helion.pl
KSIĘGARNIA INTERNETOWA · WYDAWNICTWO

IDŹ DO

PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

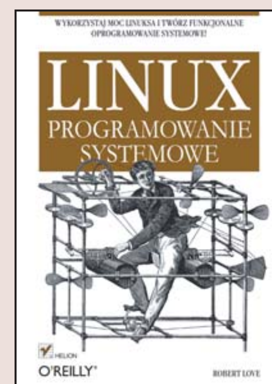
KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

Linux. Programowanie systemowe



Autor: Robert Love
Tłumaczenie: Jacek Janusz
ISBN: 978-83-246-1497-4
Tytuł oryginału: Linux System Programming: Talking Directly to the Kernel and C Library
Format: 168x237, stron: 400

Wykorzystaj moc Linuksa i twórz funkcjonalne oprogramowanie systemowe!

- Jak zarządzać plikowymi operacjami wejścia i wyjścia?
- Jak zablokować fragmenty przestrzeni adresowej?
- Jak sterować działaniem interfejsu odpytywania zdarzeń?

Dzisiaj systemu Linux nie musimy już nikomu przedstawiać, dzięki swojej funkcjonalności i uniwersalności stał się niezwykle popularny i szeroko wykorzystywany. Działa wszędzie ? poczynając od najmniejszych telefonów komórkowych, a na potężnych superkomputerach kończąc. Z Linuksa korzystają agencje wywiadowcze i wojsko, jego niezawodność doceniły również banki i instytucje finansowe. Oprogramowanie z przestrzeni użytkownika w systemie Linux może być uruchamiane na wszystkich platformach, na których poprawnie działa kod jądra.

Czytając książkę „Linux. Programowanie systemowe", dowiesz się, jak utworzyć oprogramowanie, które jest niskopoziomowym kodem, komunikującym się bezpośrednio z jądrem oraz głównymi bibliotekami systemowymi. Opisany został tu sposób działania standardowych i zaawansowanych interfejsów zdefiniowanych w Linuksie. Po lekturze napiszesz inteligentniejszy i szybszy kod, który działa we wszystkich dystrybucjach Linuksa oraz na wszystkich rodzajach sprzętu. Nauczysz się budować poprawne oprogramowanie i maksymalnie je wykorzystywać.

- Programowanie systemowe
- Biblioteka języka C
- Kompilator języka C
- Interfejs odpytywania zdarzeń
- Zarządzanie procesami i pamięcią
- Użytkownicy i grupy
- Ograniczenia zasobów systemowych
- Zarządzanie plikami i katalogami
- Identyfikatory sygnałów
- Struktury danych reprezentujące czas
- Konwersje czasu

Poznaj i ujarzmij potęgę Linuksa!

Wydawnictwo Helion
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści

Przedmowa 7

Wstęp 9

1. Wprowadzenie — podstawowe pojęcia 15
Programowanie systemowe 15
API i ABI 18
Standardy 20
Pojęcia dotyczące programowania w Linuksie 23
Początek programowania systemowego 36

2. Plikowe operacje wejścia i wyjścia 37
Otwieranie plików 38
Czytanie z pliku przy użyciu funkcji read() 43
Pisanie za pomocą funkcji write() 47
Zsynchronizowane operacje wejścia i wyjścia 51
Bezpośrednie operacje wejścia i wyjścia 55
Zamykanie plików 56
Szukanie za pomocą funkcji lseek() 57
Odczyty i zapisy pozycyjne 59
Obcinanie plików 60
Zwielokrotnione operacje wejścia i wyjścia 61
Organizacja wewnętrzna jądra 72
Zakończenie 76

3. Buforowane operacje wejścia i wyjścia 77
Operacje wejścia i wyjścia, buforowane w przestrzeni użytkownika 77
Typowe operacje wejścia i wyjścia 79
Otwieranie plików 80

Otwieranie strumienia poprzez deskryptor pliku 81
Zamykanie strumieni 82
Czytanie ze strumienia 83
Pisanie do strumienia 86
Przykładowy program używający buforowanych operacji wejścia i wyjścia 88
Szukanie w strumieniu 89
Opróżnianie strumienia 91
Błędy i koniec pliku 92
Otrzymywanie skojarzonego deskryptora pliku 93
Parametry buforowania 93
Bezpieczeństwo wątków 95
Krytyczna analiza biblioteki typowych operacji wejścia i wyjścia 97
Zakończenie 98

4. Zaawansowane operacje plikowe wejścia i wyjścia 99
Rozproszone operacje wejścia i wyjścia 100
Interfejs odpytywania zdarzeń 105
Odwzorowywanie plików w pamięci 110
Porady dla standardowych operacji plikowych wejścia i wyjścia 123
Operacje zsynchronizowane, synchroniczne i asynchroniczne 126
Zarządcy operacji wejścia i wyjścia oraz wydajność operacji wejścia i wyjścia 129
Zakończenie 141

5. Zarządzanie procesami 143
Identyfikator procesu 143
Uruchamianie nowego procesu 146
Zakończenie procesu 153
Oczekiwanie na zakończone procesy potomka 156
Użytkownicy i grupy 166
Grupy sesji i procesów 171
Demony 176
Zakończenie 178

6. Zaawansowane zarządzanie procesami 179
Szeregowanie procesów 179
Udostępnianie czasu procesora 183
Priorytety procesu 186
Wiązanie procesów do konkretnego procesora 189
Systemy czasu rzeczywistego 192
Ograniczenia zasobów systemowych 206

7. Zarządzanie plikami i katalogami 213
Pliki i ich metadane 213
Katalogi 228
Dowiązania 240
Kopiowanie i przenoszenie plików 245
Węzły urządzeń 248
Komunikacja poza kolejką 249
Śledzenie zdarzeń związanych z plikami 251

8. Zarządzanie pamięcią 261
Przestrzeń adresowa procesu 261
Przydzielanie pamięci dynamicznej 263
Zarządzanie segmentem danych 273
Anonimowe odwzorowania w pamięci 274
Zaawansowane operacje przydziału pamięci 278
Uruchamianie programów, używających systemu przydzielania pamięci 281
Przydziały pamięci wykorzystujące stos 282
Wybór mechanizmu przydzielania pamięci 286
Operacje na pamięci 287
Blokowanie pamięci 291
Przydział oportunistyczny 295

9. Sygnały 297
Koncepcja sygnałów 298
Podstawowe zarządzanie sygnałami 304
Wysyłanie sygnału 309
Współużywalność 311
Zbiory sygnałów 314
Blokowanie sygnałów 315
Zaawansowane zarządzanie sygnałami 316
Wysyłanie sygnału z wykorzystaniem pola użytkowego 324
Zakończenie 325

10. Czas 327
Struktury danych reprezentujące czas 329
Zegary POSIX 332
Pobieranie aktualnego czasu 334
Ustawianie aktualnego czasu 337
Konwersje czasu 338

Dostrajanie zegara systemowego 340
Stan uśpienia i oczekiwania 343
Liczniki 349

A Rozszerzenia kompilatora GCC dla języka C 357

B Bibliografia 369

Skorowidz 373

ROZDZIAŁ 8.
Zarządzanie pamięcią

Pamięć należy do najbardziej podstawowych, a jednocześnie najważniejszych zasobów dostępnych dla procesu. W rozdziale tym omówione zostaną tematy związane z zarządzaniem nią: przydzielanie, modyfikowanie i w końcu zwalnianie pamięci.

Słowo *przydzielanie* — powszechnie używany termin, określający czynność udostępniania obszaru pamięci — wprowadza w błąd, ponieważ wywołuje obraz wydzielania deficytowego zasobu, dla którego wielkość żądań przewyższa wielkość zapasów. Na pewno wielu użytkowników wolałoby mieć więcej dostępnej pamięci. Dla nowoczesnych systemów problem nie polega jednak na rozdzielaniu zbyt małych zapasów dla zbyt wielu użytkowników, lecz właściwym używaniu i monitorowaniu danego zasobu.

W tym rozdziale przeanalizowane zostaną wszystkie metody przydzielania pamięci dla różnych obszarów programu, jednocześnie z ukazaniem ich zalet i wad. Przedstawimy również pewne sposoby, pozwalające na ustawianie i modyfikację zawartości dowolnych obszarów pamięci, a także wyjaśnimy, w jaki sposób należy zablokować dane w pamięci, aby w programach nie trzeba było oczekiwać na operacje jądra, które zajmowałoby się przerzucaniem danych z obszaru wymiany.

Przestrzeń adresowa procesu

Linux, podobnie jak inne nowoczesne systemy operacyjne, wirtualizuje swój fizyczny zasób pamięci. Procesy nie adresują bezpośrednio pamięci fizycznej. Zamiast tego jądro wiąże każdy proces z unikalną *wirtualną przestrzenią adresową*. Jest ona liniowa, a jej adresacja rozpoczyna się od zera i wzrasta do pewnej granicznej wartości maksymalnej.

Strony i stronicowanie

Wirtualna przestrzeń adresowa składa się ze *stron*. Architektura systemu oraz rodzaj maszyny determinują rozmiar strony, który jest stały: typowymi wartościami są na przykład 4 kB (dla systemów 32-bitowych) oraz 8 kB (dla systemów 64-bitowych)[1]. Strony są albo prawidłowe,

[1] Czasami systemy wspierają rozmiary stron, które mieszczą się w pewnym zakresie. Z tego powodu rozmiar strony nie jest częścią interfejsu binarnego aplikacji (ABI). Aplikacje muszą w sposób programowy uzyskać rozmiar strony w czasie wykonania. Zostało to opisane w rozdziale 4. i będzie jednym z tematów poruszonych w tym rozdziale.

albo nieprawidłowe. *Strona prawidłowa* (ang. *valid page*) związana jest ze stroną w pamięci fizycznej lub jakąś dodatkową pamięcią pomocniczą, np. partycją wymiany lub plikiem na dysku. *Strona nieprawidłowa* (ang. *invalid page*) nie jest z niczym związana i reprezentuje nieużywany i nieprzydzielony obszar przestrzeni adresowej. Dostęp do takiej strony spowoduje błąd segmentacji. Przestrzeń adresowa nie musi być koniecznie ciągła. Mimo że jest ona adresowana w sposób liniowy, zawiera jednak mnóstwo przerw, nieposiadających adresacji.

Program nie może użyć strony, która znajduje się w dodatkowej pamięci pomocniczej zamiast w fizycznej. Będzie to możliwe dopiero wtedy, gdy zostanie ona połączona ze stroną w pamięci fizycznej. Gdy proces próbuje uzyskać dostęp do adresu z takiej strony, układ zarządzania pamięcią (MMU) generuje *błąd strony* (ang. *page fault*). Wówczas wkracza do akcji jądro, w sposób niewidoczny *przerzucając* żądaną stronę z pamięci pomocniczej do pamięci fizycznej. Ponieważ istnieje dużo więcej pamięci wirtualnej niż rzeczywistej (nawet w przypadku systemów z pojedynczą wirtualną przestrzenią adresową!), jądro również przez cały czas *wyrzuca* strony z pamięci fizycznej do dodatkowej pamięci pomocniczej, aby zrobić miejsce na nowe strony, przerzucane w drugim kierunku. Jądro przystępuje do wyrzucania danych, dla których istnieje najmniejsze prawdopodobieństwo, iż będą użyte w najbliższej przyszłości. Dzięki temu następuje poprawa wydajności.

Współdzielenie i kopiowanie podczas zapisu

Wiele stron w pamięci wirtualnej, a nawet w różnych wirtualnych przestrzeniach adresowych należących do oddzielnych procesów, może być odwzorowanych na pojedynczą stronę fizyczną. Pozwala to różnym wirtualnym przestrzeniom adresowym na *współdzielenie* danych w pamięci fizycznej. Współdzielone dane mogą posiadać uprawnienia tylko do odczytu lub zarówno do odczytu, jak i zapisu.

Gdy proces przeprowadza operację zapisu do współdzielonej strony, posiadającej uprawnienia do wykonania tej czynności, mogą zaistnieć jedna lub dwie sytuacje. Najprostsza wersja polega na tym, że jądro zezwoli na wykonanie zapisu i wówczas wszystkie procesy współdzielące daną stronę, będą mogły „zobaczyć" wyniki tej operacji. Zezwolenie wielu procesom na czytanie lub zapis do współdzielonej strony wymaga zazwyczaj zapewnienia pewnego poziomu współpracy i synchronizacji między nimi.

