Practical Case Study C

Operating Systems Programming – 300698

1 Introduction

In this case study you will implement a simple logging service built on top of a message queue.

2 Specification

The task is broken into three parts, a message logging server, a library to log messages, and a driver to test the library. You will need to review the lecture notes, and the documentation supplied in Section 5 to implement these programs.

2.1 Message Logging Server

The message logging server should attempt to create the message queue, if this fails then it should terminate with an error message, it should not run if the message queue actually exists (`IPC_EXCL` will help here).

Once connected to the message queue, the program should sit in a loop, receiving a message, and printing it to the `stdout`. Messages should be formatted:

`id: message`

where `id` is the type from the message structure and `message` is the message field.

The server should shutdown cleanly (i.e. delete the message queue) on receipt of a `SIGINT` (generated by pressing *control* and *C* keys at the same time).

The sample code files `logservice.h` and `logserver.c` should form the basis of your solution for this part.

2.2 Messaging library

The messaging library consists of two functions, both defined in `logservice.h`:

`int initLogService()`

> This function should initialise the message queue to log messages to, returning an `id` if successful, and `-1` on error.
> This function should not attempt to create the message queue, only attach it to the process.

`int logMessage(int id, char *message)`

> This function logs the message passed as the string `message` to the log service `id`. It should return `0` on success and `-1` on error.
> When sending a message, the function should encode the processes `pid` into the `type` field of the message, and the string into the `message` field.

It is your choice what to do if the message is too long (i.e. longer than `MSGCHARS`), sample behaviours include breaking the message up into smaller pieces or simply rejecting it. Whatever the choice, the documentation in the header file should reflect this choice.

The sample code files `logservice.h` and `logservice.c` should form the basis of your solution for this part.

2.3 Test Driver

This program is used to test the functionality of the library described in Section 2.2. It need not be complex, but is should be able to determine, and report if any errors have occurred.

The sample code files `logservice.h` and `logclient.c` should form the basis of your solution for this part.

3 Marking Scheme

Refer to the vUWS site for a marking rubric.

4 Sample Code

In addition to the sample code files, two additional files have been provided, a `makefile` that contains build rules, and a server launch script.

The `make` utility simplifies the build process for large projects, introductory documentation for make is included in the documentation section (Sec. 5). To use `make` to automate compile process simply type "`make`" at the terminal (in the same directory as the other files), it will use the rules defined in the `makefile` to build both the `logserver` and `logclient` executables from the source files, and it will also ensure that the `launch_server.sh` script is executable. If none of the source files have changed since the last build (based on their timestamps) the `make` utility will not rebuild the executables. There should be no need to modify the `makefile`, its format is a bit fussy so it is safer to download the file from *v*UWS than type it up.

The `launch_server.sh` script will open the `logserver` program in a new terminal window for you. This script detects the host operating system and performs an equivalent action after this detection. There is nop need to understand how this file achieves its goal.

4.1 logservice.h

```
/* logservice.h -- definitions for the  log service */
#ifndef LOGSERVICE_H /* prevent multiple inclusion */
#define LOGSERVICE_H
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

/* key for the queue */
#define KEY ftok("logservice.h", 'a')

/* message structure */
#define MSGCHARS 255
/* MSGCHARS is the number of characters in the message!*/
struct message
{
        long type;
        char message[MSGCHARS+1]; /* allow it to be a string! */
};

/* function prototypes */
int logServiceInit();
/* initialises the log service client, returns a service id */

int logMessage(int serviceId, char* message);
/* logs the message message to the log service serviceID */

#endif /* ifndef LOGSERVICE_H */
```

4.2 logservice.c

```
/* logservice.c -- implementation of the log service */
#include "logservice.h"

int logServiceInit()
{
        int id;

        return id;

}

int logMessage(int serviceId,char*message)
{
        int rv;

        return rv;
}
```

4.3 logclient.c

```
/* logclient.c -- implements a simple log service client */
#include "logservice.h"

int main(int argc,char**argv)
{
        printf("Make me useful too!\n");

        return 0;
}
```

4.4 logserver.c

```
/* logserver.c -- implementation of the log server */
#include <signal.h>
#include "logservice.h"

int main()
{
        printf("Please make me useful!\n");

        return 0;
}
```

4.5 makefile

```
# makefile -- rules to build OSP workshop C
# to use simply type "make"
# this will build the server and client and launcher script
# note, this is a configuration file for the MAKE utility
# do not try to run it directly
# if typing up the file, the indented lines need to be indented
# with TABS not spaces.

all: logserver logclient
        chmod +x launch_server.sh

clean:
        rm -f *.o logserver logclient

logclient: logclient.o logservice.o

logservice.o: logservice.c logservice.h

logserver: logserver.o

logserver.o: logserver.c logservice.h
```

4.6 launch_server.sh

```
#!/bin/bash
### This script launches the logserver process in a new window.
### Magic is needed for OSX as I can't rely on xterm being installed!
### Only works when logged in via the console, not Putty/SSH
### It is not necessary to understand this script!


if [ $(uname) == "Darwin" ]
then

        osascript -e 'tell application "Terminal" to do script "cd '$PWD'; \
                ./logserver; exit; "'
else
        xterm -title "Log Server" -e './logserver; \
                echo press enter to exit; read junk;' &
fi
```

5 Supplementary Materials

The material on the following pages is an extract of the linux system documentation and may prove useful in implementing this Workshop. These manual pages are taken from the Linux *man-pages* Project available at:
`http://www.kernel.org/doc/man-pages/`.

NAME

svipc − System V interprocess communication mechanisms

SYNOPSIS

 # include <sys/types.h>
 # include <sys/ipc.h>
 # include <sys/msg.h>
 # include <sys/sem.h>
 # include <sys/shm.h>

DESCRIPTION

This manual page refers to the Linux implementation of the System V interprocess communication mechanisms: message queues, semaphore sets, and shared memory segments. In the following, the word **resource** means an instantiation of one among such mechanisms.

Resource Access Permissions

For each resource, the system uses a common structure of type *struct ipc_perm* to store information needed in determining permissions to perform an ipc operation. The **ipc_perm** structure, defined by the <*sys/ipc.h*> system header file, includes the following members:

```
ushort cuid;    /* creator user ID */
ushort cgid;    /* creator group ID */
ushort uid;     /* owner user ID */
ushort gid;     /* owner group ID */
ushort mode;    /* r/w permissions */
```

The **mode** member of the **ipc_perm** structure defines, with its lower 9 bits, the access permissions to the resource for a process executing an ipc system call. The permissions are interpreted as follows:

```
0400    Read by user.
0200    Write by user.

0040    Read by group.
0020    Write by group.

0004    Read by others.
0002    Write by others.
```

Bits 0100, 0010, and 0001 (the execute bits) are unused by the system. Furthermore, "write" effectively means "alter" for a semaphore set.

