

**NAME**

getaddrinfo\_a, gai\_suspend, gai\_error, gai\_cancel - asynchronous network address and service translation

**SYNOPSIS**

```
#define _GNU_SOURCE /* See feature_test_macros(7) */
#include <netdb.h>

int getaddrinfo_a(int mode, struct gaicb *list[],
                  int nitems, struct sigevent *sevp);

int gai_suspend(const struct gaicb * const list[], int nitems,
                const struct timespec *timeout);

int gai_error(struct gaicb *req);

int gai_cancel(struct gaicb *req);
```

Link with *-lanl*.

**DESCRIPTION**

The `getaddrinfo_a()` function performs the same task as `getaddrinfo(3)`, but allows multiple name look-ups to be performed asynchronously, with optional notification on completion of look-up operations.

The *mode* argument has one of the following values:

**GAI\_WAIT**

Perform the look-ups synchronously. The call blocks until the look-ups have completed.

**GAI\_NOWAIT**

Perform the look-ups asynchronously. The call returns immediately, and the requests are resolved in the background. See the discussion of the *sevp* argument below.

The array *list* specifies the look-up requests to process. The *nitems* argument specifies the number of elements in *list*. The requested look-up operations are started in parallel. NULL elements in *list* are ignored. Each request is described by a *gaicb* structure, defined as follows:

```
struct gaicb {
    const char *ar_name;
    const char *ar_service;
    const struct addrinfo *ar_request;
    struct addrinfo *ar_result;
};
```

The elements of this structure correspond to the arguments of `getaddrinfo(3)`. Thus, *ar\_name* corresponds to the *node* argument and *ar\_service* to the *service* argument, identifying an Internet host and a service. The *ar\_request* element corresponds to the *hints* argument, specifying the criteria for selecting the returned socket address structures. Finally, *ar\_result* corresponds to the *res* argument; you do not need to initialize this element, it will be automatically set when the request is resolved. The *addrinfo* structure referenced by the last two elements is described in `getaddrinfo(3)`.

When *mode* is specified as **GAI\_NOWAIT**, notifications about resolved requests can be obtained by employing the *sigevent* structure pointed to by the *sevp* argument. For the definition and general details of this structure, see `sigevent(7)`. The *sevp->sigev\_notify* field can have the following values:

**SIGEV\_NONE**

Don't provide any notification.

**SIGEV\_SIGNAL**

When a look-up completes, generate the signal *sigev\_signo* for the process. See [sigevent\(7\)](#) for general details. The *si\_code* field of the *siginfo\_t* structure will be set to **SI\_ASYNCNL**.

**SIGEV\_THREAD**

When a look-up completes, invoke *sigev\_notify\_function* as if it were the start function of a new thread. See [sigevent\(7\)](#) for details.

For **SIGEV\_SIGNAL** and **SIGEV\_THREAD**, it may be useful to point *sevp->sigev\_value.sival\_ptr* to *list*.

The **gai\_suspend()** function suspends execution of the calling thread, waiting for the completion of one or more requests in the array *list*. The *nitems* argument specifies the size of the array *list*. The call blocks until one of the following occurs:

- \* One or more of the operations in *list* completes.
- \* The call is interrupted by a signal that is caught.
- \* The time interval specified in *timeout* elapses. This argument specifies a timeout in seconds plus nanoseconds (see [nanosleep\(2\)](#) for details of the *timespec* structure). If *timeout* is NULL, then the call blocks indefinitely (until one of the events above occurs).

No explicit indication of which request was completed is given; you must determine which request(s) have completed by iterating with **gai\_error()** over the list of requests.

The **gai\_error()** function returns the status of the request *req*: either **EAI\_INPROGRESS** if the request was not completed yet, 0 if it was handled successfully, or an error code if the request could not be resolved.

The **gai\_cancel()** function cancels the request *req*. If the request has been canceled successfully, the error status of the request will be set to **EAI\_CANCELLED** and normal asynchronous notification will be performed. The request cannot be canceled if it is currently being processed; in that case, it will be handled as if **gai\_cancel()** has never been called. If *req* is NULL, an attempt is made to cancel all outstanding requests that the process has made.

**RETURN VALUE**

The **getaddrinfo\_a()** function returns 0 if all of the requests have been enqueued successfully, or one of the following nonzero error codes:

**EAI\_AGAIN**

The resources necessary to enqueue the look-up requests were not available. The application may check the error status of each request to determine which ones failed.

