brk() and malloc()

Check all the true claims about brk() and malloc(). To help you, we provide the manul page of *brk*() below.

--

BRK(2) Linux Programmer's Manual BRK(2)

NAME
brk, sbrk - change data segment size

SYNOPSIS
#include <unistd.h>

int brk(void *addr);

void *sbrk(intptr_t increment);

Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

brk(), sbrk():
Since glibc 2.19:
_DEFAULT_SOURCE ||
(_XOPEN_SOURCE >= 500) &&
! (_POSIX_C_SOURCE >= 200112L)
From glibc 2.12 to 2.19:
_BSD_SOURCE || _SVID_SOURCE ||
(_XOPEN_SOURCE >= 500) &&
! (_POSIX_C_SOURCE >= 200112L)
Before glibc 2.12:
_BSD_SOURCE || _SVID_SOURCE || _XOPEN_SOURCE >= 500

DESCRIPTION
brk() and sbrk() change the location of the program break, which defines the end of the process's data segment (i.e., the
program break is the first location after the end of the uninitialized data segment). Increasing the program break has the
effect of allocating memory to the process; decreasing the break deallocates memory.

brk() sets the end of the data segment to the value specified by addr, when that value is reasonable, the system has enough
memory, and the process does not exceed its maximum data size (see setrlimit(2)).

sbrk() increments the program's data space by increment bytes. Calling

sbrk() with an increment of 0 can be used to find
the current location of the program break.

RETURN VALUE
On success, brk() returns zero. On error, -1 is returned, and errno is set to ENOMEM.

On success, sbrk() returns the previous program break. (If the break was increased, then this value is a pointer to the
start of the newly allocated memory). On error, (void *) -1 is returned, and errno is set to ENOMEM.

CONFORMING TO
4.3BSD; SUSv1, marked LEGACY in SUSv2, removed in POSIX.1-2001.

NOTES
Avoid using brk() and sbrk(): the malloc(3) memory allocation package is the portable and comfortable way of allocating memory.

Various systems use various types for the argument of sbrk(). Common are int, ssize_t, ptrdiff_t, intptr_t.

C library/kernel differences
The return value described above for brk() is the behavior provided by the glibc wrapper function for the Linux brk() system
call. (On most other implementations, the return value from brk() is the same; this return value was also specified in
SUSv2.) However, the actual Linux system call returns the new program break on success. On failure, the system call re-
turns the current break. The glibc wrapper function does some work (i.e., checks whether the new break is less than addr)
to provide the 0 and -1 return values described above.

On Linux, sbrk() is implemented as a library function that uses the brk() system call, and does some internal bookkeeping so
that it can return the old break value.

SEE ALSO
execve(2), getrlimit(2), end(3), malloc(3)

COLOPHON
This page is part of release 4.16 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 https://www.kernel.org/doc/man-pages/.

- [ ] This manual page of *brk*() applies to all Linux 4 versions
- [ ] *brk*() can be used to allocate memory
- [ ] *brk*() can be used to disallocate memory
- [ ] *malloc*() can be used to allocate memory
- [ ] *malloc*() can be used to disallocate memory
- [ ] *malloc*() is a syscall
- [ ] *brk*() fails if and only if not enough memory is available
- [ ] *brk*() is a syscall

Chunks of malloc

malloc() manages memory at user level. It requires pages from the OS if necessary, and handles the allocated pages in a fine way, thus allowing a user to require from one byte to billions of bytes. The structure used for this management is called a chunk. A chunk is defined as follows in the documentation of malloc:

An allocated chunk looks like this:

Chunks of malloc

Check all the true claims below.