The same system header file also defines the following symbolic constants:

IPC_CREAT Create entry if key doesn't exist.

IPC_EXCL Fail if key exists.

IPC_NOWAIT Error if request must wait.

IPC_PRIVATE Private key.

IPC_RMID Remove resource.

IPC_SET Set resource options.

IPC_STAT Get resource options.

Note that **IPC_PRIVATE** is a **key_t** type, while all the other symbolic constants are flag fields and can be OR'ed into an *int* type variable.

Message Queues

A message queue is uniquely identified by a positive integer (its *msqid*) and has an associated data structure of type *struct msqid_ds*, defined in <*sys/msg.h*>, containing the following members:

```
struct ipc_perm msg_perm;
ushort msg_qnum;     /* no of messages on queue */
ushort msg_qbytes;   /* bytes max on a queue */
ushort msg_lspid;    /* PID of last msgsnd() call */
ushort msg_lrpid;    /* PID of last msgrcv() call */
time_t msg_stime;    /* last msgsnd() time */
time_t msg_rtime;    /* last msgrcv() time */
time_t msg_ctime;    /* last change time */
```

msg_perm **ipc_perm** structure that specifies the access permissions on the message queue.

msg_qnum Number of messages currently on the message queue.

msg_qbytes Maximum number of bytes of message text allowed on the message queue.

msg_lspid ID of the process that performed the last **msgsnd**() system call.

msg_lrpid ID of the process that performed the last **msgrcv**() system call.

msg_stime Time of the last **msgsnd**() system call.

msg_rtime Time of the last **msgcv**() system call.

msg_ctime Time of the last system call that changed a member of the **msqid_ds** structure.

Semaphore Sets

A semaphore set is uniquely identified by a positive integer (its *semid*) and has an associated data structure of type *struct semid_ds*, defined in <*sys/sem.h*>, containing the following members:

```
struct ipc_perm sem_perm;
time_t sem_otime;    /* last operation time */
time_t sem_ctime;    /* last change time */
ushort sem_nsems;    /* count of sems in set */
```

sem_perm **ipc_perm** structure that specifies the access permissions on the semaphore set.

sem_otime Time of last **semop**() system call.

sem_ctime Time of last **semctl**() system call that changed a member of the above structure or of one semaphore belonging to the set.

sem_nsems Number of semaphores in the set. Each semaphore of the set is referenced by a non-negative integer ranging from **0** to **sem_nsems−1**.

A semaphore is a data structure of type *struct sem* containing the following members:

```
ushort semval;   /* semaphore value */
short sempid;    /* PID for last operation */
ushort semncnt; /* nr awaiting semval to increase */
ushort semzcnt; /* nr awaiting semval = 0 */
```

semval Semaphore value: a non-negative integer.

sempid ID of the last process that performed a semaphore operation on this semaphore.

semncnt Number of processes suspended awaiting for **semval** to increase.

semznt Number of processes suspended awaiting for **semval** to become zero.

Shared Memory Segments

A shared memory segment is uniquely identified by a positive integer (its *shmid*) and has an associated data structure of type *struct shmid_ds*, defined in <*sys/shm.h*>, containing the following members:

```
struct ipc_perm shm_perm;
int shm_segsz;   /* size of segment */
ushort shm_cpid;     /* PID of creator */
```

```
ushort shm_lpid;    /* PID, last operation */
short shm_nattch;   /* no. of current attaches */
time_t shm_atime;   /* time of last attach */
time_t shm_dtime;   /* time of last detach */
time_t shm_ctime;   /* time of last change */
```

shm_perm **ipc_perm** structure that specifies the access permissions on the shared memory segment.

shm_segsz Size in bytes of the shared memory segment.

shm_cpid ID of the process that created the shared memory segment.

shm_lpid ID of the last process that executed a **shmat**() or **shmdt**() system call.

shm_nattch Number of current alive attaches for this shared memory segment.

shm_atime Time of the last **shmat**() system call.

shm_dtime Time of the last **shmdt**() system call.

shm_ctime Time of the last **shmctl**() system call that changed **shmid_ds**.

SEE ALSO

msgctl(2), **msgget**(2), **msgrcv**(2), **msgsnd**(2), **semctl**(2), **semget**(2), **semop**(2), **shmat**(2), **shmctl**(2), **shmdt**(2), **shmget**(2), **ftok**(3)

NAME

ftok − convert a pathname and a project identifier to a System V IPC key

SYNOPSIS

 # include <sys/types.h>
 # include <sys/ipc.h>

key_t ftok(const char ****pathname*, int** *proj_id***);**

DESCRIPTION

The **ftok**() function uses the identity of the file named by the given *pathname* (which must refer to an existing, accessible file) and the least significant 8 bits of *proj_id* (which must be non-zero) to generate a **key_t** type System V IPC key, suitable for use with **msgget**(2), **semget**(2), or **shmget**(2).

The resulting value is the same for all pathnames that name the same file, when the same value of *proj_id* is used. The value returned should be different when the (simultaneously existing) files or the project IDs differ.

RETURN VALUE

On success the generated **key_t** value is returned. On failure −1 is returned, with *errno* indicating the error as for the **stat**(2) system call.

CONFORMING TO

POSIX.1-2001.

NOTES

Under libc4 and libc5 (and under SunOS 4.x) the prototype was

key_t ftok(char ****pathname*, char** *proj_id***);**

Today *proj_id* is an *int*, but still only 8 bits are used. Typical usage has an ASCII character *proj_id*, that is why the behaviour is said to be undefined when *proj_id* is zero.

Of course no guarantee can be given that the resulting **key_t** is unique. Typically, a best effort attempt combines the given *proj_id* byte, the lower 16 bits of the i−node number, and the lower 8 bits of the device number into a 32−bit result. Collisions may easily happen, for example between files on */dev/hda1* and files on */dev/sda1*.

