Time

System calls for Time

`stime` allows the **superuser** to set a global kernel variable to a value that gives the current time.

```
#include <time.h>
stime(pvalue);
```

where `pvalue` **points to a long integer** that gives the time measured in seconds from midnight before (00:00:00) January 1, 1970, GMT. The clock interrupt handler increments the kernel variable once a second.

Time retrieves the time as set by `stime()`:

```
#include <time.h>
time(tloc);
```

where `tloc` points to a location in the user process for return value.

`times` retrieves the **cumulative times that the calling process spent executing in kernel mode and user mode** and the cumulative times that all zombie children had executed in user mode and kernel mode. `times` **returns the elapsed time** "from an arbitrary point in the past", **usually the time of system boot**.

```
#include <sys/times.h>
times(tbuffer);
struct tms *tbuffer;


struct tms{
/* time_t is the data structure for time */
time_t tms_utime; /* user time of process */
time_t tms_stime; /* kernel time of process */
time_t tms_cutime; /* user time for children */
time_t tms_cstime; /* kernel time for children */
}
```

The child times **do not include time spent in the `fork()` and `exit()`**, and all **times can distorted by times spent handling interrupts or doint context switches**.

Alarm system call set the alarm clock of a process. User processes can schedule **alarm signals** using the alarm system calls:

```
#include <unistd.h>
int alarm(int sec);
```

generates a SIGALRM signal for the process after the number of real-time seconds specified by `sec` (seconds).

If there is a previous `alarm()` request with time remaining, `alarm()` returns non zero that is the number of seconds until the previous request would have generated a SIGALRM signal. Otherwise, `alarm()` returns 0.

Example:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/signal.h>

main(argc, argv)
      int argc;
      char *argv[];
{
      extern unsigned alarm();
      extern wakeup();
      struct stat statbuf;
      time_t axtime;

      if (argc != 2)
      {
            printf("only 1 arg\n");
            exit();
      }

      axtime = (time_t) 0;
      for (;;)
      {
            /* find out file access time */
```

```
		if (stat(argv[1], &statbuf) == -1)
		{
			printf("file %s not there\n", argv[1]);
			exit();
		}
		if (axtime != statbuf.st_atime)
		{
			printf("file %s accessed\n", argv[1]);
			axtime = statbuf.st_atime;
		}
		signal(SIGALRM, wakeup);	/* reset for alarm */
		alarm(60);
		pause();		/* sleep until signal */
	}
}

wakeup()
{
}
```

Clock

The functions of the **clock handler** are to:

- restart the clock
- schedule invocation of internal kernel functions based on internal timers
- provide execution profiling capability for the kernel and for user process
- gather system and process accounting statistics
- keep track of time
- send alarm signals to processes on request
- periodically wake up the swapper process
- control process scheduling

Algorithm for clock handler