Inaczej jest jednak, gdy układ zarządzania pamięcią przechwyci operację zapisu i wygeneruje wyjątek; w odpowiedzi, jądro w sposób niewidoczny stworzy nową kopię strony dla procesu zapisującego i zezwoli dla niej na kontynuowanie zapisu. To rozwiązanie zwane jest *kopiowaniem podczas zapisu* (ang. *copy-on-write*, w skrócie *COW*)[2]. Proces faktycznie posiada uprawnienia do odczytu dla współdzielonych danych, co przyczynia się do oszczędzania pamięci. Gdy proces chce zapisać do współdzielonej strony, otrzymuje wówczas na bieżąco unikalną jej kopię. Dzięki temu jądro może działać w taki sposób, jak gdyby proces zawsze posiadał swoją własną kopię strony. Ponieważ kopiowanie podczas zapisu jest zaimplementowane dla każdej strony z osobna, dlatego też duży plik może zostać efektywnie udostępniony wielu procesom, którym zostaną przydzielone unikalne strony fizyczne tylko wówczas, gdy będą chciały coś w nich zapisywać.

---

[2] W rozdziale 5. napisano, że funkcja `fork()` używa metody kopiowania podczas zapisu, aby powielić i udostępnić przestrzeń adresową rodzica tworzonemu procesowi potomnemu.

Regiony pamięci

Jądro rozmieszcza strony w blokach, które posiadają pewne wspólne cechy charakterystyczne, takie jak uprawnienia dostępu. Bloki te zwane są *regionami pamięci* (ang. *memory regions*), *segmentami* (ang. *segments*) lub *odwzorowaniami* (ang. *mappings*). Pewne rodzaje regionów pamięci mogą istnieć w każdym procesie:

- *Segment tekstu* zawiera kod programu dla danego procesu, literały łańcuchowe, stałe oraz inne dane tylko do odczytu. W systemie Linux segment ten posiada uprawnienia tylko do odczytu i jest odwzorowany bezpośrednio na plik obiektowy (program wykonywalny lub bibliotekę).
- *Segment stosu*, jak sama nazwa wskazuje, zawiera stos wykonania procesu. Segment stosu rozrasta się i maleje w sposób dynamiczny, zgodnie ze zmianami struktury stosu. Stos wykonania zawiera lokalne zmienne oraz dane zwracane z funkcji.
- *Segment danych* (lub *sterta*) zawiera dynamiczną pamięć procesu. Do segmentu tego można zapisaywać, a jego rozmiar się zmienia. Sterta jest zwracana przy użyciu funkcji `malloc()` (omówionej w następnym podrozdziale).
- *Segment bss*[3] zawiera niezainicjalizowane zmienne globalne. Zmienne te mają specjalne wartości (w zasadzie same zera), zgodnie ze standardem języka C. Linux optymalizuje je przy użyciu dwóch metod. Po pierwsze, ponieważ segment bss przeznaczony jest dla przechowywania niezainicjalizowanych danych, więc linker (*ld*) w rzeczywistości nie zapisuje specjalnych wartości do pliku obiektowego. Powoduje to zmniejszenie rozmiaru pliku binarnego. Po drugie, gdy segment ten zostaje załadowany do pamięci, jądro po prostu odwzorowuje go w trybie kopiowania podczas zapisu na stronę zawierającą same zera, co efektywnie ustawia domyślne wartości w zmiennych.
- Większość przestrzeni adresowej zajmuje grupa *plików odwzorowanych*, takich jak sam program wykonywalny, różne biblioteki — między innymi dla języka C, a także pliki z danymi. Ścieżka */proc/self/maps* lub wynik działania programu *pmap* są bardzo dobrymi przykładami plików odwzorowanych w procesie.

Rozdział ten omawia interfejsy, które są udostępnione przez system Linux, aby otrzymywać i zwalniać obszary pamięci, tworzyć i usuwać nowe odwzorowania oraz wykonywać inne czynności związane z pamięcią.

Przydzielanie pamięci dynamicznej

Pamięć zawiera także automatyczne i statyczne zmienne, lecz podstawą działania każdego systemu, który nią zarządza, jest przydzielanie, używanie, a w końcu zwalnianie *pamięci dynamicznej*. Pamięć dynamiczna przydzielana jest w czasie działania programu, a nie kompilacji, a jej rozmiary mogą być nieznane do momentu rozpoczęcia samego procesu przydzielania. Dla projektanta jest ona użyteczna w momencie, gdy zmienia się ilość pamięci, którą potrzebuje tworzony program lub też zmienny jest czas, w ciągu którego będzie ona używana, a dodatkowo wielkości te nie są znane przed uruchomieniem aplikacji. Na przykład, można zaimplementować przechowywanie w pamięci zawartości jakiegoś pliku lub danych wczytywanych

[3] Nazwa to relikt historii — jest to skrót od słów: *blok rozpoczęty od symbolu* (ang. *block started by symbol*).

z klawiatury. Ponieważ wielkość takiego pliku jest nieznana, a użytkownik może wprowadzić dowolną liczbę znaków z klawiatury, rozmiar bufora musi być zmienny, by programista dynamicznie go zwiększał, gdy danych zacznie przybywać.

Żadne zmienne języka C nie są zapisywane w pamięci dynamicznej. Na przykład, język C nie udostępnia mechanizmu, który pozwala na odczytanie struktury `pirate_ship` znajdującej się w takiej pamięci. Zamiast tego istnieje metoda pozwalająca na przydzielenie takiej ilości pamięci dynamicznej, która wystarczy, aby przechować w niej strukturę `pirate_ship`. Programista następnie używa tej pamięci poprzez posługiwanie się wskaźnikiem do niej — w tym przypadku, stosując wskaźnik `struct pirate_ship *`.

Klasycznym interfejsem języka C, pozwalającym na otrzymanie pamięci dynamicznej, jest funkcja `malloc()`:

```
#include <stdlib.h>

void * malloc (size_t size);
```

Poprawne jej wywołanie przydziela obszar pamięci, którego wielkość (w bajtach) określona jest w parametrze `size`. Funkcja zwraca wskaźnik do początku nowo przydzielonego regionu. Zawartość pamięci jest niezdefiniowana i nie należy oczekiwać, że będzie zawierać same zera. W przypadku błędu, funkcja `malloc()` zwraca `NULL` oraz ustawia zmienną `errno` na `ENOMEM`.

Użycie funkcji `malloc()` jest raczej proste, tak jak w przypadku poniższego przykładu przydzielającego określoną liczbę bajtów:

```
char *p;

/* przydziel 2 kB! */
p = malloc (2048);
if (!p)
   perror ("malloc");
```

Nieskomplikowany jest również kolejny przykład, przydzielający pamięć dla struktury:

```
struct treasure_map *map;

/*
* przydziel wystarczająco dużo pamięci, aby przechować strukturę treasure_map,
* a następnie przypisz adres tego obszaru do wskaźnika 'map'
*/
map = malloc (sizeof (struct treasure_map));
if (!map)
   perror ("malloc");
```

Język C automatycznie rzutuje wskaźniki typu `void` na dowolny typ, występujący podczas operacji przypisania. Dlatego też w przypadku powyższych przykładów, nie jest konieczne rzutowanie typu zwracanej wartości funkcji `malloc()` na typ l-wartości, używanej podczas operacji przypisania. Język programowania C++ nie wykonuje jednak automatycznego rzutowania wskaźnika `void`. Zgodnie z tym użytkownicy języka C++ muszą rzutować wyniki wywołania funkcji `malloc()`, tak jak pokazano to w poniższym przykładzie:

```
char *name;

/* przydziel 512 bajtów */
name = (char *) malloc (512);
if (!name)
   perror ("malloc");
```

Niektórzy programiści języka C preferują wykonywanie rzutowania wyników dowolnej funkcji, która zwraca wskaźnik `void`. Dotyczy to również funkcji `malloc()`. Ten styl programowania jest jednak niepewny z dwóch powodów. Po pierwsze, może on spowodować pominięcie błędu w przypadku, gdy wartość zwracana z funkcji kiedykolwiek ulegnie zmianie i nie będzie równa wskaźnikowi `void`. Po drugie, takie rzutowanie ukrywa błędy również w przypadku, gdy funkcja jest niewłaściwie zadeklarowana[4]. Pierwszy z tych powodów nie jest przyczyną powstawania problemów podczas użycia funkcji `malloc()`, natomiast drugi może już ich przysparzać.

Ponieważ funkcja `malloc()` może zwrócić wartość `NULL`, dlatego też jest szczególnie ważne, aby projektanci oprogramowania *zawsze* sprawdzali i obsługiwali przypadki błędów. W wielu programach funkcja `malloc()` nie jest używana bezpośrednio, lecz istnieje dla niej stworzony interfejs programowy (*wrapper*), który wyprowadza komunikat błędu i przerywa działanie programu, gdy zwraca ona wartość `NULL`. Zgodnie z konwencja nazewniczą, ten ogólny interfejs programowy zwany jest przez projektantów `xmalloc()`:

```
/* działa jak malloc(), lecz kończy wykonywanie programu w przypadku niepowodzenia */
void * xmalloc (size_t size)
{
   void *p;

   p = malloc (size);
   if (!p)
   {
      perror ("xmalloc");
      exit (EXIT_FAILURE);
   }

   return p;
}
```

Przydzielanie pamięci dla tablic

Dynamiczne przydzielanie pamięci może być skomplikowane, jeśli rozmiar danych, przekazany w parametrze `size`, jest również zmienny. Jednym z tego typu przykładów jest dynamiczne przydzielanie pamięci dla tablic, których rozmiar jednego elementu może być stały, lecz liczba alokowanych elementów jest zmienna.

Aby uprościć wykonywanie tej czynności, język C udostępnia funkcję `calloc()`:

```
#include <stdlib.h>

void * calloc (size_t nr, size_t size);
```

Poprawne wywołanie funkcji `calloc()` zwraca wskaźnik do bloku pamięci o wielkości wystarczającej do przechowania tablicy o liczbie elementów określonej w parametrze `nr`. Każdy z elementów posiada rozmiar `size`. Zgodnie z tym ilość pamięci, przydzielona w przypadku użycia zarówno funkcji `malloc()`, jak i `calloc()`, jest taka sama (obie te funkcje mogą zwrócić więcej pamięci, niż jest to wymagane, lecz nigdy mniej):

```
int *x, *y;

x = malloc (50 * sizeof (int));
```

[4] Funkcje niezadeklarowane zwracają domyślnie wartości o typie `int`. Rzutowanie liczby całkowitej na wskaźnik nie jest wykonywane automatycznie i powoduje powstanie ostrzeżenia podczas kompilacji programu. Użycie rzutowania typów nie pozwala na generowanie takiego ostrzeżenia.

```
if (!x)
{
   perror ("malloc");
   return -1;
}

y = calloc (50, sizeof (int));
if (!y)
{
   perror ("calloc");
   return -1;
}
```

Zachowanie powyższych dwóch funkcji nie jest jednak identyczne. W przeciwieństwie do funkcji `malloc()`, która nie zapewnia, jaka będzie zawartość przydzielonej pamięci, funkcja `calloc()` zeruje wszystkie bajty w zwróconym obszarze pamięci. Dlatego też każdy z 50 elementów w tablicy liczb całkowitych `y` posiada wartość `0`, natomiast wartości elementów tablicy `x` są niezdefiniowane. Dopóki w programie nie ma potrzeby natychmiastowego zainicjalizowania wszystkich 50 wartości, programiści powinni używać funkcji `calloc()`, aby zapewnić, że elementy tablicy nie są wypełnione przypadkowymi danymi. Należy zauważyć, że zero binarne może być różne od zera występującego w liczbie zmiennoprzecinkowej!

Użytkownicy często chcą „wyzerować" pamięć dynamiczną, nawet wówczas, gdy nie używają tablic. W dalszej części tego rozdziału poddana analizie zostanie funkcja `memset()`, która dostarcza interfejsu pozwalającego na ustawienie wartości dla dowolnego bajta w obszarze pamięci. Funkcja `calloc()` wykonuje jednak tę operację szybciej, gdyż jądro może od razu udostępnić obszar pamięci, który wypełniony jest już zerami.

W przypadku błędu funkcja `calloc()`, podobnie jak `malloc()`, zwraca `−1` oraz ustawia zmienną `errno` na wartość `ENOMEM`.