SEE ALSO

msgget(2), **semget**(2), **shmget**(2), **stat**(2), **svipc**(7)

NAME

msgget − get a message queue identifier

SYNOPSIS

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgget(key_t *key***, int** *msgflg***);**

DESCRIPTION

The **msgget**() system call returns the message queue identifier associated with the value of the *key* argument. A new message queue is created if *key* has the value **IPC_PRIVATE** or *key* isn't **IPC_PRIVATE**, no message queue with the given key *key* exists, and **IPC_CREAT** is specified in *msgflg*.

If *msgflg* specifies both **IPC_CREAT** and **IPC_EXCL** and a message queue already exists for *key*, then **msgget**() fails with *errno* set to **EEXIST**. (This is analogous to the effect of the combination **O_CREAT | O_EXCL** for **open**(2).)

Upon creation, the least significant bits of the argument *msgflg* define the permissions of the message queue. These permission bits have the same format and semantics as the permissions specified for the *mode* argument of **open**(2). (The execute permissions are not used.)

If a new message queue is created, then its associated data structure *msqid_ds* (see **msgctl**(2)) is initialised as follows:

msg_perm.cuid and *msg_perm.uid* are set to the effective user ID of the calling process.

msg_perm.cgid and *msg_perm.gid* are set to the effective group ID of the calling process.

The least significant 9 bits of *msg_perm.mode* are set to the least significant 9 bits of *msgflg*.

msg_qnum, *msg_lspid*, *msg_lrpid*, *msg_stime* and *msg_rtime* are set to 0.

msg_ctime is set to the current time.

msg_qbytes is set to the system limit **MSGMNB**.

If the message queue already exists the permissions are verified, and a check is made to see if it is marked for destruction.

RETURN VALUE

If successful, the return value will be the message queue identifier (a nonnegative integer), otherwise −1 with *errno* indicating the error.

ERRORS

On failure, *errno* is set to one of the following values:

EACCES A message queue exists for *key*, but the calling process does not have permission to access the queue, and does not have the **CAP_IPC_OWNER** capability.

EEXIST A message queue exists for *key* and *msgflg* specified both **IPC_CREAT** and **IPC_EXCL**.

ENOENT No message queue exists for *key* and *msgflg* did not specify **IPC_CREAT**.

ENOMEM A message queue has to be created but the system does not have enough memory for the new data structure.

ENOSPC A message queue has to be created but the system limit for the maximum number of message queues (**MSGMNI**) would be exceeded.

NOTES

IPC_PRIVATE isn't a flag field but a *key_t* type. If this special value is used for *key*, the system call ignores everything but the least significant 9 bits of *msgflg* and creates a new message queue (on success).

The following is a system limit on message queue resources affecting a **msgget**() call:

MSGMNI System wide maximum number of message queues: policy dependent (on Linux, this limit can be read and modified via */proc/sys/kernel/msgmni*).

BUGS

The name choice IPC_PRIVATE was perhaps unfortunate, IPC_NEW would more clearly show its function.

CONFORMING TO

SVr4, POSIX.1-2001.

LINUX NOTES

Until version 2.3.20 Linux would return EIDRM for a **msgget**() on a message queue scheduled for deletion.

SEE ALSO

msgctl(2), **msgrcv**(2), **msgsnd**(2), **ftok**(3), **capabilities**(7), **mq_overview**(7), **svipc**(7)

NAME

msgctl − message control operations

SYNOPSIS

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgctl(int *msqid***, int** *cmd***, struct msqid_ds *****buf***);**

DESCRIPTION

msgctl() performs the control operation specified by *cmd* on the message queue with identifier *msqid*.

The *msqid_ds* data structure is defined in <sys/msg.h> as follows:

```
struct msqid_ds {
   struct ipc_perm msg_perm;     /* Ownership and permissions
   time_t        msg_stime;    /* Time of last msgsnd() */
   time_t        msg_rtime;    /* Time of last msgrcv() */
   time_t        msg_ctime;    /* Time of last change */
   unsigned long  __msg_cbytes; /* Current number of bytes in
                                   queue (non-standard) */
   msgqnum_t      msg_qnum;     /* Current number of messages
                                   in queue */
   msglen_t       msg_qbytes;   /* Maximum number of bytes
                                   allowed in queue */
   pid_t         msg_lspid;    /* PID of last msgsnd() */
   pid_t         msg_lrpid;    /* PID of last msgrcv() */
};
```

The *ipc_perm* structure is defined in <sys/ipc.h> as follows (the highlighted fields are settable using **IPC_SET**):

```
struct ipc_perm {
   key_t key;           /* Key supplied to msgget() */
   uid_t uid;           /* Effective UID of owner */
   gid_t gid;           /* Effective GID of owner */
   uid_t cuid;          /* Effective UID of creator */
   gid_t cgid;          /* Effective GID of creator */
   unsigned short mode;  /* Permissions */
   unsigned short seq;   /* Sequence number */
};
```

Valid values for *cmd* are:

IPC_STAT
: Copy information from the kernel data structure associated with *msqid* into the *msqid_ds* structure pointed to by *buf*. The caller must have read permission on the message queue.

IPC_SET
: Write the values of some members of the *msqid_ds* structure pointed to by *buf* to the kernel data structure associated with this message queue, updating also its *msg_ctime* member. The following members of the structure are updated: *msg_qbytes*, *msg_perm.uid*, *msg_perm.gid*, and (the least significant 9 bits of) *msg_perm.mode*. The effective UID of the calling process must match the owner (*msg_perm.uid*) or creator (*msg_perm.cuid*) of the message queue, or the caller must be privileged. Appropriate privilege (Linux: the **CAP_IPC_RESOURCE** capability) is required to raise the *msg_qbytes* value beyond the system parameter **MSGMNB**.

IPC_RMID
Immediately remove the message queue, awakening all waiting reader and writer processes (with an error return and *errno* set to **EIDRM**). The calling process must have appropriate privileges or its effective user ID must be either that of the creator or owner of the message queue.

IPC_INFO (Linux specific)
Returns information about system-wide message queue limits and parameters in the structure pointed to by *buf*. This structure is of type *msginfo* (thus, a cast is required), defined in *<sys/msg.h>* if the _GNU_SOURCE feature test macro is defined:

```
struct msginfo {
    int msgpool; /* Size in bytes of buffer pool used
                    to hold message data; unused */
    int msgmap;  /* Max. # of entries in message
                    map; unused */
    int msgmax;  /* Max. # of bytes that can be
                    written in a single message */
    int msgmnb;  /* Max. # of bytes that can be written to
                    queue; used to initialize msg_qbytes
                    during queue creation (msgget()) */
    int msgmni;  /* Max. # of message queues */
    int msgssz;  /* Message segment size; unused */
    int msgtql;  /* Max. # of messages on all queues
                    in system; unused */
    unsigned short int msgseg;
                /* Max. # of segments; unused */
};
```

The *msgmni*, *msgmax*, and *msgmnb* settings can be changed via */proc* files of the same name; see **proc**(5) for details.

