact that all modern CPUs have (at least) two different execution modes, one of which has unlimited rights while the other is subject to various restriction – for exmple access ro certain memory areas can be prohibited

Process Representation

All algorithms of the linux kernel concerned with process and programs are build around a data structure named ***task_struct*** *defined in* ***include/sched.h***



```
struct task_struct {
#ifdef CONFIG_THREAD_INFO_IN_TASK
    /*
     * For reasons of header soup (see current_thread_info()), this
     * must be the first element of task_struct.
     */
    struct thread_info      thread_info;
#endif
    /* -1 unrunnable, 0 runnable, >0 stopped: */
    volatile long           state;

    /*
     * This begins the randomizable portion of task_struct. Only
     * scheduling-critical items should be added above here.
     */
    randomized_struct_fields_start

    void                *stack;
    refcount_t          usage;
    /* Per task flags (PF_*), defined further below: */
    unsigned int            flags;
    unsigned int            ptrace;

#ifdef CONFIG_SMP
    int             on_cpu;
    struct __call_single_node   wake_entry;
#ifdef CONFIG_THREAD_INFO_IN_TASK
    /* Current CPU: */
    unsigned int            cpu;
#endif
    unsigned int            wakee_flips;
    unsigned long           wakee_flip_decay_ts;
    struct task_struct      *last_wakee;

    /*
     * recent_used_cpu is initially set as the last CPU used by a task
     * that wakes affine another task. Waker/wakee relationships can
     * push tasks around a CPU where each wakeup moves to the next one.
     * Tracking a recently used CPU allows a quick search for a recently
     * used CPU that may be idle.
     */
    int             recent_used_cpu;
    int             wake_cpu;
#endif
    int             on_rq;

    int             prio;
    int             static_prio;
```

All algorithms of the linux kernel concerned with process and programs are build around a data structure named ***task_struct*** *defined in* ***include/sched.h***

Admittedly, it is difficult to digest the amount of information in this structure. However, the structure
contents can be broken down into sections, each of which represents a specific aspect of the process:

We wont covet all of the sections of task_struct in here. We can broke down into section.

1 - State and executing information such as pending signals, binary format used (and any emulation information for binary formats of other systems), process identification number (pid) pointers to parents and other related processes, priorities, and time information on program execution

2 - Information on allocated virtual memory

3 - Process credentials such as user and group ID, capabilities, 2 and so on. System calls can be used
to query (or modify) these data; I deal with these in greater detail when describing the specific subsystems.

4 - Files used: Not only the binary file with the program code but also filesystem information on all
files handled by the process must be saved.

5 - Thread information, which records the CPU-specific runtime data of the process (the remaining
fields in the structure are not dependent on the hardware used).

6 - Information on interprocess communication required when working with other applications.

7 - Signal handlers used by the process to respond to incoming signals.

State member hold the current state of a task

There is a few states:
TASK_RUNNING
TASK_INTERRUPTIBLE
TASK_UNINTERUPTIBLE
TASK_STOPPED
TASK_TRACED

As exit_state in task_struct
EXIT_ZOMBIE
EXIT_DEAD

Linux provides the resource limit (rlimit) mechanism to impose certain system resource usage limits on processes

<resource.h>

```
struct rlimit {
    unsigned long   rlim_cur;
    unsigned long   rlim_max;
}
```

Monitoring the process limits

In Linux

cat /*proc*/self/limits

```
cin@cin:~/syzkaller$ cat /proc/self/limits
Limit                     Soft Limit           Hard Limit           Units
Max cpu time              unlimited            unlimited            seconds
Max file size             unlimited            unlimited            bytes
Max data size             unlimited            unlimited            bytes
Max stack size            8388608              unlimited            bytes
Max core file size        0                    unlimited            bytes
Max resident set          unlimited            unlimited            bytes
Max processes             95554                95554                processes
Max open files            1024                 1048576              files
Max locked memory         3141873664           3141873664           bytes
Max address space         unlimited            unlimited            bytes
Max file locks            unlimited            unlimited            locks
Max pending signals       95554                95554                signals
Max msgqueue size         819200               819200               bytes
Max nice priority         0                    0
Max realtime priority     0                    0
Max realtime timeout      unlimited            unlimited            us
```

`RLIMIT_CPU`	Maximum CPU time in milliseconds.
`RLIMIT_FSIZE`	Maximum file size allowed.
`RLIMIT_DATA`	Maximum size of the data segment.
`RLIMIT_STACK`	Maximum size of the (user mode) stack.
`RLIMIT_CORE`	Maximum size for `core` dump files.
`RLIMIT_RSS`	Maximum size of the *resident size set*; in other words, the maximum number of page frames that a process uses. Not used at the moment.
`RLIMIT_NPROC`	Maximum number of processes that the user associated with the real UID of a process may own.
`RLIMIT_NOFILE`	Maximum number of open files.
`RLIMIT_MEMLOCK`	Maximum number of non-swappable pages.
`RLIMIT_AS`	Maximum size of virtual address space that may be occupied by a process.
`RLIMIT_LOCKS`	Maximum number of file locks.
`RLIMIT_SIGPENDING`	Maximum number of pending signals.
`RLIMIT_MSGQUEUE`	Maximum number of message queues.
`RLIMIT_NICE`	Maximum nice level for non-real-time processes.
`RLIMIT_RTPRIO`	Maximum real-time priority.

Process Types

New processes are generated using the fork and exec system calls

fork

generates an identical copy of the current process; this copy is known as a child process. All resources of the original process are copied in a suitable way so that after the system call there are two independent instances of the original process.

exec

replaces a running process with another application loaded from an executable binary file. In other words, a new program is loaded. Because exec does not create a new process, an old program must first be duplicated using fork, and then exec must be called to generate an additional application on the system.

Linux also provided ***clone*** system call in addition to the two calls above that are avaliable in all UNIX flavors.

In principle, clone works in the same way as fork, but the new process is not independent of its parent process and can share some resources