Dlaczego w standardach nie zdefiniowano nigdy funkcji „przydziel i wyzeruj", różnej od `calloc()`, pozostaje tajemnicą. Projektanci mogą jednak w prosty sposób zdefiniować swój własny interfejs:

```
/* działa tak samo jak funkcja malloc(), lecz przydzielona pamięć zostaje wypełniona zerami */
void * malloc0 (size_t size)
{
   return calloc (1, size);
}
```

Można bez kłopotu połączyć funkcję `malloc0()` z poprzednio przedstawioną funkcją `xmalloc()`:

```
/* działa podobnie jak malloc(), lecz wypełnia pamięć zerami i przerywa działanie programu w przypadku błędu */
void * xmalloc0 (size_t size)
{
   void *p;

   p = calloc (1, size);
   if (!p)
   {
      perror ("xmalloc0");
      exit (EXIT_FAILURE);
   }

   return p;
}
```

Zmiana wielkości obszaru przydzielonej pamięci

Język C dostarcza interfejsu pozwalającego na zmianę wielkości (zmniejszenie lub powiększenie) istniejącego obszaru przydzielonej pamięci:

```
#include <stdlib.h>

void * realloc (void *ptr, size_t size);
```

Poprawne wywołanie funkcji `realloc()` zmienia rozmiar regionu pamięci, wskazywanego przez `ptr`, na nową wartość, której wielkość podana jest w parametrze `size` i wyrażona w bajtach. Funkcja zwraca wskaźnik do obszaru pamięci posiadającego nowy rozmiar. Wskaźnik ten nie musi być równy wartości parametru `ptr`, który był używany w funkcji podczas wykonywania operacji powiększenia rozmiaru obszaru. Jeśli funkcja `realloc()` nie potrafi powiększyć istniejącego obszaru pamięci poprzez zmianę rozmiaru dla wcześniej przydzielonego miejsca, wówczas może ona zarezerwować pamięć dla nowego regionu pamięci o rozmiarze `size`, wyrażonym w bajtach, skopiować zawartość poprzedniego regionu w nowe miejsce, a następnie zwolnić niepotrzebny już obszar źródłowy. W przypadku każdej operacji zachowana zostaje zawartość dla takiej wielkości obszaru pamięci, która równa jest mniejszej wartości z dwóch rozmiarów: poprzedniego i aktualnego. Z powodu ewentualnego istnienia operacji kopiowania, wywołanie funkcji `realloc()`, które wykonuje powiększenie obszaru pamięci, może być stosunkowo kosztowne.

Jeśli `size` wynosi zero, rezultat jest taki sam jak w przypadku wywołania funkcji `free()` z parametrem `ptr`.

Jeśli parametr `ptr` jest równy `NULL`, wówczas rezultat wykonania operacji jest taki sam jak dla oryginalnej funkcji `malloc()`. Jeśli wskaźnik `ptr` jest różny od `NULL`, powinien zostać zwrócony przez wcześniejsze wykonanie jednej z funkcji `malloc()`, `calloc()` lub `realloc()`.

W przypadku błędu, funkcja `realloc()` zwraca `NULL` oraz ustawia zmienną `errno` na wartość `ENOMEM`. Stan obszaru pamięci, wskazywanego przez parametr `ptr`, pozostaje niezmieniony.

Rozważmy przykład programu, który zmniejsza obszar pamięci. Najpierw należy użyć funkcji `calloc()`, która przydzieli wystarczającą ilość pamięci, aby zapamiętać w niej dwuelementową tablicę struktur `map`:

```
struct map *p;

/* przydziel pamięć na dwie struktury 'map' */
p = calloc (2, sizeof (struct map));
if (!p)
{
   perror ("calloc");
   return -1;
}

/* w tym momencie można używać p[0] i p[1] ... */
```

Załóżmy, że jeden ze skarbów został już znaleziony, dlatego też nie ma potrzeby użycia drugiej mapy. Podjęto decyzję, że rozmiar obszaru pamięci zostanie zmieniony, a połowa przydzielonego wcześniej regionu zostanie zwrócona do systemu (operacja ta nie byłaby właściwie zbyt potrzebna, chyba że rozmiar struktury `map` byłby bardzo duży, a program rezerwowałby dla niej pamięć przez dłuższy czas):

```
struct map *r;

/* obecnie wymagana jest tylko pamięć dla jednej mapy */
r = realloc (p, sizeof (struct map));
if (!r)
{
   /* należy zauważyć, że 'p' jest wciąż poprawnym wskaźnikiem! */
   perror ("realloc");
   return -1;
}

/* tu można już używać wskaźnika 'r'... */

free (r);
```

W powyższym przykładzie, po wywołaniu funkcji `realloc()` zostaje zachowany element `p[0]`. Jakiekolwiek dane, które przedtem znajdowały się w tym elemencie, będą obecne również teraz. Jeśli wywołanie funkcji się nie powiedzie, należy zwrócić uwagę na to, że wskaźnik `p` nie zostanie zmieniony i stąd też będzie wciąż poprawny. Można go ciągle używać i w końcu należy go zwolnić. Jeśli wywołanie funkcji się powiedzie, należy zignorować wskaźnik `p` i zamiast niego użyć `r` (który jest przypuszczalnie równy `p`, gdyż najprawdopodobniej nastąpiła zmiana rozmiaru aktualnie przydzielonego obszaru). Obecnie programista odpowiedzialny będzie za zwolnienie pamięci dla wskaźnika `r`, gdy tylko przestanie on być potrzebny.

Zwalnianie pamięci dynamicznej

W przeciwieństwie do obszarów pamięci przydzielonych automatycznie, które same zostają zwolnione, gdy następuje przesunięcie wskaźnika stosu, dynamicznie przydzielone regiony pamięci pozostają trwałą częścią przestrzeni adresowej procesu, dopóki nie zostaną ręcznie zwolnione. Dlatego też programista odpowiedzialny jest za zwolnienie do systemu dynamicznie przydzielonej pamięci (oba rodzaje przydzielonej pamięci — statyczna i dynamiczna — zostają zwolnione, gdy cały proces kończy swoje działanie).

Pamięć, przydzielona za pomocą funkcji `malloc()`, `calloc()` lub `realloc()`, musi zostać zwolniona do systemu, jeśli nie jest już więcej używana. W tym celu stosuje się funkcję `free()`:

```
#include <stdlib.h>

void free (void *ptr);
```

Wywołanie funkcji `free()` zwalnia pamięć, wskazywaną przez wskaźnik `ptr`. Parametr `ptr` powinien być zainicjalizowany przez wartość zwróconą wcześniej przez funkcję `malloc()`, `calloc()` lub `realloc()`. Oznacza to, że nie można użyć funkcji `free()`, aby zwolnić fragment obszaru pamięci — na przykład połowę — poprzez przekazanie do niej parametru wskazującego na środek wcześniej przydzielonego obszaru.

Wskaźnik `ptr` może być równy `NULL`, co powoduje, że funkcja `free()` od razu wraca do procesu wywołującego. Dlatego też niepotrzebne jest sprawdzanie wskaźnika `ptr` przed wywołaniem funkcji `free()`.

Oto przykład użycia funkcji `free()`:

```
void print_chars (int n, char c)
{
   int i;

   for (i = 0; i < n; i++)
```

```
    {
       char *s;
       int j;

       /*
       * Przydziel i wyzeruj tablicę znaków o liczbie elementów równej i+2.
       * Należy zauważyć, że wywołanie 'sizeof (char)' zwraca zawsze wartość 1.
       */
       s = calloc (i + 2, 1);
       if (!s)
       {
          perror ("calloc");
          break;
       }

       for (j = 0; j < i + 1; j++)
          s[j] = c;

       printf ("%s\n", s);

       /* Wszystko zrobione, obecnie należy zwolnić pamięć. */
       free (s);
    }
 }
```

Powyższy przykład przydziela pamięć dla `n` tablic typu `char`, zawierających coraz większą liczbę elementów, poczynając od dwóch (2 bajty), a kończąc na `n + 1` elementach (n + 1 bajtów). Wówczas dla każdej tablicy następuje w pętli zapisanie znaku `c` do poszczególnych jej elementów, za wyjątkiem ostatniego (pozostawiając tam bajt o wartości `0`, który jednocześnie jest ostatnim w danej tablicy), wyprowadzenie zawartości tablicy w postaci łańcucha znaków, a następnie zwolnienie przydzielonej dynamicznie pamięci.

Wywołanie funkcji `print_chars()` z parametrami `n` równym `5`, a `c` równym `X`, wymusi uzyskanie następującego wyniku:

```
X
XX
XXX
XXXX
XXXXX
```

Istnieją oczywiście dużo efektywniejsze metody pozwalające na zaimplementowanie takiej funkcji. Ważne jest jednak, że pamięć można dynamicznie przydzielać i zwalniać, nawet wówczas, gdy rozmiar i liczba przydzielonych obszarów znana jest tylko w momencie działania programu.

Systemy uniksowe, takie jak SunOS i SCO, udostępniają własny wariant funkcji `free()`, zwany `cfree()`, który w zależności od systemu działa tak samo jak `free()` lub posiada trzy parametry i wówczas zachowuje się jak funkcja `calloc()`. Funkcja `free()` w systemie Linux może obsłużyć pamięć uzyskaną dzięki użyciu dowolnego mechanizmu, służącego do jej przydzielania i już omówionego. Funkcja `cfree()` nie powinna być używana, za wyjątkiem zapewnienia wstecznej kompatybilności. Wersja tej funkcji dla Linuksa jest identyczna z `free()`.

Należy zauważyć, że gdyby w powyższym przykładzie nie użyto funkcji `free()`, pojawiłyby się pewne następstwa tego. Program mógłby nigdy nie zwolnić zajętego obszaru do systemu i co gorsze, stracić swoje jedyne odwołanie do pamięci — wskaźnik `s` — i przez to spowodować, że dostęp do niej stałby się w ogóle niemożliwy. Ten rodzaj błędu programistycznego

zwany jest *wyciekaniem pamięci* (ang. *memory leak*). Wyciekanie pamięci i tym podobne pomyłki, związane z pamięcią dynamiczną, są najczęstszymi i niestety najbardziej szkodliwymi błędami występującymi podczas programowania w języku C. Ponieważ język C zrzuca całą odpowiedzialność za zarządzanie pamięcią na programistów, muszą oni zwracać szczególną uwagę na wszystkie przydzielone obszary.

Równie często spotykaną pułapką języka C jest *używanie zasobów po ich zwolnieniu*. Problem ten występuje w momencie, gdy blok pamięci zostaje zwolniony, a następnie ponownie użyty. Gdy tylko funkcja `free()` zwolni dany obszar pamięci, program nie może już ponownie używać jego zawartości. Programiści powinni zwracać szczególną uwagę na zawieszone wskaźniki lub wskaźniki różne od `NULL`, które pomimo tego wskazują na niepoprawne obszary pamięci. Istnieją dwa powszechnie używane narzędzia pomagające w tych sytuacjach; są to *Electric Fence* i *valgrind*[5].

Wyrównanie

Wyrównanie danych dotyczy relacji pomiędzy ich adresem oraz obszarami pamięci udostępnianymi przez sprzęt. Zmienna posiadająca adres w pamięci, który jest wielokrotnością jej rozmiaru, zwana jest *zmienną naturalnie wyrównaną*. Na przykład, zmienna 32-bitowa jest naturalnie wyrównana, jeśli posiada adres w pamięci, który jest wielokrotnością 4 — oznacza to, że najniższe dwa bity adresu są równe zeru. Dlatego też typ danych, którego rozmiar wynosi 2^n bajtów, musi posiadać adres, którego *n* najmniej znaczących bitów jest ustawionych na zero.

Reguły, które dotyczą wyrównania, pochodzą od sprzętu. Niektóre architektury maszynowe posiadają bardzo rygorystyczne wymagania dotyczące wyrównania danych. W przypadku pewnych systemów, załadowanie danych, które nie są wyrównane, powoduje wygenerowanie pułapki procesora. Dla innych systemów dostęp do niewyrównanych danych jest bezpieczny, lecz związany z pogorszeniem sprawności działania. Podczas tworzenia kodu przenośnego należy unikać problemów związanych z wyrównaniem. Także wszystkie używane typy danych powinny być naturalnie wyrównane.

Przydzielanie pamięci wyrównanej

W większości przypadków kompilator oraz biblioteka języka C w sposób przezroczysty obsługują zagadnienia, związane z wyrównaniem. POSIX definiuje, że obszar pamięci, zwracany w wyniku wykonania funkcji `malloc()`, `calloc()` oraz `realloc()`, musi być prawidłowo wyrównany dla każdego standardowego typu danych języka C. W przypadku Linuksa funkcje te zawsze zwracają obszar pamięci, która wyrównana jest do adresu będącego wielokrotnością ośmiu bajtów w przypadku systemów 32-bitowych oraz do adresu, będącego wielokrotnością szesnastu bajtów dla systemów 64-bitowych.