MSG_INFO (Linux specific)
Returns a *msginfo* structure containing the same information as for **IPC_INFO**, except that the following fields are returned with information about system resources consumed by message queues: the *msgpool* field returns the number of message queues that currently exist on the system; the *msgmap* field returns the total number of messages in all queues on the system; and the *msgtql* field returns the total number of bytes in all messages in all queues on the system.

MSG_STAT (Linux specific)
Returns a *msqid_ds* structure as for **IPC_STAT**. However, the *msqid* argument is not a queue identifier, but instead an index into the kernel's internal array that maintains information about all message queues on the system.

RETURN VALUE

On success, **IPC_STAT**, **IPC_SET**, and **IPC_RMID** return 0. A successful **IPC_INFO** or **MSG_INFO** operation returns the index of the highest used entry in the kernel's internal array recording information about all message queues. (This information can be used with repeated **MSG_STAT** operations to obtain information about all queues on the system.) A successful **MSG_STAT** operation returns the identifier of the queue whose index was given in *msqid*.

On error, −1 is returned with *errno* indicating the error.

ERRORS

On failure, *errno* is set to one of the following:

EACCES The argument *cmd* is equal to **IPC_STAT** or **MSG_STAT**, but the calling process does not have read permission on the message queue *msqid*, and does not have the **CAP_IPC_OWNER** capability.

EFAULT The argument *cmd* has the value **IPC_SET** or **IPC_STAT**, but the address pointed to by *buf* isn't accessible.

EIDRM The message queue was removed.

EINVAL Invalid value for *cmd* or *msqid*. Or: for a **MSG_STAT** operation, the index value specified in *msqid* referred to an array slot that is currently unused.

EPERM The argument *cmd* has the value **IPC_SET** or **IPC_RMID**, but the effective user ID of the calling process is not the creator (as found in *msg_perm.cuid*) or the owner (as found in *msg_perm.uid*) of the message queue, and the process is not privileged (Linux: it does not have the **CAP_SYS_ADMIN** capability).

NOTES

The **IPC_INFO**, **MSG_STAT** and **MSG_INFO** operations are used by the **ipcs**(8) program to provide information on allocated resources. In the future these may modified or moved to a /proc file system interface.

Various fields in the *struct msqid_ds* were shorts under Linux 2.2 and have become longs under Linux 2.4. To take advantage of this, a recompilation under glibc-2.1.91 or later should suffice. (The kernel distinguishes old and new calls by an IPC_64 flag in *cmd*.)

CONFORMING TO

SVr4, POSIX.1-2001.

SEE ALSO

msgget(2), **msgrcv**(2), **msgsnd**(2), **capabilities**(7), **mq_overview**(7), **svipc**(7)

NAME

msgop − message operations

SYNOPSIS

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgsnd(int *msqid***, const void ****msgp***, size_t** *msgsz***, int** *msgflg***);**

ssize_t msgrcv(int *msqid***, void ****msgp***, size_t** *msgsz***, long** *msgtyp***, int** *msgflg***);**

DESCRIPTION

The **msgsnd**() and **msgrcv**() system calls are used, respectively, to send messages to, and receive messages from, a message queue. The calling process must have write permission on the message queue in order to send a message, and read permission to receive a message.

The *msgp* argument is a pointer to caller-defined structure of the following general form:

```
struct msgbuf {
        long    mtype;   /* message type, must be > 0 */
        char    mtext[1];        /* message data */
};
```

The *mtext* field is an array (or other structure) whose size is specified by *msgsz*, a non-negative integer value. Messages of zero length (i.e., no *mtext* field) are permitted. The *mtype* field must have a strictly positive integer value. This value can be used by the receiving process for message selection (see the description of **msgrcv**() below).

The **msgsnd**() system call appends a copy of the message pointed to by *msgp* to the message queue whose identifier is specified by *msqid*.

If sufficient space is available in the queue, **msgsnd**() succeeds immediately. (The queue capacity is defined by the *msg_bytes* field in the associated data structure for the message queue. During queue creation this field is initialised to **MSGMNB** bytes, but this limit can be modified using **msgctl**().) If insufficient space is available in the queue, then the default behaviour of **msgsnd**() is to block until space becomes available. If **IPC_NOWAIT** is specified in *msgflg*, then the call instead fails with the error **EAGAIN**.

A blocked **msgsnd**() call may also fail if the queue is removed (in which case the system call fails with *errno* set to **EIDRM**), or a signal is caught (in which case the system call fails with *errno* set to **EINTR**). (**msgsnd** and **msgrcv** are never automatically restarted after being interrupted by a signal handler, regardless of the setting of the **SA_RESTART** flag when establishing a signal handler.)

Upon successful completion the message queue data structure is updated as follows:

msg_lspid is set to the process ID of the calling process.

msg_qnum is incremented by 1.

msg_stime is set to the current time.

The system call **msgrcv**() removes a message from the queue specified by *msqid* and places it in the buffer pointed to *msgp*.

The argument *msgsz* specifies the maximum size in bytes for the member *mtext* of the structure pointed to by the *msgp* argument. If the message text has length greater than *msgsz*, then the behaviour depends on whether **MSG_NOERROR** is specified in *msgflg*. If **MSG_NOERROR** is specified, then the message text will be truncated (and the truncated part will be lost); if **MSG_NOERROR** is not specified, then the message isn't removed from the queue and the system call fails returning −1 with *errno* set to **E2BIG**.

The argument *msgtyp* specifies the type of message requested as follows:

If *msgtyp* is 0, then the first message in the queue is read.

If *msgtyp* is greater than 0, then the first message in the queue of type *msgtyp* is read, unless **MSG_EXCEPT** was specified in *msgflg*, in which case the first message in the queue of type not equal to *msgtyp* will be read.

If *msgtyp* is less than 0, then the first message in the queue with the lowest type less than or equal to the absolute value of *msgtyp* will be read.

The *msgflg* argument is a bit mask constructed by ORing together zero or more of the following flags:

IPC_NOWAIT
Return immediately if no message of the requested type is in the queue. The system call fails with *errno* set to **ENOMSG**.