**EAI\_MEMORY**

Out of memory.

**EAI\_SYSTEM**

*mode* is invalid.

The **gai\_suspend()** function returns 0 if at least one of the listed requests has been completed. Otherwise, it returns one of the following nonzero error codes:

**EAI\_AGAIN**

The given timeout expired before any of the requests could be completed.

**EAI\_ALLDONE**

There were no actual requests given to the function.

**EAI\_INTR**

A signal has interrupted the function. Note that this interruption might have been caused by signal notification of some completed look-up request.

The **gai\_error()** function can return **EAI\_INPROGRESS** for an unfinished look-up request, 0

for a successfully completed look-up (as described above), one of the error codes that could be returned by `getaddrinfo(3)`, or the error code `EAI_CANCELLED` if the request has been canceled explicitly before it could be finished.

The `gai_cancel()` function can return one of these values:

**EAI\_CANCELLED**

The request has been canceled successfully.

**EAI\_NOTCANCELLED**

The request has not been canceled.

**EAI\_ALLDONE**

The request has already completed.

The `gai_strerror(3)` function translates these error codes to a human readable string, suitable for error reporting.

**CONFORMING TO**

These functions are GNU extensions; they first appeared in glibc in version 2.2.3.

**NOTES**

The interface of `getaddrinfo_a()` was modeled after the `lio_listio(3)` interface.

**EXAMPLE**

Two examples are provided: a simple example that resolves several requests in parallel synchronously, and a complex example showing some of the asynchronous capabilities.

**Synchronous example**

The program below simply resolves several hostnames in parallel, giving a speed-up compared to resolving the hostnames sequentially using `getaddrinfo(3)`. The program might be used like this:

```
$ ./a.out ftp.us.kernel.org enoent.linuxfoundation.org gnu.cz
ftp.us.kernel.org: 128.30.2.36
enoent.linuxfoundation.org: Name or service not known
gnu.cz: 87.236.197.13
```

Here is the program source code

```
#define _GNU_SOURCE
#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int
main(int argc, char *argv[])
{
    int i, ret;
    struct gaicb *reqs[argc - 1];
    char host[NI_MAXHOST];
    struct addrinfo *res;

    if (argc < 2) {
        fprintf(stderr, Usage: %s HOST...n, argv[0]);
        exit(EXIT_FAILURE);
    }

    for (i = 0; i < argc - 1; i++) {
        reqs[i] = malloc(sizeof(*reqs[0]));
        if (reqs[i] == NULL) {
            perror(malloc);
            exit(EXIT_FAILURE);
        }
    }
}
```

```

}
memset(reqs[i], 0, sizeof(*reqs[0]));
reqs[i]->ar_name = argv[i + 1];
}

ret = getaddrinfo_a(GAI_WAIT, reqs, argc - 1, NULL);
if (ret != 0) {
fprintf(stderr, getaddrinfo_a() failed: %sn,
gai_strerror(ret));
exit(EXIT_FAILURE);
}

for (i = 0; i < argc - 1; i++) {
printf(“%s: , reqs[i]->ar_name);
ret = gai_error(reqs[i]);
if (ret == 0) {
res = reqs[i]->ar_result;

ret = getnameinfo(res->ai_addr, res->ai_addrlen,
host, sizeof(host),
NULL, 0, NI_NUMERICHOST);
if (ret != 0) {
fprintf(stderr, getnameinfo() failed: %sn,
gai_strerror(ret));
exit(EXIT_FAILURE);
}
puts(host);
} else {
puts(gai_strerror(ret));
}
}
exit(EXIT_SUCCESS);
}

```

### Asynchronous example

This example shows a simple interactive `getaddrinfo_a()` front-end. The notification facility is not demonstrated.