- [ ]

This figure shows that a linked list is used to store allocated chunks

☐

Chunks reference each other with a size information

☐

Size reserved for application data is stored in the chunk header

☐

The memory where chunk header is saved is not writable by the application

☐

if a process writes more bytes than the allocated one, a chunk size might be erased

Dynamic memory allocations

Check the memory sections of processes containing memory dynamically allocated during run time

☐

Data

☐

Text

☐

Stack

☐

Heap

Exceptions in RTOS

Check all the true claims listed below about exceptions, interrupts and processes in RTOS

☐

Urgent processes can have a higher priority than interrupts

☐

Moving part of an Interrupt Service Routine as a regular process makes the system slower

☐

Urgent processes can preempt Interrupt Service Routines

☐

Scheduling policies of RTOS always select first Interrupt Service Routines

☐ Exceptions in real-time systems are sometimes ignored to execute urgent processes first

Exiting a blocked state

We assume that a process *p* is currently in "blocked" state. Which reasons might apply for the process *p* to go back to "Runnable" state?

☐ Process *p* has been killed
☐

The currently running process has used all its quantum of time

☐

The currently running process has called _exit()

☐

The administrator has started a poweroff procedure

☐

An IO operation started by *p* has completed

☐

A signal not ignored by *p* has been sent to *p*

Fork: true or false?

Select all the true claims about *fork*() in Linux

☐

fork() is a syscall

☐ *fork*() creates a new process
☐

After *fork*() has completed, child and parent processes share the same address space

☐

After *fork*() has completed, the memory content of the child and parent process are equal

☐

After *fork*(), the child process runs first

☐

fork() returns the *pid* of the parent process to the child process

Handling pages

Linux maintains a list of allocated memory on the heap for each:

☐

process

☐

page

☐

thread

☐

kernel module

Hard disk drives

Hard disk drives may have an embedded request scheduling policy (e.g. SSTF). Check all the true claims.

☐

Embedding a scheduling policy into a disk discharges the Operating System from doing it

☐

Embedding a scheduling policy into a disk avoids having a disk driver in the Operating System

☐

Embedding a scheduling policy into a disk allows the Operating System to have its own request scheduler for the disk

☐

A DMA engine must be used for the scheduling policy to be activated

☐

File caching by the Operating System makes the use of an embedded scheduling policy useless

Kernel

Check all the true claims.

☐

Kernel is the central program of an Operating System

☐

When a computer starts, the kernel of the Operating System is the first code to be executed

☐

The kernel first starts before system services can be started

☐

Kernel remains in memory until the computer is stopped

Message queues : Linux code

The following code is taken from the ipc/msg.c file of the Linux Kernel 4.19.225. Check all the true claims that follow.

```
/**
 * newque - Create a new msg queue
 * @ns: namespace
 * @params: ptr to the structure that contains the key and msgflg
```

```
 *
 * Called with msg_ids.rwsem held (writer)
 */
static int newque(struct ipc_namespace *ns, struct ipc_params *params)
{
	struct msg_queue *msq;
	int retval;
	key_t key = params->key;
	int msgflg = params->flg;

	msq = kvmalloc(sizeof(*msq), GFP_KERNEL);
	if (unlikely(!msq))
		return -ENOMEM;

	msq->q_perm.mode = msgflg & S_IRWXUGO;
	msq->q_perm.key = key;

	msq->q_perm.security = NULL;
	retval = security_msg_queue_alloc(&msq->q_perm);
	if (retval) {
		kvfree(msq);
		return retval;
	}

	msq->q_stime = msq->q_rtime = 0;
	msq->q_ctime = ktime_get_real_seconds();
	msq->q_cbytes = msq->q_qnum = 0;
	msq->q_qbytes = ns->msg_ctlmnb;
	msq->q_lspid = msq->q_lrpid = NULL;
	INIT_LIST_HEAD(&msq->q_messages);
	INIT_LIST_HEAD(&msq->q_receivers);
	INIT_LIST_HEAD(&msq->q_senders);

	/* ipc_addid() locks msq upon success. */
	retval = ipc_addid(&msg_ids(ns), &msq->q_perm, ns->msg_ctlmni);
	if (retval < 0) {
		ipc_rcu_putref(&msq->q_perm, msg_rcu_free);
		return retval;
	}

	ipc_unlock_object(&msq->q_perm);
	rcu_read_unlock();

	return msq->q_perm.id;
}
```

The function creates a new message

☐

The function returns a pointer to a message queue

☐

This function returns an error if no "id" is available

☐

The function allocates a new structure but has to disallocate it in case of error

☐

The created object can handle receivers

Preemption points

Which of the following assertions on preemption points are true?