MSG_EXCEPT
Used with *msgtyp* greater than 0 to read the first message in the queue with message type that differs from *msgtyp*.

MSG_NOERROR
To truncate the message text if longer than *msgsz* bytes.

If no message of the requested type is available and **IPC_NOWAIT** isn't specified in *msgflg*, the calling process is blocked until one of the following conditions occurs:

A message of the desired type is placed in the queue.

The message queue is removed from the system. In this case the system call fails with *errno* set to **EIDRM**.

The calling process catches a signal. In this case the system call fails with *errno* set to **EINTR**.

Upon successful completion the message queue data structure is updated as follows:

msg_lrpid is set to the process ID of the calling process.

msg_qnum is decremented by 1.

msg_rtime is set to the current time.

RETURN VALUE

On failure both functions return −1 with *errno* indicating the error, otherwise **msgsnd**() returns 0 and **msgrcv**() returns the number of bytes actually copied into the *mtext* array.

ERRORS

When **msgsnd**() fails, *errno* will be set to one among the following values:

EACCES The calling process does not have write permission on the message queue, and does not have the **CAP_IPC_OWNER** capability.

EAGAIN The message can't be sent due to the *msg_qbytes* limit for the queue and **IPC_NOWAIT** was specified in *msgflg*.

EFAULT The address pointed to by *msgp* isn't accessible.

EIDRM The message queue was removed.

EINTR Sleeping on a full message queue condition, the process caught a signal.

EINVAL Invalid *msqid* value, or non-positive *mtype* value, or invalid *msgsz* value (less than 0 or greater than the system value **MSGMAX**).

ENOMEM The system does not have enough memory to make a copy of the message pointed to by *msgp*.

When **msgrcv**() fails, *errno* will be set to one among the following values:

E2BIG The message text length is greater than *msgsz* and **MSG_NOERROR** isn't specified in *msgflg*.

EACCES The calling process does not have read permission on the message queue, and does not have the **CAP_IPC_OWNER** capability.

EAGAIN No message was available in the queue and **IPC_NOWAIT** was specified in *msgflg*.

EFAULT The address pointed to by *msgp* isn't accessible.

EIDRM While the process was sleeping to receive a message, the message queue was removed.

EINTR While the process was sleeping to receive a message, the process caught a signal.

EINVAL *msgqid* was invalid, or *msgsz* was less than 0.

ENOMSG **IPC_NOWAIT** was specified in *msgflg* and no message of the requested type existed on the message queue.

CONFORMING TO

SVr4, POSIX.1-2001.

NOTES

The *msgp* argument is declared as *struct msgbuf* * with libc4, libc5, glibc 2.0, glibc 2.1. It is declared as *void* * with glibc 2.2 and later, as required by SUSv2 and SUSv3.

The following limits on message queue resources affect the **msgsnd**() call:

MSGMAX Maximum size for a message text: 8192 bytes (on Linux, this limit can be read and modified via */proc/sys/kernel/msgmax*).

MSGMNB Default maximum size in bytes of a message queue: 16384 bytes (on Linux, this limit can be read and modified via */proc/sys/kernel/msgmnb*). The superuser can increase the size of a message queue beyond **MSGMNB** by a **msgctl**() system call.

The implementation has no intrinsic limits for the system wide maximum number of message headers (**MSGTQL**) and for the system wide maximum size in bytes of the message pool (**MSGPOOL**).

SEE ALSO

msgctl(2), **msgget**(2), **msgrcv**(2), **msgsnd**(2), **capabilities**(7), **mq_overview**(7), **svipc**(7)

NAME

getpid, getppid − get process identification

SYNOPSIS

#include <sys/types.h>
#include <unistd.h>

pid_t getpid(void);
pid_t getppid(void);

DESCRIPTION

getpid() returns the process ID of the current process. (This is often used by routines that generate unique temporary filenames.)

getppid() returns the process ID of the parent of the current process.

CONFORMING TO

POSIX.1-2001, 4.3BSD, SVr4

SEE ALSO

fork(2), **kill**(2), **exec**(3), **mkstemp**(3), **tempnam**(3), **tmpfile**(3), **tmpnam**(3)

NAME

signal − ANSI C signal handling

SYNOPSIS

#include <signal.h>

typedef void (*sighandler_t)(int);

sighandler_t signal(int *signum***, sighandler_t** *handler***);**

DESCRIPTION

The **signal**() system call installs a new signal handler for the signal with number *signum*. The signal handler is set to *sighandler* which may be a user specified function, or either **SIG_IGN** or **SIG_DFL**.

Upon arrival of a signal with number *signum* the following happens. If the corresponding handler is set to **SIG_IGN**, then the signal is ignored. If the handler is set to **SIG_DFL**, then the default action associated with the signal (see **signal**(7)) occurs. Finally, if the handler is set to a function *sighandler* then first either the handler is reset to SIG_DFL or an implementation-dependent blocking of the signal is performed and next *sighandler* is called with argument *signum*.

Using a signal handler function for a signal is called "catching the signal". The signals **SIGKILL** and **SIGSTOP** cannot be caught or ignored.

RETURN VALUE

The **signal**() function returns the previous value of the signal handler, or **SIG_ERR** on error.

PORTABILITY

The original Unix **signal**() would reset the handler to SIG_DFL, and System V (and the Linux kernel and libc4,5) does the same. On the other hand, BSD does not reset the handler, but blocks new instances of this signal from occurring during a call of the handler. The glibc2 library follows the BSD behaviour.

If one on a libc5 system includes **<bsd/signal.h>** instead of **<signal.h>** then **signal**() is redefined as **__bsd_signal** and signal has the BSD semantics. This is not recommended.

If one on a glibc2 system defines a feature test macro such as **_XOPEN_SOURCE** or uses a separate **sysv_signal** function, one obtains classical behaviour. This is not recommended.

Trying to change the semantics of this call using defines and includes is not a good idea. It is better to avoid **signal**() altogether, and use **sigaction**(2) instead.

NOTES

The effects of this call in a multi-threaded process are unspecified.