Czasami programiści żądają przydzielenia takiego obszaru pamięci dynamicznej, który wyrównany jest do większego rozmiaru, posiadającego na przykład wielkość strony. Mimo istnienia różnych argumentacji, najbardziej podstawowym wymaganiem jest zdefiniowanie prawidłowo wyrównanych buforów, używanych podczas bezpośrednich operacji blokowych wejścia i wyjścia lub innej komunikacji między oprogramowaniem a sprzętem. W tym celu POSIX 1003.1d udostępnia funkcję zwaną `posix_memalign()`:

[5] Znajdują się one odpowiednio w następujących miejscach: *http://perens.com/FreeSoftware/ElectricFence/ oraz http://valgrind.org*.

```
/* należy użyć jednej z dwóch poniższych definicji - każda z nich jest odpowiednia */
#define _XOPEN_SOURCE 600
#define _GNU_SOURCE

#include <stdlib.h>
int posix_memalign (void **memptr, size_t alignment, size_t size);
```

Poprawne wywołanie funkcji `posix_memalign()` przydziela pamięć dynamiczną o rozmiarze przekazanym w parametrze `size` i wyrażonym w bajtach, zapewniając jednocześnie, że obszar ten zostanie wyrównany do adresu pamięci, będącego wielokrotnością parametru `alignment`. Parametr `alignment` musi być potęgą liczby 2 oraz wielokrotnością rozmiaru wskaźnika `void`. Adres przydzielonej pamięci zostaje umieszczony w parametrze `memptr`, a funkcja zwraca zero.

W przypadku błędu nie następuje przydzielenie pamięci, parametr `memptr` ma wartość nieokreśloną, a funkcja zwraca jedną z poniższych wartości kodów błędu:

`EINVAL`
: Parametr `alignment` nie jest potęgą liczby 2 lub wielokrotnością rozmiaru wskaźnika `void`.

`ENOMEM`
: Nie ma wystarczającej ilości pamięci, aby dokończyć rozpoczętą operację przydzielania pamięci.

Należy zauważyć, że zmienna `errno` nie zostaje ustawiona — funkcja bezpośrednio zwraca kod błędu.

Obszar pamięci, uzyskany za pomocą funkcji `posix_memalign()`, może zostać zwolniony przy użyciu `free()`. Sposób użycia funkcji jest prosty:

```
char *buf;
int ret;

/* przydziel 1 kB pamięci wyrównanej do adresu równego wielokrotności 256 bajtów */
ret = posix_memalign (&buf, 256, 1024);
if (ret)
{
   fprintf (stderr, "posix_memalign: %s\n", strerror (ret));
   return -1;
}

/ * tu można używać pamięci, wskazywanej przez 'buf'... */

free (buf);
```

Starsze interfejsy. Zanim w standardzie POSIX została zdefiniowana funkcja `posix_memalign()`, systemy BSD oraz SunOS udostępniały odpowiednio następujące interfejsy:

```
#include <malloc.h>

void * valloc (size_t size);
void * memalign (size_t boundary, size_t size);
```

Funkcja `valloc()` działa identycznie jak `malloc()`, za wyjątkiem tego, że przydzielona pamięć jest wyrównana do rozmiaru strony. Jak napisano w rozdziale 4., rozmiar systemowej strony można łatwo uzyskać po wywołaniu funkcji `getpagesize()`.

Funkcja `memalign()` jest podobna, lecz wyrównuje przydzieloną pamięć do rozmiaru przekazanego w parametrze `boundary` i wyrażonego w bajtach. Rozmiar ten musi być potęgą liczby 2. W poniższym przykładzie obie wspomniane funkcje alokacyjne zwracają blok pamięci o wielkości wystarczającej do przechowania struktury `ship`. Jest on wyrównany do rozmiaru strony:

```
struct ship *pirate, *hms;

pirate = valloc (sizeof (struct ship));
if (!pirate)
{
   perror ("valloc");
   return -1;
}

hms = memalign (getpagesize ( ), sizeof (struct ship));
if (!hms)
{
   perror ("memalign");
   free (pirate);
   return -1;
}

/* tu można używać obszaru pamięci wskazywanego przez 'pirate' i 'hms'... */

free (hms);
free (pirate);
```

W przypadku systemu Linux obszar pamięci, otrzymany za pomocą tych dwóch funkcji, może zostać zwolniony po wywołaniu funkcji `free()`. Nie musi tak być jednak w przypadku innych systemów uniksowych, gdyż niektóre z nich nie dostarczają żadnego mechanizmu pozwalającego na bezpieczne zwolnienie pamięci przydzielonej za pomocą wyżej wspomnianych funkcji. Dla programów, które powinny być przenośne, może nie istnieć inny wybór poza niezwalnianiem pamięci przydzielonej za pomocą tych interfejsów!

Programiści Linuksa powinni używać powyższych funkcji tylko wtedy, gdy należy zachować kompatybilność ze starszymi systemami; funkcja `posix_memalign()` jest lepsza. Użycie trzech wspomnianych funkcji jest niezbędne jedynie wtedy, gdy wymagany jest inny rodzaj wyrównania, niż dostarczony razem z funkcją `malloc()`.

Inne zagadnienia związane z wyrównaniem

Problemy związane z wyrównaniem obejmują większy obszar zagadnień niż tylko wyrównanie naturalne dla standardowych typów danych oraz dynamiczny przydział pamięci. Na przykład, typy niestandardowe oraz złożone posiadają bardziej skomplikowane wymagania niż typy standardowe. Ponadto, zagadnienia związane z wyrównaniem są szczególnie ważne w przypadku przypisywania wartości między wskaźnikami różnych typów oraz użycia rzutowania.

Typy niestandardowe. Niestandardowe i złożone typy danych posiadają większe wymagania dotyczące wyrównania przydzielonego obszaru pamięci. Zachowanie zwykłego wyrównania naturalnego nie jest wystarczające. W tych przypadkach stosuje się cztery poniższe reguły:

- Wyrównanie dla struktury jest równe wyrównaniu dla największego pod względem rozmiaru typu danych, z których zbudowane są jej pola. Na przykład, jeśli największy typ danych w strukturze jest 32-bitową liczbą całkowitą, która jest wyrównana do adresu będącego wielokrotnością czterech bajtów, wówczas sama struktura musi być także wyrównana do adresu będącego wielokrotnością co najmniej czterech bajtów.
- Użycie struktur wprowadza także konieczność stosowania wypełnienia, które jest wykorzystywane w celu zapewnienia, że każdy typ składowy będzie poprawnie wyrównany, zgodnie z jego wymaganiami. Dlatego też, jeśli po polu posiadającym typ `char` (o wyrównaniu prawdopodobnie równym jednemu bajtowi) pojawi się pole z typem `int` (posiadające

wyrównanie prawdopodobnie równe czterem bajtom), wówczas kompilator wstawi dodatkowe trzy bajty wypełnienia pomiędzy tymi dwoma polami o różnych typach danych, aby zapewnić, że `int` znajdzie się w obszarze wyrównanym do wielokrotności czterech bajtów. Programiści czasami porządkują pola w strukturze — na przykład, według malejącego rozmiaru typów składowych — aby zminimalizować obszar pamięci „tracony" na wypełnienie. Opcja kompilatora GCC, zwana `-Wpadded`, może pomóc w tym przypadku, ponieważ generuje ostrzeżenie w momencie, gdy kompilator wstawia domyślne wypełnienia.

- Wyrównanie dla unii jest równe wyrównaniu dla największego pod względem rozmiaru typu danych, z których zbudowane są jej pola.
- Wyrównanie dla tablicy jest równe wyrównaniu dla jej podstawowego typu danych. Dlatego też wymagania dla tablic są równe wymaganiu dotyczącemu pojedynczego elementu, z których się składają tablice. Zachowanie to powoduje, że wszystkie elementy tablicy posiadają wyrównanie naturalne.

Działania na wskaźnikach. Ponieważ kompilator w sposób przezroczysty obsługuje większość żądań związanych z wyrównaniem, dlatego też, aby doświadczyć ewentualnych problemów, wymagany jest większy wysiłek. Mimo to jest nieprawdą, że nie istnieją komplikacje związane z wyrównaniem, gdy używa się wskaźników i rzutowania.

Dostęp do danych poprzez rzutowanie wskaźnika z bloku pamięci o mniejszej wartości wyrównania na blok, posiadający większą wartość wyrównania, może spowodować, że dane te nie będą właściwie wyrównane dla typu o większym rozmiarze. Na przykład, przypisanie zmiennej `c` do `badnews` w poniższym fragmencie kodu powoduje, że zmienna ta będzie zrzutowana na typ `unsigned long`:

```
char greeting[] = "Ahoj Matey";
char *c = greeting[1];
unsigned long badnews = *(unsigned long *) c;
```

Typ `unsigned long` jest najprawdopodobniej wyrównany do adresu będącego wielokrotnością ośmiu bajtów; zmienna `c` prawie na pewno przesunięta jest o 1 bajt poza tę granicę. Odczytanie zmiennej `c` podczas wykonywania rzutowania spowoduje powstanie błędu wyrównania. W zależności od architektury może być to przyczyną różnych zachowań, poczynając od mniej ważnych, np. pogorszenie sprawności działania, a kończąc na poważnych, jak załamanie programu. W architekturach maszynowych, które potrafią wykryć, lecz nie mogą poprawnie obsłużyć błędów wyrównania, jądro wysyła do takich niepoprawnych procesów sygnał `SIGBUS`, który przerywa ich działanie. Sygnały zostaną omówione w rozdziale 9.

Przykłady podobne do powyższego są częściej spotykane, niż sądzimy. Niepoprawne konstrukcje programowe, spotykane w świecie realnym, nie będą wyglądać tak bezmyślnie, lecz będą najprawdopodobniej trudniejsze do wykrycia.

Zarządzanie segmentem danych

Od zawsze system Unix udostępniał interfejsy pozwalające na bezpośrednie zarządzanie segmentem danych. Jednak większość programów nie posiada bezpośredniego dostępu do tych interfejsów, ponieważ funkcja `malloc()` i inne sposoby przydzielania pamięci są łatwiejsze w użyciu, a jednocześnie posiadają większe możliwości. Interfejsy te zostaną jednak omówione,

aby zaspokoić ciekawość czytelników i udostępnić dociekliwym programistom metodę pozwalającą na zaimplementowanie swojego własnego mechanizmu przydzielania pamięci, opartego na stercie:

```
#include <unistd.h>

int brk (void *end);
void * sbrk (intptr_t increment);
```

Funkcje te dziedziczą swoje nazwy z dawnych systemów uniksowych, dla których sterta i stos znajdowały się w tym samym segmencie. Przydzielanie obszarów pamięci dynamicznej na stercie powoduje jej narastanie od dolnej części segmentu, w kierunku adresów wyższych; stos rośnie w kierunku przeciwnym — od szczytu segmentu do niższych adresów. Linia graniczna pomiędzy tymi dwoma strukturami danych zwana jest *podziałem* lub *punktem podziału* (ang. *break* lub *break point*). W nowoczesnych systemach operacyjnych, w których segment danych posiada swoje własne odwzorowanie pamięci, końcowy adres tego odwzorowania w dalszym ciągu zwany jest punktem podziału.

Wywołanie funkcji `brk()` ustawia punkt podziału (koniec segmentu danych) na adres przekazany w parametrze `end`. W przypadku sukcesu, funkcja zwraca wartość `0`. W przypadku błędu, zwraca `–1` oraz ustawia zmienną `errno` na `ENOMEM`.

Wywołanie funkcji `sbrk()` zwiększa adres końca segmentu o wartość przekazaną w parametrze `increment`, który może być przyrostem dodatnim lub ujemnym. Funkcja `sbrk()` zwraca uaktualnioną wartość położenia punktu podziału. Dlatego też użycie parametru `increment` równego zeru powoduje wyprowadzenie aktualnej wartości położenia punktu podziału:

```
printf ("Aktualny punkt podziału posiada adres %p\n", sbrk (0));
```

Oba standardy — POSIX i C — celowo nie definiują żadnej z powyższych funkcji. Prawie wszystkie systemy uniksowe wspierają jednak jedną lub obie te funkcje. Programy przenośne powinny używać interfejsów zdefiniowanych w standardach.

Anonimowe odwzorowania w pamięci

W celu wykonania operacji przydzielania pamięci, zaimplementowanej w bibliotece *glibc*, używany jest segment danych oraz odwzorowania pamięci. Klasyczną metodą, zastosowaną w celu implementacji funkcji `malloc()`, jest podział segmentu danych na ciąg partycji o rozmiarach potęgi liczby 2 oraz zwracanie tego obszaru, który najlepiej pasuje do żądanej wielkości. Zwalnianie pamięci jest prostym oznaczaniem, że dana partycja jest „wolna". Kiedy graniczące ze sobą partycje są nieużywane, mogą zostać połączone w jeden większy obszar pamięci. Jeśli szczyt sterty jest zupełnie nieprzydzielony, system może użyć funkcji `brk()`, aby obniżyć adres położenia punktu podziału, a przez to zmniejszyć rozmiar tej struktury danych i zwrócić pamięć do jądra.