An example session might look like this:

```

$ ./a.out
> a ftp.us.kernel.org enoent.linuxfoundation.org gnu.cz
> c 2
[2] gnu.cz: Request not canceled
> w 0 1
[00] ftp.us.kernel.org: Finished
> l
[00] ftp.us.kernel.org: 216.165.129.139
[01] enoent.linuxfoundation.org: Processing request in progress
[02] gnu.cz: 87.236.197.13
> l
[00] ftp.us.kernel.org: 216.165.129.139
[01] enoent.linuxfoundation.org: Name or service not known
[02] gnu.cz: 87.236.197.13

```

The program source is as follows:

```
#define _GNU_SOURCE
```

```

#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

static struct gaicb **reqs = NULL;
static int nreqs = 0;

static char *
getcmd(void)
{
    static char buf[256];

    fputs(> , stdout); fflush(stdout);
    if (fgets(buf, sizeof(buf), stdin) == NULL)
        return NULL;

    if (buf[strlen(buf) - 1] == '\n')
        buf[strlen(buf) - 1] = 0;

    return buf;
}

/* Add requests for specified hostnames */
static void
add_requests(void)
{
    int nreqs_base = nreqs;
    char *host;
    int ret;

    while ((host = strtok(NULL, " ")) != NULL) {
        nreqs++;
        reqs = realloc(reqs, nreqs * sizeof(reqs[0]));
        reqs[nreqs - 1] = calloc(1, sizeof(*reqs[0]));
        reqs[nreqs - 1]->ar_name = strdup(host);
    }

    /* Queue nreqs_base..nreqs requests. */
    ret = getaddrinfo_a(GAI_NOWAIT, &reqs[nreqs_base],
        nreqs - nreqs_base, NULL);
    if (ret) {
        fprintf(stderr, "getaddrinfo_a() failed: %sn",
            gai_strerror(ret));
        exit(EXIT_FAILURE);
    }
}

/* Wait until at least one of specified requests completes */
static void
wait_requests(void)
{
    char *id;
    int i, ret, n;
    struct gaicb const **wait_reqs = calloc(nreqs, sizeof(*wait_reqs));
    /* NULL elements are ignored by gai_suspend(). */
    while ((id = strtok(NULL, " ")) != NULL) {

```

```

n = atoi(id);
if (n >= nreqs) {
printf(Bad request number: %sn, id);
return;
}

wait_reqs[n] = reqs[n];
}

ret = gai_suspend(wait_reqs, nreqs, NULL);
if (ret) {
printf(gai_suspend(): %sn, gai_strerror(ret));
return;
}

for (i = 0; i < nreqs; i++) {
if (wait_reqs[i] == NULL)
continue;

ret = gai_error(reqs[i]);
if (ret == EAI_INPROGRESS)
continue;

printf("[%02d] %s: %sn, i, reqs[i]->ar_name,
ret == 0 ? Finished : gai_strerror(ret));
}
}

/* Cancel specified requests */
static void
cancel_requests(void)
{
char *id;
int ret, n;

while ((id = strtok(NULL, )) != NULL) {
n = atoi(id);

if (n >= nreqs) {
printf(Bad request number: %sn, id);
return;
}

ret = gai_cancel(reqs[n]);
printf("[%s] %s: %sn, id, reqs[atoi(id)]->ar_name,
gai_strerror(ret));
}
}

/* List all requests */
static void
list_requests(void)
{
int i, ret;
char host[NI_MAXHOST];
struct addrinfo *res;

for (i = 0; i < nreqs; i++) {
printf("[%02d] %s: , i, reqs[i]->ar_name);

```

```

ret = gai_error(reqs[i]);
if (!ret) {
res = reqs[i]->ar_result;

ret = getnameinfo(res->ai_addr, res->ai_addrlen,
host, sizeof(host),
NULL, 0, NI_NUMERICHOST);
if (ret) {
fprintf(stderr, getnameinfo() failed: %sn,
gai_strerror(ret));
exit(EXIT_FAILURE);
}
puts(host);
} else {
puts(gai_strerror(ret));
}
}
}

int
main(int argc, char *argv[])
{
char *cmdline;
char *cmd;

while ((cmdline = getcmd()) != NULL) {
cmd = strtok(cmdline, );

if (cmd == NULL) {
list_requests();
} else {
switch (cmd[0]) {
case a:
add_requests();
break;
case w:
wait_requests();
break;
case c:
cancel_requests();
break;
case l:
list_requests();
break;
default:
fprintf(stderr, Bad command: %cn, cmd[0]);
break;
}
}
}
exit(EXIT_SUCCESS);
}

```

**SEE ALSO**

[getaddrinfo\(3\)](#), [inet\(3\)](#), [lio\\_listio\(3\)](#), [hostname\(7\)](#), [ip\(7\)](#), [sigevent\(7\)](#)

**COLOPHON**

This page is part of release 3.74 of the Linux *man-pages* project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at <http://www.kernel.org/doc/man-pages/>.