The routine *handler* must be very careful, since processing elsewhere was interrupted at some arbitrary point. POSIX has the concept of "safe function". If a signal interrupts an unsafe function, and *handler* calls an unsafe function, then the behavior is undefined. Safe functions are listed explicitly in the various standards. The POSIX.1-2003 list is

_Exit() _exit() abort() accept() access() aio_error() aio_return() aio_suspend() alarm() bind() cfgetispeed() cfgetospeed() cfsetispeed() cfsetospeed() chdir() chmod() chown() clock_gettime() close() connect() creat() dup() dup2() execle() execve() fchmod() fchown() fcntl() fdatasync() fork() fpathconf() fstat() fsync() ftruncate() getegid() geteuid() getgid() getgroups() getpeername() getpgrp() getpid() getppid() getsockname() getsockopt() getuid() kill() link() listen() lseek() lstat() mkdir() mkfifo() open() pathconf() pause() pipe() poll() posix_trace_event() pselect() raise() read() readlink() recv() recvfrom() recvmsg() rename() rmdir() select() sem_post() send() sendmsg() sendto() setgid() setpgid() setsid() setsockopt() setuid() shutdown() sigaction() sigaddset() sigdelset() sigemptyset() sigfillset() sigismember() signal() sigpause() sigpending() sigprocmask() sigqueue() sigset() sigsuspend() sleep() socket() socketpair() stat() symlink() sysconf() tcdrain() tcflow() tcflush() tcgetattr() tcgetpgrp() tcsendbreak() tcsetattr() tcsetpgrp() time()

timer_getoverrun() timer_gettime() timer_settime() times() umask() uname() unlink() utime() wait() waitpid() write().

According to POSIX, the behaviour of a process is undefined after it ignores a **SIGFPE**, **SIGILL**, or **SIGSEGV** signal that was not generated by the **kill**(2) or the **raise**(3) functions. Integer division by zero has undefined result. On some architectures it will generate a **SIGFPE** signal. (Also dividing the most negative integer by −1 may generate **SIGFPE**.) Ignoring this signal might lead to an endless loop.

See **sigaction**(2) for details on what happens when **SIGCHLD** is set to **SIG_IGN**.

The use of **sighandler_t** is a GNU extension. Various versions of libc predefine this type; libc4 and libc5 define *SignalHandler*, glibc defines *sig_t* and, when **_GNU_SOURCE** is defined, also *sighandler_t*.

CONFORMING TO

C89, POSIX.1-2001.

SEE ALSO

kill(1), **alarm**(2), **kill**(2), **pause**(2), **sigaction**(2), **sigpending**(2), **sigprocmask**(2), **sigqueue**(2), **sigsuspend**(2), **killpg**(3), **raise**(3), **sigsetops**(3), **sigvec**(3), **feature_test_macros**(7), **signal**(7)

NAME

sigaction − examine and change a signal action

SYNOPSIS

#include <signal.h>

int sigaction(int *signum***, const struct sigaction ****act***, struct sigaction ****oldact***);**

DESCRIPTION

The **sigaction**() system call is used to change the action taken by a process on receipt of a specific signal.

signum specifies the signal and can be any valid signal except **SIGKILL** and **SIGSTOP**.

If *act* is non−null, the new action for signal *signum* is installed from *act*. If *oldact* is non−null, the previous action is saved in *oldact*.

The *sigaction* structure is defined as something like

```
struct sigaction {
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask;
    int sa_flags;
    void (*sa_restorer)(void);
}
```

On some architectures a union is involved: do not assign to both *sa_handler* and *sa_sigaction*.

The *sa_restorer* element is obsolete and should not be used. POSIX does not specify a *sa_restorer* element.

sa_handler specifies the action to be associated with *signum* and may be **SIG_DFL** for the default action, **SIG_IGN** to ignore this signal, or a pointer to a signal handling function. This function receives the signal number as its only argument.

If **SA_SIGINFO** is specified in *sa_flags*, then *sa_sigaction* (instead of *sa_handler*) specifies the signal-handling function for *signum*. This function receives the signal number as its first argument, a pointer to a *siginfo_t* as its second argument and a pointer to a *ucontext_t* (cast to void *) as its third argument.

sa_mask gives a mask of signals which should be blocked during execution of the signal handler. In addition, the signal which triggered the handler will be blocked, unless the **SA_NODEFER** flag is used.

sa_flags specifies a set of flags which modify the behaviour of the signal handling process. It is formed by the bitwise OR of zero or more of the following:

SA_NOCLDSTOP
: If *signum* is **SIGCHLD**, do not receive notification when child processes stop (i.e., when they receive one of **SIGSTOP**, **SIGTSTP**, **SIGTTIN** or **SIGTTOU**) or resume (i.e., they receive **SIGCONT**) (see **wait**(2)).

SA_NOCLDWAIT
: (Linux 2.6 and later) If *signum* is **SIGCHLD**, do not transform children into zombies when they terminate. See also **waitpid**(2).

SA_RESETHAND
: Restore the signal action to the default state once the signal handler has been called. **SA_ONESHOT** is an obsolete, non-standard synonym for this flag.

SA_ONSTACK
: Call the signal handler on an alternate signal stack provided by **sigaltstack**(2). If an alternate stack is not available, the default stack will be used.

SA_RESTART
Provide behaviour compatible with BSD signal semantics by making certain system calls restartable across signals.

SA_NODEFER
Do not prevent the signal from being received from within its own signal handler. **SA_NOMASK** is an obsolete, non-standard synonym for this flag.

SA_SIGINFO
The signal handler takes 3 arguments, not one. In this case, *sa_sigaction* should be set instead of *sa_handler*. (The *sa_sigaction* field was added in Linux 2.1.86.)

The *siginfo_t* parameter to *sa_sigaction* is a struct with the following elements

```
siginfo_t {
    int     si_signo;   /* Signal number */
    int     si_errno;   /* An errno value */
    int     si_code;    /* Signal code */
    pid_t   si_pid;     /* Sending process ID */
    uid_t   si_uid;     /* Real user ID of sending process */
    int     si_status;  /* Exit value or signal */
    clock_t si_utime;   /* User time consumed */
    clock_t si_stime;   /* System time consumed */
    sigval_t si_value;  /* Signal value */
    int     si_int;     /* POSIX.1b signal */
    void *  si_ptr;     /* POSIX.1b signal */
    void *  si_addr;    /* Memory location which caused fault */
    int     si_band;    /* Band event */
    int     si_fd;      /* File descriptor */
}
```

si_signo, *si_errno* and *si_code* are defined for all signals. (*si_signo* is unused on Linux.) The rest of the struct may be a union, so that one should only read the fields that are meaningful for the given signal. POSIX.1b signals and **SIGCHLD** fill in *si_pid* and *si_uid*. **SIGCHLD** also fills in *si_status*, *si_utime* and *si_stime*. *si_int* and *si_ptr* are specified by the sender of the POSIX.1b signal. **SIGILL**, **SIGFPE**, **SIGSEGV**, and **SIGBUS** fill in *si_addr* with the address of the fault. **SIGPOLL** fills in *si_band* and *si_fd*.