☐ Preemption points make the system run faster.

☐

When running, the kernel can be preempted only at preemption points.

☐

When a preemption point is reached, the OS scheduler is called.

☐

Preemption points can be reached only if interrupts have been disabled

☐

Real-time tasks are executed only after a preemption point has been reached

Random device

$ ls -al /dev/random

crw-rw-rw- 1 root root 1, 8 Oct 21 13:07 /dev/random

- [ ] /dev/random corresponds to any kind of devices
- [ ] This device has no major number
- [ ] The minor number of this device is "8"
- [ ] This device is a character device

Synchronization

A process that is perpetually denied necessary resources is in a _____ state

- ○ starvation
- ○ deadlock
- ○ cache
- ○ progress

Threading and pointers

Carefully read the (fantastic) following code, and check the true claims below.

```
#include <stdlib.h>
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>

long b;

void *t1(void *arg) {
```

```
  int a = 10;
  b = &a;
  sleep(random() % 5);
  printf("End of t1; a=%d\n", *((int *)b));
}

void *t2(void *arg) {
    sleep(random() % 5);
    int *myP = b;
    myP[0] = 5;
    myP[100] = 6;
    printf("End of t2\n");
}



int main(void) {
  int i;
  pthread_t tid_h;
  pthread_t tid_ny;

  pthread_create(&tid_h, NULL, t1, NULL);
  pthread_create(&tid_ny, NULL, t2, NULL);

  pthread_join(tid_h, NULL);
  pthread_join(tid_ny, NULL);

 return (0);
}
```

- ☐ This program may provoke a segmentation fault
- ☐ This program always provokes a segmentation fault
- ☐ This program may smash the stack of thread tid_h without the OS noticing it
- ☐ If tid_ny totally executes first, then no error is produced

Time slot and process states

Assuming a round robin scheduling with time slots, after a process p has used all its quantum of time, p switches from the current state to?

- ○ Blocked?
- ○ Runnable?
- ○ Terminated?

USB keys

Check all the true claims among the following ones.

- ☐ A USB key is a block device
- ☐ A USB key is generally a read and write device
- ☐ A USB key has a mechanical latency
- ☐ A USB key must be ejected before files can be written to it
- ☐ Caching makes writing operations to the memory cells of USB keys faster

Creation of processes

Assume we compile this program and we start it: a new process "p" is created in the Operating System. How many new processes are started by "p" and its descendants?

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

int main(void) {
  pid_t ret;
  ret = fork();
```

```
  ret = fork();
  if (ret == 0) {
    ret = fork();
  }
}
```

Answer

Hard disks on my computer

How many hard disks do I have on my computer?

```
$ ls -al /dev/sd*
brw-rw---- 1 root disk 8,  0 Oct 21 13:07 /dev/sda
brw-rw---- 1 root disk 8,  1 Oct 21 13:07 /dev/sda1
brw-rw---- 1 root disk 8, 16 Oct 21 13:07 /dev/sdb
brw-rw---- 1 root disk 8, 17 Oct 21 13:07 /dev/sdb1
brw-rw---- 1 root disk 8, 32 Oct 21 13:07 /dev/sdc
brw-rw---- 1 root disk 8, 33 Oct 21 13:07 /dev/sdc1
```

Answer

How many processes?

How many processes are created by this shell command?

```
$ cat /etc/os-release | grep VERSION > test
```

Answer

Support for several processors or cores

From this command, does your kernel support several processors?

```
$ uname -a
Linux box 4.2.9 #2 SMP Fri Sep 8 14:42:43 UTC 2017 i686 GNU/Linux
```

- True
- False

Envoyer