Algorytm ten zwany jest *schematem przydziału wspieranej pamięci* (ang. *buddy memory allocation scheme*). Posiada takie zalety jak prędkość i prostota, ale również wady w postaci dwóch rodzajów fragmentacji. *Fragmentacja wewnętrzna* (ang. *internal fragmentation*) występuje wówczas, gdy więcej pamięci, niż zażądano, zostanie użyte w celu wykonania operacji przydziału. Wynikiem tego jest nieefektywne użycie dostępnej pamięci. *Fragmentacja zewnętrzna* (ang. *external fragmentation*) występuje wówczas, gdy istnieje wystarczająca ilość pamięci, aby zapewnić wykonanie operacji przydziału, lecz jest ona podzielona na dwa lub więcej niesąsiadujących ze sobą frag-

mentów. Fragmentacja ta może powodować nieefektywne użycie pamięci (ponieważ może zostać użyty większy, mniej pasujący blok) lub niepoprawne wykonanie operacji jej przydziału (jeśli nie ma innych bloków).

Ponadto, schemat ten pozwala, aby pewien przydzielony obszar mógł „unieruchomić" inny, co może spowodować, że biblioteka *glibc* nie będzie mogła zwrócić zwolnionej pamięci do jądra. Załóżmy, że istnieją dwa przydzielone obszary pamięci: blok *A* i blok *B*. Blok *A* znajduje się dokładnie w punkcie podziału, a blok *B* zaraz pod nim. Nawet jeśli program zwolni blok *B*, biblioteka *glibc* nie będzie mogła uaktualnić położenia punktu podziału, dopóki blok *A* również nie zostanie zwolniony. W ten sposób aplikacje, których czas życia w systemie jest długi, mogą unieruchomić wszystkie inne przydzielone obszary pamięci.

Nie zawsze jest to problemem, gdyż biblioteka *glibc* nie zwraca w sposób rutynowy pamięci do systemu[6]. Sterta zazwyczaj nie zostaje zmniejszona po każdej operacji zwolnienia pamięci. Zamiast tego biblioteka *glibc* zachowuje zwolnioną pamięć, aby użyć jej w następnej operacji przydzielania. Tylko wówczas, gdy rozmiar sterty jest znacząco większy od ilości przydzielonej pamięci, biblioteka *glibc* faktycznie zmniejsza wielkość segmentu danych. Przydział dużej ilości pamięci może jednak przeszkodzić temu zmniejszeniu.

Zgodnie z tym, w przypadku przydziałów dużej ilości pamięci, w bibliotece *glibc* nie jest używana sterta. Biblioteka *glibc* tworzy *anonimowe odwzorowanie w pamięci*, aby zapewnić poprawne wykonanie żądania przydziału. Anonimowe odwzorowania w pamięci są podobne do odwzorowań dotyczących plików i omówionych w rozdziale 4., za wyjątkiem tego, że nie są związane z żadnym plikiem — stąd też przydomek „anonimowy". Takie anonimowe odwzorowanie jest po prostu dużym blokiem pamięci, wypełnionym zerami i gotowym do użycia. Należy traktować go jako nową stertę używaną wyłącznie w jednej operacji przydzielania pamięci. Ponieważ takie odwzorowania są umieszczane poza stertą, nie przyczyniają się do fragmentacji segmentu danych.

Przydzielanie pamięci za pomocą anonimowych odwzorowań ma kilka zalet:

- Nie występuje fragmentacja. Gdy program nie potrzebuje już anonimowego odwzorowania w pamięci, jest ono usuwane, a pamięć zostaje natychmiast zwrócona do systemu.
- Można zmieniać rozmiar anonimowych odwzorowań w pamięci, posiadają one modyfikowane uprawnienia, a także mogą otrzymywać poradę — podobnie, jak ma to miejsce w przypadku zwykłych odwzorowań (szczegóły w rozdziale 4.).
- Każdy przydział pamięci realizowany jest w oddzielnym odwzorowaniu. Nie ma potrzeby użycia globalnej sterty.

Istnieją również wady używania anonimowych odwzorowań w pamięci, w porównaniu z użyciem sterty:

- Rozmiar każdego odwzorowania w pamięci jest całkowitą wielokrotnością rozmiaru strony systemowej. Zatem takie operacje przydziałów, dla których rozmiary nie są całkowitą wielokrotnością rozmiaru strony, generują powstawanie nieużywanych obszarów „wolnych". Problem przestrzeni wolnej dotyczy głównie małych obszarów przydziału, dla których pamięć nieużywana jest stosunkowo duża w porównaniu z rozmiarem przydzielonego bloku.

---

[6] W celu przydzielania pamięci, biblioteka *glibc* używa również dużo bardziej zaawansowanego algorytmu niż zwykłego schematu przydziału wspieranej pamięci. Algorytm ten zwany jest *algorytmem areny* (ang. *arena algorithm*).

- Tworzenie nowego odwzorowania w pamięci wymaga większego nakładu pracy niż zwracanie pamięci ze sterty, które może w ogóle nie obciążać jądra. Im obszar przydziału jest mniejszy, tym to zjawisko jest bardziej widoczne.

  Porównując zalety i wady, można stwierdzić, że funkcja `malloc()` w bibliotece *glibc* używa segmentu danych, aby zapewnić poprawne wykonanie operacji przydziału niewielkich obszarów, natomiast anonimowych odwzorowań w pamięci, aby zapewnić przydzielenie dużych obszarów. Próg działania jest konfigurowalny (szczegóły w podrozdziale Zaawansowane operacje przydziału pamięci, znajdującym się w dalszej części tego rozdziału) i może być inny dla każdej wersji biblioteki *glibc*. Obecnie próg wynosi 128 kB: operacje przydziału o obszarach mniejszych lub równych 128 kB używają sterty, natomiast większe przydziały korzystają z anonimowych odwzorowań w pamięci.

Tworzenie anonimowych odwzorowań w pamięci

Wymuszenie użycia mechanizmu odwzorowania w pamięci zamiast wykorzystania sterty w celu wykonania określonego przydziału, kreowanie własnego systemu zarządzającego przydziałem pamięci, ręczne tworzenie anonimowego odwzorowania w pamięci — te wszystkie operacje są łatwe do zrealizowania w systemie Linux. W rozdziale 4. napisano, że odwzorowanie w pamięci może zostać utworzone przez funkcję systemową `mmap()`, natomiast usunięte przez funkcję systemową `munmap()`:

```
#include <sys/mman.h>

void * mmap (void *start, size_t length, int prot, int flags, int fd, off_t offset);
int munmap (void *start, size_t length);
```

Kreowanie anonimowego odwzorowania w pamięci jest nawet prostsze niż tworzenie odwzorowania opartego na pliku, ponieważ nie trzeba tego pliku otwierać i nim zarządzać. Podstawową różnicą między tymi dwoma rodzajami odwzorowania jest specjalny znacznik, wskazujący, że dane odwzorowanie jest anonimowe.

Oto przykład:

```
void *p;

p = mmap (NULL,                          /* nieważne, w jakim miejscu pamięci */
  512 * 1024,                            /* 512 kB */
    PROT_READ | PROT_WRITE,              /* zapis/odczyt */
      MAP_ANONYMOUS | MAP_PRIVATE,       /* odwzorowanie anonimowe i prywatne */
        -1,                              /* deskryptor pliku (ignorowany) */
           0);                           /* przesunięcie (ignorowane) */

if (p == MAP_FAILED)
   perror ("mmap");
else
   /* 'p' wskazuje na obszar 512 kB anonimowej pamięci... */
```

W większości anonimowych odwzorowań parametry funkcji `mmap()` są takie same jak w powyższym przykładzie, oczywiście za wyjątkiem rozmiaru, przekazanego w parametrze `length` i wyrażonego w bajtach, który jest określany przez programistę. Pozostałe parametry są następujące:

- Pierwszy parametr, `start`, ustawiony jest na wartość `NULL`, co oznacza, że anonimowe odwzorowanie może rozpocząć się w dowolnym miejscu w pamięci — decyzja w tym przypadku należy do jądra. Podawanie wartości różnej od `NULL` jest dopuszczalne, dopóki

jest ona wyrównana do wielkości strony, lecz ogranicza to przenośność. Położenie odwzorowania jest rzadko wykorzystywane przez programy.

- Parametr `prot` zwykle ustawia oba bity `PROT_READ` oraz `PROT_WRITE`, co powoduje, że odwzorowanie posiada uprawienia do odczytu i zapisu. Odwzorowanie bez uprawnień nie ma sensu, gdyż nie można z niego czytać ani do niego zapisywać. Z drugiej strony, zezwolenie na wykonywanie kodu z anonimowego odwzorowania jest rzadko potrzebne, a jednocześnie tworzy potencjalną lukę bezpieczeństwa.
- Parametr `flags` ustawia bit `MAP_ANONYMOUS`, który oznacza, że odwzorowanie jest anonimowe, oraz bit `MAP_PRIVATE`, który nadaje odwzorowaniu status prywatności.
- Parametry `fd` i `offset` są ignorowane, gdy ustawiony jest znacznik `MAP_ANONYMOUS`. Niektóre starsze systemy oczekują jednak, że w parametrze `fd` zostanie przekazana wartość `-1`, dlatego też warto to uczynić, gdy ważnym czynnikiem jest przenośność.

Pamięć, otrzymana za pomocą mechanizmu anonimowego odwzorowania, wygląda tak samo jak pamięć ze sterty. Jedną korzyścią z użycia anonimowego odwzorowania jest to, że strony są już wypełnione zerami. Jest to wykonywane bez jakichkolwiek kosztów, ponieważ jądro odwzorowuje anonimowe strony aplikacji na stronę wypełnioną zerami, używając do tego celu mechanizmu kopiowania podczas zapisu. Dlatego też nie jest wymagane użycie funkcji `memset()` dla zwróconego obszaru pamięci. Faktycznie istnieje jedna korzyść z użycia funkcji `calloc()` zamiast zestawu `malloc()`oraz `memset()`: biblioteka *glibc* jest poinformowana, że obszar anonimowego odwzorowania jest już wypełniony zerami, a funkcja `calloc()`, po poprawnym przydzieleniu pamięci, nie wymaga jawnego jej zerowania.

Funkcja systemowa `munmap()` zwalnia anonimowe odwzorowanie, zwracając przydzieloną pamięć do jądra:

```
int ret;

/* wykonano wszystkie działania, związane z użyciem wskaźnika 'p', dlatego należy zwrócić 512 kB pamięci */
ret = munmap (p, 512 * 1024);
if (ret)
   perror ("munmap");
```

Szczegóły użycia funkcji `mmap()`, `munmap()` oraz ogólny opis mechanizmu odwzorowania znajdują się w rozdziale 4.

Odwzorowanie pliku /dev/zero

Inne systemy operacyjne, takie jak BSD, nie posiadają znacznika `MAP_ANONYMOUS`. Zamiast tego zaimplementowane jest dla nich podobne rozwiązanie, przy użyciu odwzorowania specjalnego pliku urządzenia */dev/zero*. Ten plik urządzenia dostarcza takiej samej semantyki jak anonimowa pamięć. Odwzorowanie zawiera strony uzyskane za pomocą mechanizmu kopiowania podczas zapisu, wypełnione zerami; dlatego też zachowanie to jest takie samo jak w przypadku anonimowej pamięci.

Linux zawsze posiadał urządzenie */dev/zero* oraz udostępniał możliwość odwzorowania tego pliku i uzyskania obszaru pamięci wypełnionego zerami. Rzeczywiście, zanim wprowadzono znacznik `MAP_ANONYMOUS`, programiści w Linuksie używali powyższego rozwiązania. Aby

zapewnić wsteczną kompatybilność ze starszymi wersjami Linuksa lub przenośność do innych systemów Uniksa, projektanci w dalszym ciągu mogą używać pliku urządzenia */dev/zero*, aby stworzyć anonimowe odwzorowanie. Operacja ta nie różni się od tworzenia odwzorowania dla innych plików:

```
void *p;
int fd;

/* otwórz plik /dev/zero do odczytu i zapisu */
fd = open ("/dev/zero", O_RDWR);
if (fd < 0)
{
   perror ("open");
   return -1;
}

/* odwzoruj obszar [0, rozmiar strony) dla urządzenia /dev/zero */
p = mmap (NULL,                     /* nieważne, w jakim miejscu pamięci */
  getpagesize ( ),                  /* odwzoruj jedną stronę */
    PROT_READ | PROT_WRITE,         /* uprawnienia odczytu i zapisu */
      MAP_PRIVATE,                  /* odwzorowanie prywatne */
        fd,                         /* odwzoruj plik /dev/zero */
          0);                       /* bez przesunięcia */
if (p == MAP_FAILED)
{
   perror ("mmap");
   if (close (fd))
      perror ("close");
   return -1;
}

/* zamknij plik /dev/zero, jeśli nie jest już potrzebny */
if (close (fd))
   perror ("close");

/* wskaźnik 'p' wskazuje na jedną stronę w pamięci, można go używać... */
```

Pamięć, otrzymana za pomocą powyżej przedstawionego sposobu, może oczywiście zostać zwolniona przy użyciu funkcji `munmap()`.