si_code indicates why this signal was sent. It is a value, not a bitmask. The values which are possible for any signal are listed in this table:

si_code	
Value	Signal origin
SI_USER	kill(), sigsend(), or raise()
SI_KERNEL	The kernel
SI_QUEUE	sigqueue()
SI_TIMER	POSIX timer expired
SI_MESGQ	POSIX message queue state changed (since Linux 2.6.6)
SI_ASYNCIO	AIO completed
SI_SIGIO	queued SIGIO
SI_TKILL	tkill() or tgkill() (since Linux 2.4.19)

SIGILL	
ILL_ILLOPC	illegal opcode
ILL_ILLOPN	illegal operand
ILL_ILLADR	illegal addressing mode
ILL_ILLTRP	illegal trap
ILL_PRVOPC	privileged opcode
ILL_PRVREG	privileged register
ILL_COPROC	coprocessor error
ILL_BADSTK	internal stack error

SIGFPE	
FPE_INTDIV	integer divide by zero
FPE_INTOVF	integer overflow
FPE_FLTDIV	floating point divide by zero
FPE_FLTOVF	floating point overflow
FPE_FLTUND	floating point underflow
FPE_FLTRES	floating point inexact result
FPE_FLTINV	floating point invalid operation
FPE_FLTSUB	subscript out of range

SIGSEGV	
SEGV_MAPERR	address not mapped to object
SEGV_ACCERR	invalid permissions for mapped object

SIGBUS	
BUS_ADRALN	invalid address alignment
BUS_ADRERR	non-existent physical address
BUS_OBJERR	object specific hardware error

SIGTRAP	
TRAP_BRKPT	process breakpoint
TRAP_TRACE	process trace trap

SIGCHLD	
CLD_EXITED	child has exited
CLD_KILLED	child was killed
CLD_DUMPED	child terminated abnormally
CLD_TRAPPED	traced child has trapped
CLD_STOPPED	child has stopped
CLD_CONTINUED	stopped child has continued (since Linux 2.6.9)

SIGPOLL	
POLL_IN	data input available
POLL_OUT	output buffers available
POLL_MSG	input message available
POLL_ERR	i/o error
POLL_PRI	high priority input available
POLL_HUP	device disconnected

RETURN VALUE

sigaction() returns 0 on success and −1 on error.

ERRORS

EFAULT

act or *oldact* points to memory which is not a valid part of the process address space.

EINVAL

An invalid signal was specified. This will also be generated if an attempt is made to change the action for **SIGKILL** or **SIGSTOP**, which cannot be caught or ignored.

NOTES

According to POSIX, the behaviour of a process is undefined after it ignores a **SIGFPE**, **SIGILL**, or **SIGSEGV** signal that was not generated by **kill**() or **raise**(). Integer division by zero has undefined result. On some architectures it will generate a **SIGFPE** signal. (Also dividing the most negative integer by −1 may generate SIGFPE.) Ignoring this signal might lead to an endless loop.

POSIX.1-1990 disallowed setting the action for **SIGCHLD** to **SIG_IGN**. POSIX.1-2001 allows this possibility, so that ignoring **SIGCHLD** can be used to prevent the creation of zombies (see **wait**(2)). Nevertheless, the historical BSD and System V behaviours for ignoring **SIGCHLD** differ, so that the only completely portable method of ensuring that terminated children do not become zombies is to catch the **SIGCHLD** signal and perform a **wait**(2) or similar.

POSIX.1-1990 only specified **SA_NOCLDSTOP**. POSIX.1-2001 added **SA_NOCLDWAIT**, **SA_RESETHAND**, **SA_NODEFER**, and **SA_SIGINFO**. Use of these latter values in *sa_flags* may be less portable in applications intended for older Unix implementations.

Support for **SA_SIGINFO** was added in Linux 2.2.

The **SA_RESETHAND** flag is compatible with the SVr4 flag of the same name.

The **SA_NODEFER** flag is compatible with the SVr4 flag of the same name under kernels 1.3.9 and newer. On older kernels the Linux implementation allowed the receipt of any signal, not just the one we are installing (effectively overriding any *sa_mask* settings).

sigaction() can be called with a null second argument to query the current signal handler. It can also be used to check whether a given signal is valid for the current machine by calling it with null second and third arguments.

It is not possible to block **SIGKILL** or **SIGSTOP** (by specifying them in *sa_mask*). Attempts to do so are silently ignored.

See **sigsetops**(3) for details on manipulating signal sets.

BUGS

In kernels up to and including 2.6.13, specifying **SA_NODEFER** in *sa_flags* preventing not only the delivered signal from being masked during execution of the handler, but also the signals specified in *sa_mask*. This bug is was fixed in kernel 2.6.14.

CONFORMING TO

POSIX.1-2001, SVr4.

UNDOCUMENTED

Before the introduction of **SA_SIGINFO** it was also possible to get some additional information, namely by using a *sa_handler* with second argument of type *struct sigcontext*. See the relevant kernel sources for details. This use is obsolete now.

SEE ALSO

kill(1), **kill**(2), **pause**(2), **sigaltstack**(2), **signal**(2), **sigpending**(2), **sigprocmask**(2), **sigqueue**(2), **sigsuspend**(2), **wait**(2), **killpg**(3), **raise**(3), **siginterrupt**(3), **sigsetops**(3), **sigvec**(3), **core**(5), **signal**(7)

NAME

make − GNU make utility to maintain groups of programs

SYNOPSIS

make [**−f** *makefile*] [options] ... [targets] ...

WARNING

This man page is an extract of the documentation of GNU *make*. It is updated only occasionally, because the GNU project does not use nroff. For complete, current documentation, refer to the Info file **make.info** which is made from the Texinfo source file **make.texi**.

DESCRIPTION

The purpose of the *make* utility is to determine automatically which pieces of a large program need to be recompiled, and issue the commands to recompile them. The manual describes the GNU implementation of *make*, which was written by Richard Stallman and Roland McGrath, and is currently maintained by Paul Smith. Our examples show C programs, since they are most common, but you can use *make* with any programming language whose compiler can be run with a shell command. In fact, *make* is not limited to programs. You can use it to describe any task where some files must be updated automatically from others whenever the others change.