```
algorithm clock
input: none
output: none
{
        restart clock;
```

```
        if(callout table not empty){
                adjust callout times;
                schedule callout function if time elapsed;
        }
        if(kernel profiling on)
                note program counter at time of interrupt;
        if(user profiling on)
                note program counter at time of interrupt;
        gather system statistics;
        gather statistics per process;
        adjust measure of process CPU utlization;
        if(1 second or more since last here and interrupt not in critical region of
code){
                for(all processes in the system){
                        adjust alarm time if active;
                        adjust measure of CPU utlization;
                        if(process to execute in user mode)
                                adjust process priority;
                }
                wakeup swapper process is necessary;
        }
}
```

Restarting the clock

- When the clock interrupts the system, most machines require that the clock be reprimed (reprepare) by software instructions so that it will interrupt the processor again after a suitable interval.
- Such instructions are hardware dependent.

Internal System Timeouts

- Some kernel operations, particularly device drivers and network protocols, require the invocation of kernel functions on a real time basis.
- Example: Putting kernel into raw mode so that the kernel satisfies user read requests at fixed intervals.
- The kernel stores the necessary information in the **callout table**, which consists of the **function to be invoked** when the time expires, a **parameter for the function**, and the **time** in clock ticks **until the function should be called**. User has no direct control over this.

Function	Time to Fire
a()	-2
b()	3
c()	10

Before

Function	Time to Fire
a()	-2
b()	3
f()	2
c()	8

After

Figure 8.10. Callout Table and New Entry for f

ADDING A NEW ENTRY

- The kernel finds the correct (timed) position to insert the new entry and **appropiately adjusts the time field of the entry immediately after the new entry**.
- New entry: f with time 5s. Now `c()` will execute in 13s (10 + 3)
- It finds that it needs to execute after `b()` to satisfy the 5s time (3 < 5), its time will be now

  $$5 - 3 = 2$$

  and the entry immediately after

  $$10 - 2 = 8$$

  so that `c()` will still fire in 13s.
- So the time for the new entry is time - time for previous entry.

DATA STRUCTURES FOR THIS

- Linked list can be used
- Or the kernel can read just the position of the entries when changing the table (not too expensive if the kernel does not uses the callout table too much).

Procedure

- At every clock interrupt, the clock handler checks if there are any entries in the callout table
- If there are any, it decrements the time field of the first entry.
- Because of the way, the kernel keeps the entries, decrementing the first entry effectively decrements the time field for all entries in the table.

- If the time field of the first entry < 0, the specified function should be invoked.
- The clock handler does not directly invoke the function since it will block later clock interrupts, instead it schedules the function by generating a "software interrupt"
- Because, s/w interrupts are at lower priority than other interrupts, they are blocked until kernel handles all other interrupts.
- The entry is then removed.

Profiling

- Gives a measure of how much time the system is executing in user mode vs. kernel mode, and how much time it spends executing individual routines in the kernel.
- The kernel profile driver monitors the relative performance of kernel modules by sampling system activity at the time of the clock interrupt.
- The profile driver has a list of kernel address to sample, usually addresses of kernel function.
- If the kernel profiling is enabled, **the clock handler invokes the interrupt handler of the profile driver**, which determines the whether the processor mode at the time of interrupt was user or kernel.
- If the mode was user, the **profiler increments a count for user execution**, but of the mode was kernel, it **increments an internal counter corresponding to the program counter**.
- User processes can read the profile driver to obtain kernel counts and do statistical measurements.

Clock handler invokes interrupt handler of profile driver $\rightarrow$ determines processor mode (user or kernel) $\rightarrow$ increments a counter

Profile execution of processes at user-level with `profil()`

```
profil(buff, bufsize, offset, scale);
```

where `buff` is the address of an array in user space and `bufsize` is its size
`offset` is the virtual address of user subroutine
`scale` is a factor that maps user virtual addresses into array.

In the user mode, the clock handler examines the user program counter at the time of interrupt, compares it to offset, and increments a location in `buff` whose address is a function of `bufsize` and `scale`.

compare `offset` and `pc` $\rightarrow$ increments a location in `buff`

Algorithm	Address	Count
bread	100	5
breada	150	0
bwrite	200	0
brelse	300	2
getblk	400	1
user	—	2

Figure 8.11. Sample Addresses of Kernel Algorithms

Accounting and Statistics

- Every process has 2 fields in its u area to **keep a record of elapsed kernel** and **user time**.
- When handling clock interrupts, the kernel updates the appropiate field for the executing process, depending on whether it was executing in the user mode or the kernel mode.
- Parent processes gather statistics for their child processes in the `wait()`.
- Every process has one field in its u area for **the kernel to log its memory usage**.
- When the clock interrupts, the kernel calculates the total memory used by a process as a function of its private memory regions and its proportional usage of the shared memory regions (since a process may use a part of the shared memory region).
- Example: shared region of size $= 50K$ with 4 other processes and text and data regions of size $25K$ and $40K$ respectively. So memory usage equals,

$$memory = 50K/5 + 25K + 40K = 75K$$