Ta metoda generuje dodatkowe obciążenie przez użycie funkcji systemowej, otwierającej i zamykającej plik urządzenia. Dlatego też wykorzystanie pamięci anonimowej jest rozwiązaniem szybszym.

Zaawansowane operacje przydziału pamięci

Wiele operacji przydziału pamięci, omówionych w tym rozdziale, jest ograniczanych i sterowanych przez parametry jądra, które mogą zostać modyfikowane przez programistę. Aby to wykonać, należy użyć funkcji `mallopt()`:

```
#include <malloc.h>

int mallopt (int param, int value);
```

Wywołanie funkcji `mallopt()` ustawia parametr związany z zarządzaniem pamięcią, którego nazwa przekazana jest w argumencie `param`. Parametr ten zostaje ustawiony na wartość równą argumentowi `value`. W przypadku sukcesu funkcja zwraca wartość niezerową; w przypadku błędu zwraca `0`. Należy zauważyć, że funkcja `mallopt()` nie ustawia zmiennej `errno`. Najczęściej

jej wywołanie również kończy się sukcesem, dlatego też nie należy optymistycznie podchodzić do zagadnienia uzyskiwania użytecznej informacji z jej kodu powrotu.

Linux wspiera obecnie sześć wartości dla parametru `param`, które zdefiniowane są w pliku nagłówkowym `<malloc.h>`:

`M_CHECK_ACTION`
: Wartość zmiennej środowiskowej `MALLOC_CHECK_` (omówiona w następnym podrozdziale).

`M_MMAP_MAX`
: Maksymalna liczba odwzorowań, które mogą zostać udostępnione przez system, aby poprawnie zrealizować żądania przydzielania pamięci dynamicznej. Gdy to ograniczenie zostanie osiągnięte, wówczas dla kolejnych przydziałów pamięci zostanie użyty segment danych, dopóki jedno z odwzorowań nie zostanie zwolnione. Wartość `0` całkowicie uniemożliwia użycie mechanizmu anonimowych odwzorowań jako podstawy do wykonywania operacji przydziału pamięci dynamicznej.

`M_MMAP_THRESHOLD`
: Wielkość progu (wyrażona w bajtach), powyżej którego żądanie przydziału pamięci zostanie zrealizowane za pomocą anonimowego odwzorowania zamiast udostępnienia segmentu danych. Należy zauważyć, że przydziały mniejsze od tego progu mogą również zostać zrealizowane za pomocą anonimowych odwzorowań, ze względu na swobodę postępowania pozostawioną systemowi. Wartość `0` umożliwia użycie anonimowych odwzorowań dla wszystkich operacji przydziału, stąd też w rzeczywistości nie zezwala na wykorzystanie dla nich segmentu danych.

`M_MXFAST`
: Maksymalny rozmiar (wyrażony w bajtach) podajnika szybkiego. *Podajniki szybkie* (ang. *fast bins*) są specjalnymi fragmentami pamięci na stercie, które nigdy nie zostają połączone z sąsiednimi obszarami i nie są zwrócone do systemu. Pozwala to na wykonywanie bardzo szybkich operacji przydziału, kosztem zwiększonej fragmentacji. Wartość `0` całkowicie uniemożliwia użycie podajników szybkich.

`M_TOP_PAD`
: Wartość uzupełnienia (w bajtach) użytego podczas zmiany rozmiaru segmentu danych. Gdy biblioteka *glibc* wykonuje funkcję `brk()`, aby zwiększyć rozmiar segmentu danych, może zażyczyć sobie więcej pamięci, niż w rzeczywistości potrzebuje, w nadziei na to, że dzięki temu w najbliższej przyszłości nie będzie konieczne wykonanie kolejnego wywołania tejże funkcji. Podobnie dzieje się w przypadku, gdy biblioteka *glibc* zmniejsza rozmiar segmentu danych — zachowuje ona dla siebie pewną ilość pamięci, zwracając do systemu mniej, niż mogłaby naprawdę oddać. Ten dodatkowy obszar pamięci jest omawianym *uzupełnieniem*. Wartość `0` uniemożliwia całkowicie użycie wypełnienia.

`M_TRIM_THRESHOLD`
: Minimalna ilość wolnej pamięci (w bajtach), która może istnieć na szczycie segmentu danych. Jeśli liczba ta będzie mniejsza od podanego progu, biblioteka *glibc* wywoła funkcję `brk()`, aby zwrócić pamięć do jądra.

Standard XPG, który w luźny sposób definiuje funkcję `mallopt()`, określa trzy inne parametry: `M_GRAIN`, `M_KEEP` oraz `M_NLBLKS`. Linux również je definiuje, lecz ustawianie dla nich wartości nie powoduje żadnych zmian. W tabeli 8.1. znajduje się pełny opis wszystkich poprawnych parametrów oraz odpowiednich dla nich domyślnych wartości. Podane są również zakresy akceptowalnych wartości.

Tabela 8.1. Parametry funkcji `mallopt()`

Parametr	Źródło pochodzenia	Wartość domyślna	Poprawne wartości	Wartości specjalne
M_CHECK_ACTION	Specyficzny dla Linuksa	0	0 – 2	
M_GRAIN	Standard XPG	Brak wsparcia w Linuksie	>= 0	
M_KEEP	Standard XPG	Brak wsparcia w Linuksie	>= 0	
M_MMAP_MAX	Specyficzny dla Linuksa	64 * 1024	>= 0	0 uniemożliwia użycie mmap()
M_MMAP_THRESHOLD	Specyficzny dla Linuksa	128 * 1024	>= 0	0 uniemożliwia użycie sterty
M_MXFAST	Standard XPG	64	0 – 80	0 uniemożliwia użycie podajników szybkich
M_NLBLKS	Standard XPG	Brak wsparcia w Linuksie	>= 0	
M_TOP_PAD	Specyficzny dla Linuksa	0	>= 0	0 uniemożliwia użycie uzupełnienia

Dowolne wywołanie funkcji `mallopt()` w programach musi wystąpić przed pierwszym użyciem funkcji `malloc()` lub innych interfejsów, służących do przydzielania pamięci. Użycie jest proste:

```
int ret;

/* użyj funkcji mmap( ) dla wszystkich przydziałów pamięci większych od 64 kB */
ret = mallopt (M_MMAP_THRESHOLD, 64 * 1024);
if (!ret)
   fprintf (stderr, "Wywołanie funkcji mallopt() nie powiodło się!\n");
```

Dokładne dostrajanie przy użyciu funkcji malloc_usable_size() oraz malloc_trim()

Linux dostarcza kilku funkcji, które pozwalają na niskopoziomową kontrolę działania systemu przydzielania pamięci dla biblioteki *glibc*. Pierwsza z tych funkcji pozwala na uzyskanie informacji, ile faktycznie dostępnych bajtów zawiera dany obszar przydzielonej pamięci:

```
#include <malloc.h>

size_t malloc_usable_size (void *ptr);
```

Poprawne wywołanie funkcji `malloc_usable_size()` zwraca rzeczywisty rozmiar przydziału dla obszaru pamięci wskazywanego przez `ptr`. Ponieważ biblioteka *glibc* może zaokrąglać wielkości przydziałów, aby dopasować się do istniejącego fragmentu pamięci, przydzielonego do anonimowego odwzorowania, dlatego też wielkość przestrzeni dla danego przydziału, nadającej się do użytku, może być większa od tej, jaką zażądano. Oczywiście obszary przydziałów pamięci nie będą nigdy mniejsze od tych, jakie są wymagane. Oto przykład użycia funkcji:

```
size_t len = 21;
size_t size;
char *buf;

buf = malloc (len);
if (!buf)
{
   perror ("malloc");
```

```
    return -1;
}

size = malloc_usable_size (buf);

/* w rzeczywistości można użyć 'size' bajtów z obszaru pamięci 'buf'... */
```

Wywołanie drugiej funkcji nakazuje bibliotece *glibc*, aby natychmiast zwróciła całą zwolnioną pamięć do jądra:

```
#include <malloc.h>

int malloc_trim (size_t padding);
```

Poprawne wywołanie funkcji `malloc_trim()` powoduje maksymalne zmniejszenie rozmiaru segmentu danych, za wyjątkiem obszarów uzupełnień, które są zarezerwowane. Następnie funkcja zwraca `1`. W przypadku błędu zwraca `0`. Zazwyczaj biblioteka *glibc* samodzielnie przeprowadza takie operacje zmniejszania rozmiaru segmentu danych, gdy tylko wielkość pamięci zwolnionej osiąga wartość `M_TRIM_THRESHOLD`. Biblioteka używa uzupełnienia określonego w parametrze `M_TOP_PAD`.

Programista nie będzie potrzebował nigdy użyć obu wspomnianych funkcji do niczego innego niż tylko celów edukacyjnych i wspomagających uruchamianie programów. Nie są one przenośne i udostępniają programowi użytkownika niskopoziomowe szczegóły systemu przydzielania pamięci, zaimplementowanego w bibliotece *glibc*.

Uruchamianie programów, używających systemu przydzielania pamięci

Programy mogą ustawiać zmienną środowiskową `MALLOC_CHECK_`, aby umożliwić poszerzone wspomaganie podczas uruchamiania programów wykorzystujących podsystem pamięci. Opcja poszerzonego wspomagania uruchamiania działa kosztem zmniejszenia efektywności operacji przydzielania pamięci, lecz obciążenie to jest często tego warte podczas tworzenia aplikacji i w trakcie jej uruchamiania.

Ponieważ zmienna środowiskowa steruje procesem wspomagania uruchamiania, dlatego też nie istnieje potrzeba, aby ponownie kompilować program. Na przykład, można wykonać proste polecenie, podobne do poniżej przedstawionego:

```
$ MALLOC_CHECK_=1 ./rudder
```

Jeśli zmienna `MALLOC_CHECK_` zostanie ustawiona na `0`, podsystem pamięci w sposób automatyczny zignoruje wszystkie błędy. W przypadku, gdy będzie ona równa `1`, na standardowe wyjście błędów `stderr` zostanie wysłany komunikat informacyjny. Jeśli zmienna ta będzie równa `2`, wykonanie programu zostanie natychmiast przerwane przy użyciu funkcji `abort()`. Ponieważ zmienna `MALLOC_CHECK_` modyfikuje zachowanie działającego programu, jest ignorowana przez aplikacje posiadające ustawiony bit SUID.