To prepare to use *make*, you must write a file called the *makefile* that describes the relationships among files in your program, and the states the commands for updating each file. In a program, typically the executable file is updated from object files, which are in turn made by compiling source files.

Once a suitable makefile exists, each time you change some source files, this simple shell command:

make

suffices to perform all necessary recompilations. The *make* program uses the makefile data base and the last-modification times of the files to decide which of the files need to be updated. For each of those files, it issues the commands recorded in the data base.

make executes commands in the *makefile* to update one or more target *names*, where *name* is typically a program. If no **−f** option is present, *make* will look for the makefiles *GNUmakefile*, *makefile*, and *Makefile*, in that order.

Normally you should call your makefile either *makefile* or *Makefile*. (We recommend *Makefile* because it appears prominently near the beginning of a directory listing, right near other important files such as *README*.) The first name checked, *GNUmakefile*, is not recommended for most makefiles. You should use this name if you have a makefile that is specific to GNU *make*, and will not be understood by other versions of *make*. If *makefile* is '−', the standard input is read.

make updates a target if it depends on prerequisite files that have been modified since the target was last modified, or if the target does not exist.

OPTIONS

−b, **−m** These options are ignored for compatibility with other versions of *make*.

−B, **−−always−make**
Unconditionally make all targets.

−C *dir*, **−−directory**=*dir*
Change to directory *dir* before reading the makefiles or doing anything else. If multiple **−C** options are specified, each is interpreted relative to the previous one: **−C** / **−C** etc is equivalent to **−C** /etc. This is typically used with recursive invocations of *make*.

−d Print debugging information in addition to normal processing. The debugging information says which files are being considered for remaking, which file-times are being compared and with what results, which files actually need to be remade, which implicit rules are considered and which are applied---everything interesting about how *make* decides what to do.

−−debug*[=FLAGS]*
Print debugging information in addition to normal processing. If the *FLAGS* are omitted, then the behavior is the same as if **−d** was specified. *FLAGS* may be *a* for all debugging output (same as using **−d**), *b* for basic debugging, *v* for more verbose basic debugging, *i* for showing implicit rules, *j* for details on invocation of commands, and *m* for debugging while remaking makefiles.

−e, **−−environment−overrides**
Give variables taken from the environment precedence over variables from makefiles.

+**−f** *file*, **−−file**=*file*, **−−makefile**=*FILE*
Use *file* as a makefile.

−i, **−−ignore−errors**
Ignore all errors in commands executed to remake files.

−I *dir*, **−−include−dir**=*dir*
Specifies a directory *dir* to search for included makefiles. If several **−I** options are used to specify several directories, the directories are searched in the order specified. Unlike the arguments to other flags of *make*, directories given with **−I** flags may come directly after the flag: **−I***dir* is allowed, as well as **−I** *dir*. This syntax is allowed for compatibility with the C preprocessor's **−I** flag.

−j [*jobs*], **−−jobs**[=*jobs*]
Specifies the number of *jobs* (commands) to run simultaneously. If there is more than one **−j** option, the last one is effective. If the **−j** option is given without an argument, *make* will not limit the number of jobs that can run simultaneously.

−k, **−−keep−going**
Continue as much as possible after an error. While the target that failed, and those that depend on it, cannot be remade, the other dependencies of these targets can be processed all the same.

−l [*load*], **−−load−average**[=*load*]
Specifies that no new jobs (commands) should be started if there are others jobs running and the load average is at least *load* (a floating-point number). With no argument, removes a previous load limit.

−L, **−−check−symlink−times**
Use the latest mtime between symlinks and target.

−n, **−−just−print**, **−−dry−run**, **−−recon**
Print the commands that would be executed, but do not execute them.

−o *file*, **−−old−file**=*file*, **−−assume−old**=*file*
Do not remake the file *file* even if it is older than its dependencies, and do not remake anything on account of changes in *file*. Essentially the file is treated as very old and its rules are ignored.

−p, **−−print−data−base**
Print the data base (rules and variable values) that results from reading the makefiles; then execute as usual or as otherwise specified. This also prints the version information given by the **−v** switch (see below). To print the data base without trying to remake any files, use **make −p −f***/dev/null*.

−q, **−−question**
''Question mode''. Do not run any commands, or print anything; just return an exit status that is zero if the specified targets are already up to date, nonzero otherwise.

−r, **−−no−builtin−rules**
Eliminate use of the built−in implicit rules. Also clear out the default list of suffixes for suffix rules.

−R, **−−no−builtin−variables**
Don't define any built−in variables.

−s, **−−silent**, **−−quiet**
Silent operation; do not print the commands as they are executed.

−S, **−−no−keep−going**, **−−stop**
Cancel the effect of the **−k** option. This is never necessary except in a recursive *make* where **−k** might be inherited from the top-level *make* via MAKEFLAGS or if you set **−k** in MAKEFLAGS in your environment.

−t, **−−touch**
Touch files (mark them up to date without really changing them) instead of running their commands. This is used to pretend that the commands were done, in order to fool future invocations of *make*.

−v, **−−version**
Print the version of the *make* program plus a copyright, a list of authors and a notice that there is no warranty.

−w, **−−print−directory**
Print a message containing the working directory before and after other processing. This may be useful for tracking down errors from complicated nests of recursive *make* commands.

−−no−print−directory
Turn off **−w**, even if it was turned on implicitly.

−W *file*, **−−what−if**=*file*, **−−new−file**=*file*, **−−assume−new**=*file*
Pretend that the target *file* has just been modified. When used with the **−n** flag, this shows you what would happen if you were to modify that file. Without **−n**, it is almost the same as running a *touch* command on the given file before running *make*, except that the modification time is changed only in the imagination of *make*.

−−warn−undefined−variables
Warn when an undefined variable is referenced.

EXIT STATUS

GNU *make* exits with a status of zero if all makefiles were successfully parsed and no targets that were built failed. A status of one will be returned if the **−q** flag was used and *make* determines that a target needs to be rebuilt. A status of two will be returned if any errors were encountered.

SEE ALSO

The GNU Make Manual

BUGS

See the chapter ‘Problems and Bugs’ in *The GNU Make Manual*.

AUTHOR

This manual page contributed by Dennis Morse of Stanford University. It has been reworked by Roland McGrath. Further updates contributed by Mike Frysinger.

COPYRIGHT