NAME

getrusage – get information about resource utilization

SYNOPSIS

#include <sys/resource.h>

int getrusage(int *who***, struct rusage** **rusage***);**

DESCRIPTION

getrusage( ) returns information about the resources utilized by the current process, or all its terminated child processes. The interpretation for some values reported, such as **ru_idrss**, are dependent on the clock tick interval. This interval is an implementation dependent value.

The **who** parameter is one of **RUSAGE_SELF** or **RUSAGE_CHILDREN**. The buffer to which *rusage* points will be filled in with a structure with the following members:

```
struct timeval ru_utime;  /* user time used */
struct timeval ru_stime;  /* system time used */
int     ru_maxrss;        /* maximum resident set size */
int     ru_idrss;         /* integral resident set size */
int     ru_minflt;        /* page faults not requiring physical I/O */
int     ru_majflt;        /* page faults requiring physical I/O */
int     ru_nswap;         /* swaps */
int     ru_inblock;       /* block input operations */
int     ru_oublock;       /* block output operations */
int     ru_msgsnd;        /* messages sent */
int     ru_msgrcv;        /* messages received */
int     ru_nsignals;      /* signals received */
int     ru_nvcsw;         /* voluntary context switches */
int     ru_nivcsw;        /* involuntary context switches */
```

The fields are interpreted as follows:

ru_utime The total amount of time spent executing in user mode. Time is given in seconds and microseconds.

ru_stime The total amount of time spent executing in system mode. Time is given in seconds and microseconds.

ru_maxrss The maximum resident set size. Size is given in pages (the size of a page, in bytes, is given by the **getpagesize**(3C) function). See the **NOTES** section of this page.

ru_idrss An “integral” value indicating the amount of memory in use by a process while the process is running. This value is the sum of the resident set sizes of the process running when a clock tick occurs. The value is given in pages times clock ticks. It does not take sharing into account. See the **NOTES** section of this page.

ru_minflt The number of page faults serviced which did not require any physical I/O activity. See the **NOTES** section of this page.

ru_majflt The number of page faults serviced which required physical I/O activity. This could include page ahead operations by the kernel. See the **NOTES** section of this page.

ru_nswap The number of times a process was swapped out of main memory.

ru_inblock The number of times the file system had to perform input in servicing a **read**(2) request.

ru_oublock The number of times the file system had to perform output in servicing a **write**(2) request.

ru_msgsnd The number of messages sent over sockets.

ru_msgrcv The number of messages received from sockets.

ru_nsignals The number of signals delivered.

ru_nvcsw The number of times a context switch resulted due to a process voluntarily giving up the processor before its time slice was completed (usually to await availability of a resource).

ru_nivcsw The number of times a context switch resulted due to a higher priority process becoming runnable or because the current process exceeded its time slice.

RETURN VALUES

If successful, the value of the appropriate structure is filled in, and **0** is returned. If the call fails, −**1** is returned.

ERRORS

getrusage( ) will fail if:

EFAULT The address specified by the *rusage* argument is not in a valid portion of the process's address space.

EINVAL The **who** parameter is not a valid value.

SEE ALSO

sar(1M), **read**(2), **times**(2), **wait**(2), **write**(2), **getpagesize**(3C), **gettimeofday**(3C)

NOTES

Only the *timeval* fields of **struct rusage** are supported in this implementation.

The numbers **ru_inblock** and **ru_oublock** account only for real I/O, and are approximate measures at best. Data supplied by the cache mechanism is charged only to the first process to read and the last process to write the data.

The way resident set size is calculated is an approximation, and could misrepresent the true resident set size.

Page faults can be generated from a variety of sources and for a variety of reasons. The customary cause for a page fault is a direct reference by the program to a page which is not in memory. Now, however, the kernel can generate page faults on behalf of the user, for example, servicing **read**(2) and **write**(2) functions. Also, a page fault can be caused by an absent hardware translation to a page, even though the page is in physical memory.

In addition to hardware detected page faults, the kernel may cause pseudo page faults in order to perform some housekeeping. For example, the kernel may generate page faults, even if the pages exist in physical memory, in order to lock down pages involved in a raw I/O request.

By definition, major page faults require physical I/O, while minor page faults do not require physical I/O. For example, reclaiming the page from the free list would avoid I/O and generate a minor page fault. More commonly, minor page faults occur during process startup as references to pages which are already in memory. For example, if an address space faults on some “hot” executable or shared library, this results in a minor page fault for the address space. Also, any one doing a **read**(2) or **write**(2) to something that is in the page cache will get a minor page fault(s) as well.

There is no way to obtain information about a child process which has not yet terminated.