Otrzymywanie danych statystycznych

Linux dostarcza funkcji `mallinfo()`, która może zostać użyta w celu uzyskania danych statystycznych dotyczących działania systemu przydzielania pamięci:

```
#include <malloc.h>

struct mallinfo mallinfo (void);
```

Wywołanie funkcji `mallinfo()` zwraca dane statystyczne zapisane w strukturze `mallinfo`. Struktura zwracana jest przez wartość, a nie przez wskaźnik. Jej zawartość jest również zdefiniowana w pliku nagłówkowym `<malloc.h>`:

```
/* wszystkie rozmiary w bajtach */
struct mallinfo
{
    int arena;     /* rozmiar segmentu danych, używanego przez funkcję malloc */
    int ordblks;   /* liczba wolnych fragmentów pamięci */
    int smblks;    /* liczba podajników szybkich */
    int hblks;     /* liczba anonimowych odwzorowań */
    int hblkhd;    /* rozmiar anonimowych odwzorowań */
    int usmblks;   /* maksymalny rozmiar całkowitego przydzielonego obszaru */
    int fsmblks;   /* rozmiar dostępnych podajników szybkich */
    int uordblks;  /* rozmiar całkowitego przydzielonego obszaru */
    int fordblks;  /* rozmiar dostępnych fragmentów pamięci */
    int keepcost;  /* rozmiar obszaru, który może zostać zwrócony do systemu przy użyciu funkcji malloc_trim() */
};
```

Użycie funkcji jest proste:

```
struct mallinfo m;

m = mallinfo ( );

printf ("Liczba wolnych fragmentów pamięci: %d\n", m.ordblks);
```

Linux dostarcza również funkcji `malloc_stats()`, która wyprowadza na standardowe wyjście błędów dane statystyczne związane z podsystemem pamięci:

```
#include <malloc.h>

void malloc_stats (void);
```

Wywołanie funkcji `malloc_stats()` dla programu, który intensywnie używa pamięci, powoduje wyprowadzenie kilku większych liczb:

```
Arena 0:
system bytes     =  865939456
in use bytes     =  851988200
Total (incl. mmap):
system bytes     = 3216519168
in use bytes     = 3202567912
max mmap regions =      65536
max mmap bytes   = 2350579712
```

Przydziały pamięci wykorzystujące stos

Wszystkie mechanizmy omówione do tej pory, dotyczące wykonywania operacji przydziału pamięci dynamicznej, używały sterty lub odwzorowań w pamięci, aby zrealizować przydzielenie obszaru tejże pamięci. Należało tego oczekiwać, gdyż sterta i odwzorowania w pamięci są z definicji bardzo dynamicznymi strukturami. Inną, powszechnie używaną strukturą w przestrzeni adresowej programu jest stos, w którym zapamiętane są *automatyczne zmienne* dla aplikacji.

Nie istnieje jednak przeciwwskazanie, aby programista nie mógł używać stosu dla realizowania operacji przydzielania pamięci dynamicznej. Dopóki taka metoda przydziału pamięci nie przepełni stosu, może być prosta w realizacji i powinna działać zupełnie dobrze. Aby dynamicznie przydzielić pamięć na stosie, należy użyć funkcji systemowej `alloca()`:

```
#include <alloca.h>

void * alloca (size_t size);
```

W przypadku sukcesu, wywołanie funkcji `alloca()` zwraca wskaźnik do obszaru pamięci posiadającego rozmiar przekazany w parametrze `size` i wyrażony w bajtach. Pamięć ta znajduje się na stosie i zostaje automatycznie zwolniona, gdy wywołująca funkcja kończy swoje działanie. Niektóre implementacje zwracają wartość `NULL` w przypadku błędu, lecz dla większości z nich wywołanie funkcji `alloca()` nie może się nie udać lub nie jest możliwe informowanie o niepoprawnym jej wykonaniu. Na błąd wskazuje przepełniony stos.

Użycie jest identyczne jak w przypadku funkcji `malloc()`, lecz nie trzeba (w rzeczywistości *nie wolno*) zwalniać przydzielonej pamięci. Poniżej przedstawiony zostanie przykład funkcji, która otwiera dany plik z systemowego katalogu konfiguracyjnego (równego prawdopodobnie */etc*), lecz dla zwiększenia przenośności jego nazwa określana jest w czasie wykonania programu. Funkcja musi przydzielić pamięć dla nowego bufora, skopiować do niego nazwę systemowego katalogu konfiguracyjnego, a następnie połączyć ten bufor z dostarczoną nazwą pliku:

```
int open_sysconf (const char *file, int flags, int mode)
{
   const char *etc = SYSCONF_DIR; /* "/etc/" */
   char *name;

   name = alloca (strlen (etc) + strlen (file) + 1);
   strcpy (name, etc);
   strcat (name, file);
   return open (name, flags, mode);
}
```

Po powrocie z funkcji, pamięć przydzielona za pomocą funkcji `alloca()` zostaje automatycznie zwolniona, ponieważ wskaźnik stosu przesuwa się do pozycji funkcji wywołującej. Oznacza to, że nie można użyć przydzielonego obszaru pamięci po tym, gdy zakończy się funkcja używająca wywołania `alloca()`! Ponieważ nie należy wykonywać żadnego porządkowania pamięci za pomocą funkcji `free()`, ostateczny kod programu staje się trochę bardziej przejrzysty. Oto ta sama funkcja, lecz zaimplementowana przy użyciu wywołania `malloc()`:

```
int open_sysconf (const char *file, int flags, int mode)
{
   const char *etc = SYSCONF_DIR; /* "/etc/" */
   char *name;
   int fd;

   name = malloc (strlen (etc) + strlen (file) + 1);
   if (!name)
   {
      perror ("malloc");
      return -1;
   }

   strcpy (name, etc);
   strcat (name, file);
   fd = open (name, flags, mode);
   free (name);
```

```
    return fd;
}
```

Należy zauważyć, że w parametrach wywołania funkcji nie powinno używać się bezpośredniego wywołania `alloca()`. Powodem takiego zachowania jest to, że przydzielona pamięć będzie istnieć na stosie pośrodku obszaru zarezerwowanego do przechowywania parametrów funkcji. Na przykład, poniższy kod jest niepoprawny:

```
/* TAK NIE NALEŻY ROBIĆ! */
ret = foo (x, alloca (10));
```

Interfejs `alloca()` posiada ciekawą historię. W przypadku wielu systemów jego działanie było nieprawidłowe lub w pewnym sensie niezdefiniowane. W systemach posiadających nieduży stos o stałym rozmiarze, użycie funkcji `alloca()` było łatwym sposobem, aby go przepełnić i w rezultacie załamać wykonanie programu. W niektórych systemach funkcja `alloca()` nie jest do tej pory zaimplementowana. Błędne i niespójne implementacje funkcji `alloca()` spowodowały, że cieszy się ona złą reputacją.

Jeśli program powinien być przenośny, nie należy używać w nim funkcji `alloca()`. W przypadku systemu Linux funkcja ta jest jednak bardzo użytecznym i niedocenionym narzędziem. Działa wyjątkowo dobrze — w przypadku wielu architektur realizowanie przydzielania pamięci za pomocą tej funkcji nie powoduje niczego ponad zwiększenie wskaźnika stosu, dlatego też łatwo przewyższa ona pod względem wydajności funkcję `malloc()`. W przypadku niewielkich obszarów przydzielonej pamięci i kodu, specyficznego dla Linuksa, użycie funkcji `alloca()` może spowodować bardzo dobrą poprawę wydajności.

Powielanie łańcuchów znakowych na stosie

Powszechnym przykładem użycia funkcji `alloca()` jest tymczasowe powielanie łańcucha znakowego. Na przykład:

```
/* należy powielić łańcuch 'song' */
char *dup;

dup = alloca (strlen (song) + 1);
strcpy (dup, song);

/* tutaj można już używać wskaźnika 'dup'... */

return; /* 'dup' zostaje automatycznie zwolniony */
```

Z powodu częstego użycia tego rozwiązania, a również korzyści związanych z prędkością działania, jaką oferuje funkcja `alloca()`, systemy linuksowe udostępniają wersję funkcji `strdup()`, która pozwala na powielenie danego łańcucha znakowego na stosie:

```
#define _GNU_SOURCE
#include <string.h>

char * strdupa (const char *s);
char * strndupa (const char *s, size_t n);
```

Wywołanie funkcji `strdupa()` zwraca kopię łańcucha `s`. Wywołanie funkcji `strndupa()` powiela `n` znaków łańcucha `s`. Jeśli łańcuch `s` jest dłuższy od `n`, proces powielania kończy się w pozycji `n`, a funkcja dołącza na koniec skopiowanego łańcucha znak pusty. Funkcje te oferują te same korzyści co funkcja `alloca()`. Powielony łańcuch zostaje automatycznie zwolniony, gdy wywołująca funkcja kończy swoje działanie.

POSIX nie definiuje funkcji `alloca()`, `strdupa()` i `strndupa()`, a w innych systemach operacyjnych występują one sporadycznie. Jeśli należy zapewnić przenośność programu, wówczas użycie tych funkcji jest odradzane. W Linuksie wspomniane funkcje działają jednak całkiem dobrze i mogą zapewnić znakomitą poprawę wydajności, zamieniając skomplikowane czynności, związane z przydziałem pamięci dynamicznej, na zaledwie przesunięcie wskaźnika stosu.

Tablice o zmiennej długości

Standard C99 wprowadził *tablice o zmiennej długości* (ang. *variable-length arrays*, w skrócie *VLA*), których rozmiar ustalany jest podczas działania programu, a nie w czasie jego kompilacji. Kompilator GNU dla języka C wspierał takie tablice już od jakiegoś czasu, lecz odkąd standard C99 formalnie je zdefiniował, pojawił się istotny bodziec, aby ich używać. Podczas użycia tablic o zmiennej długości unika się przydzielania pamięci dynamicznej w taki sam sposób, jak podczas stosowania funkcji `alloca()`.

Sposób użycia łańcuchów o zmiennej długości jest dokładnie taki, jak się oczekuje:

```
for (i = 0; i < n; ++i)
{
   char foo[i + 1];
   /* tu można użyć 'foo'... */
}
```

W powyższym fragmencie kodu zmienna `foo` jest łańcuchem znaków o różnej długości, równej `i + 1`. Podczas każdej iteracji w pętli zostaje dynamicznie utworzona zmienna `foo`, a następnie automatycznie zwolniona, gdy znajdzie się poza zakresem widoczności. Gdyby zamiast łańcuchów o zmiennej długości użyto funkcji `alloca()`, pamięć nie zostałaby zwolniona, dopóki funkcja nie zakończyłaby swojego działania. Użycie łańcuchów o zmiennej długości zapewnia, że pamięć zostanie zwolniona podczas każdej iteracji w pętli. Dlatego też użycie takich łańcuchów zużywa w najgorszym razie `n` bajtów pamięci, podczas gdy użycie funkcji `alloca()` wykorzystywałoby `n*(n+1)/2` bajtów.

Funkcja `open_sysconf()` może zostać obecnie ponownie napisana, wykorzystując do jej implementacji łańcuch znaków o zmiennej długości:

```
int open_sysconf (const char *file, int flags, int mode)
{
   const char *etc = SYSCONF_DIR; /* "/etc/" */
   char name[strlen (etc) + strlen (file) + 1];

   strcpy (name, etc);
   strcat (name, file);

   return open (name, flags, mode);
}
```

Podstawową różnicą między użyciem funkcji `alloca()`, a użyciem tablic o zmiennej długości jest to, iż pamięć otrzymana przy użyciu tej pierwszej metody istnieje w czasie wykonywania funkcji, natomiast pamięć uzyskana przy użyciu drugiej metody istnieje do momentu, gdy zmienna, która ją reprezentuje, znajdzie się poza zakresem widoczności. Może się to zdarzyć, zanim funkcja zakończy swoje działanie — będąc cechą pozytywną lub negatywną. W przypadku pętli `for`, która została zastosowana w powyższym przykładzie, odzyskiwanie pamięci przy każdej iteracji zmniejsza realne zużycie pamięci bez żadnych efektów ubocznych (do

wykonania programu nie była potrzebna dodatkowa pamięć). Jeśli jednakże z pewnych powodów wymagane jest, aby przydzielona pamięć była dostępna dłużej niż tylko przez pojedynczą iterację pętli, wówczas bardziej sensowne jest użycie funkcji `alloca()`.

Łączenie wywołania funkcji `alloca()` oraz użycia tablic o zmiennej długości w jednym miejscu programu może powodować zaskakujące efekty. Należy postępować rozsądnie i używać tylko jednej z tych dwóch opcji w tworzonych funkcjach.

Wybór mechanizmu przydzielania pamięci

Wiele opcji przydzielania pamięci, omówionych w tym rozdziale, może być powodem powstania pytania o to, jakie rozwiązanie jest najbardziej odpowiednie dla danej czynności. W większości sytuacji użycie funkcji `malloc()` zaspokaja wszystkie potrzeby programisty. Czasami jednak inny sposób działania pozwala na uzyskanie lepszych wyników. Tabela 8.2. przedstawia informacje pomagające wybrać mechanizm przydzielania pamięci.

Tabela 8.2. Sposoby przydzielania pamięci w Linuksie

Sposób przydzielania pamięci	Zalety	Wady
Funkcja `malloc()`	Prosta, łatwa, powszechnie używana.	Pamięć zwracana nie musi być wypełniona zerami.
Funkcja `calloc()`	Prosta metoda przydzielania pamięci dla tablic, pamięć zwracana wypełniona jest zerami.	Dziwny interfejs w przypadku, gdy pamięć musi zostać przydzielona dla innych struktur danych niż tablice.
Funkcja `realloc()`	Zmienia wielkość istniejących obszarów przydzielonej pamięci.	Użyteczna wyłącznie dla operacji zmiany wielkości istniejących obszarów przydzielonej pamięci.
Funkcje `brk()` i `sbrk()`	Pozwala na szczegółową kontrolę działania sterty.	Zbyt niskopoziomowa dla większości użytkowników.
Anonimowe odwzorowania w pamięci	Łatwe w obsłudze, współdzielone, pozwalają projektantowi na ustalanie poziomu zabezpieczeń oraz dostarczania porady; optymalne rozwiązanie dla dużych przydziałów pamięci.	Niezbyt pasujące do niewielkich przydziałów pamięci; funkcja `malloc()` w razie potrzeby automatycznie używa anonimowych odwzorowań w pamięci.
Funkcja `posix_memalign()`	Przydziela pamięć wyrównaną do dowolnej, rozsądnej wartości.	Stosunkowo nowa, dlatego też jej przenośność jest dyskusyjna; użycie ma sens dopiero wówczas, gdy wyrównanie ma duże znaczenie.
Funkcje `memalign()` i `valloc()`	Bardziej popularna w innych systemach uniksowych niż funkcja `posix_memalign()`.	Nie jest zdefiniowana przez POSIX, oferuje mniejsze możliwości kontroli wyrównania niż `posix_memalign()`.
Funkcja `alloca()`	Bardzo szybki przydział pamięci, nie ma potrzeby, aby po użyciu jawnie ją zwalniać; bardzo dobra w przypadku niewielkich przydziałów pamięci.	Brak możliwości informowania o błędach, niezbyt dobra w przypadku dużych przydziałów pamięci, błędne działanie w niektórych systemach uniksowych.
Tablice o zmiennej długości	Podobnie jak `alloca()`, lecz pamięć zostanie zwolniona, gdy tablica znajdzie się poza zasięgiem widoczności, a nie podczas powrotu z funkcji.	Metoda użyteczna jedynie dla tablic; w niektórych sytuacjach może być preferowany sposób zwalniania pamięci, charakterystyczny dla funkcji `alloca()`; metoda mniej popularna w innych systemach uniksowych niż użycie funkcji `alloca()`.

Wreszcie, nie należy zapominać o alternatywie dla wszystkich powyższych opcji, czyli o automatycznym i statycznym przydziale pamięci. Przydzielanie obszarów dla zmiennych automatycznych na stosie lub dla zmiennych globalnych na stercie jest często łatwiejsze i nie wymaga obsługi wskaźników oraz troski o prawidłowe zwolnienie pamięci.

Operacje na pamięci

Język C dostarcza zbioru funkcji pozwalających bezpośrednio operować na obszarach pamięci. Funkcje te działają w wielu przypadkach w sposób podobny do interfejsów służących do obsługi łańcuchów znakowych, takich jak `strcmp()` i `strcpy()`, lecz używana jest w nich wartość rozmiaru bufora dostarczonego przez użytkownika, zamiast zakładania, że łańcuchy są zakończone znakiem zerowym. Należy zauważyć, że żadna z tych funkcji nie może zwrócić błędu. Zabezpieczenie przed powstaniem błędu jest zadaniem dla programisty — jeśli do funkcji przekazany zostanie wskaźnik do niepoprawnego obszaru pamięci, rezultatem jej wykonania nie będzie nic innego, jak tylko błąd segmentacji!

Ustawianie wartości bajtów

Wśród zbioru funkcji modyfikujących zawartość pamięci, najczęściej używana jest prosta funkcja `memset()`:

```
#include <string.h>

void * memset (void *s, int c, size_t n);
```

Wywołanie funkcji `memset()` ustawia `n` bajtów na wartość `c`, poczynając od adresu przekazanego w parametrze `s`, a następnie zwraca wskaźnik do zmienionego obszaru `s`. Funkcji używa się często, aby wypełnić dany obszar pamięci zerami:

```
/* wypełnij zerami obszar [s,s+256) */
memset (s, '\0', 256);
```

Funkcja `bzero()` jest starszym i niezalecanym interfejsem, wprowadzonym w systemie BSD w celu wykonania tej samej czynności. W nowym kodzie powinna być używana funkcja `memset()`, lecz Linux udostępnia `bzero()` w celu zapewnienia przenośności oraz wstecznej kompatybilności z innymi systemami:

```
#include <strings.h>

void bzero (void *s, size_t n);
```

Poniższe wywołanie jest identyczne z poprzednim użyciem funkcji `memset()`:

```
bzero(s, 256);
```

Należy zwrócić uwagę na to, że funkcja `bzero()`, podobnie jak inne interfejsy, których nazwy zaczynają się od litery `b`, wymaga dołączenia pliku nagłówkowego `<strings.h>`, a nie `<string.h>`.

Porównywanie bajtów

Podobnie jak ma to miejsce w przypadku użycia funkcji `strcmp()`, funkcja `memcmp()` porównuje dwa obszary pamięci, aby sprawdzić, czy są one identyczne:

Nie należy używać funkcji `memset()`, jeśli można użyć funkcji `calloc()`! Należy unikać przydzielania pamięci za pomocą funkcji `malloc()`, a następnie bezpośredniego wypełniania jej zerami przy użyciu funkcji `memset()`. Mimo że uzyska się takie same wyniki, dużo lepsze będzie użycie pojedynczego wywołania funkcji `calloc()`, która zwraca pamięć wypełnioną zerami. Nie tylko zaoszczędzi się na jednym wywołaniu funkcji, ale dodatkowo wywołanie `calloc()` będzie mogło otrzymać od jądra odpowiednio przygotowany obszar pamięci. W tym przypadku następuje uniknięcie ręcznego wypełniania bajtów zerami i poprawa wydajności.

```
#include <string.h>

int memcmp (const void *s1, const void *s2, size_t n);
```

Wywołanie tej funkcji powoduje porównanie pierwszych `n` bajtów dla obszarów pamięci `s1` i `s2` oraz zwraca `0`, jeśli bloki pamięci są sobie równe, wartość mniejszą od zera, jeśli `s1` jest mniejszy od `s2` oraz wartość większą od zera, jeśli `s1` jest większy od `s2`.

System BSD ponownie udostępnia niezalecany już interfejs, który realizuje w dużym stopniu to samo zadanie:

```
#include <strings.h>

int bcmp (const void *s1, const void *s2, size_t n);
```

Wywołanie funkcji `bmcp()` powoduje porównanie pierwszych `n` bajtów dla obszarów pamięci `s1` i `s2`, zwracając `0`, jeśli bloki są sobie równe lub wartość niezerową, jeśli są różne.

Z powodu istnienia wypełnienia struktur (opisanego wcześniej w podrozdziale Inne zagadnienia związane z wyrównaniem), porównywanie ich przy użyciu funkcji `memcmp()` lub `bcmp()` jest niepewne. W obszarze wypełnienia może istnieć niezainicjalizowany fragment nieużytecznych danych powodujący powstanie różnic podczas porównywania dwóch egzemplarzy danej struktury, które poza tym są sobie równe. Zgodnie z tym, poniższy kod nie jest bezpieczny:

```
/* czy dwa egzemplarze struktury dinghy są sobie równe? (BŁĘDNY KOD) */
int compare_dinghies (struct dinghy *a, struct dinghy *b)
{
   return memcmp (a, b, sizeof (struct dinghy));
}
```

Zamiast stosować powyższe, błędne rozwiązanie, programiści, którzy muszą porównywać ze sobą struktury, powinni czynić to dla każdego elementu struktury osobno. Ten sposób pozwala na uzyskanie pewnej optymalizacji, lecz wymaga większego wysiłku niż niepewne użycie prostej funkcji `memcmp()`. Oto poprawny kod:

```
/* czy dwa egzemplarze struktury dinghy są sobie równe? */
int compare_dinghies (struct dinghy *a, struct dinghy *b)
{
   int ret;

   if (a->nr_oars < b->nr_oars)
      return -1;
   if (a->nr_oars > b->nr_oars)
      return 1;

   ret = strcmp (a->boat_name, b->boat_name);
   if (ret)
      return ret;

   /* i tak dalej, dla każdego pola struktury... */
}
```

Przenoszenie bajtów

Funkcja `memmove()` kopiuje pierwszych `n` bajtów z obszaru pamięci `src` do `dst`, a następnie zwraca wskaźnik do `dst`:

```
#include <string.h>

void * memmove (void *dst, const void *src, size_t n);
```

System BSD ponownie udostępnia niezalecany już interfejs, który wykonuje tę samą czynność:

```
#include <strings.h>

void bcopy (const void *src, void *dst, size_t n);
```

Należy zwrócić uwagę na to, że mimo iż obie funkcje używają takich samych parametrów, kolejność dwóch pierwszych jest zmieniona w `bcopy()`.

Obie funkcje `bcopy()` oraz `memmove()` mogą bezpiecznie obsługiwać nakładające się obszary pamięci (na przykład, gdy część obszaru `dst` znajduje się wewnątrz `src`). Dzięki temu bajty w pamięci mogą przykładowo zostać przesunięte w stronę wyższych lub niższych adresów wewnątrz danego regionu. Ponieważ taka sytuacja jest rzadkością, a programista wiedziałby, jeśliby miała ona miejsce, dlatego też standard języka C definiuje wariant funkcji `memmove()`, który nie wspiera nakładających się rejonów pamięci. Ta wersja może działać potencjalnie szybciej:

```
#include <string.h>

void * memcpy (void *dst, const void *src, size_t n);
```

Powyższa funkcja działa identycznie jak `memmove()`, za wyjątkiem tego, że obszary `dst` i `src` nie mogą posiadać wspólnej części. Jeśli tak jest, rezultat wykonania funkcji jest niezdefiniowany.

Inną funkcją, wykonującą bezpieczne kopiowanie pamięci, jest `memccpy()`:

```
#include <string.h>

void * memccpy (void *dst, const void *src, int c, size_t n);
```

Funkcja `memccpy()` działa tak samo jak `memcpy()`, za wyjątkiem tego, że zatrzymuje proces kopiowania, jeśli wśród pierwszych `n` bajtów obszaru `src` zostanie odnaleziony bajt o wartości `c`. Funkcja zwraca wskaźnik do następnego bajta, występującego po `c` w obszarze `dst` lub `NULL`, jeśli `c` nie odnaleziono.

Ostatecznie funkcja `mempcpy()` pozwala poruszać się po pamięci:

```
#define _GNU_SOURCE
#include <string.h>

void * mempcpy (void *dst, const void *src, size_t n);
```

Funkcja `mempcpy()` działa tak samo jak `memcpy()`, za wyjątkiem tego, że zwraca wskaźnik do miejsca znajdującego się w pamięci za ostatnim skopiowanym bajtem. Jest to przydatne, gdy zbiór danych należy skopiować do następujących po sobie obszarów pamięci — nie stanowi to jednak zbyt dużego usprawnienia, ponieważ wartość zwracana jest zaledwie równa `dst + n`. Funkcja ta jest specyficzna dla GNU.

Wyszukiwanie bajtów

Funkcje `memchr()` oraz `memrchr()` wyszukują dany bajt w bloku pamięci:

```
#include <string.h>

void * memchr (const void *s, int c, size_t n);
```

Funkcja memchr() przeszukuje obszar pamięci o wielkości n bajtów, wskazywany przez parametr s, aby odnaleźć w nim znak c, który jest interpretowany jako typ unsigned char. Funkcja zwraca wskaźnik do miejsca w pamięci, w którym znajduje się bajt pasujący do parametru c. Jeśli wartość c nie zostanie odnaleziona, funkcja zwróci NULL.

Funkcja memrchr() działa tak samo jak funkcja memchr(), za wyjątkiem tego, że przeszukuje obszar pamięci o wielkości n bajtów, wskazywany przez parametr s, rozpoczynając od jego końca zamiast od początku:

```
#define _GNU_SOURCE
#include <string.h>

void * memrchr (const void *s, int c, size_t n);
```

W przeciwieństwie do memchr(), funkcja memrchr() jest rozszerzeniem GNU i nie należy do standardu języka C.

Aby przeprowadzać bardziej skomplikowane operacje wyszukiwania, można użyć funkcji o dziwnej nazwie memmem(), przeszukującej blok pamięci w celu odnalezienia dowolnego łańcucha bajtów:

```
#define _GNU_SOURCE
#include <string.h>

void * memmem (const void *haystack, size_t haystacklen, const void *needle,
  size_t needlelen);
```

Funkcja memmem() zwraca wskaźnik do pierwszego miejsca wystąpienia łańcucha bajtów needle o długości needlelen, wyrażonej w bajtach. Przeszukiwany obszar pamięci wskazywany jest przez parametr haystack i posiada długość haystacklen bajtów. Jeśli funkcja nie odnajdzie łańcucha needle w haystack, zwraca NULL. Jest również rozszerzeniem GNU.

Manipulowanie bajtami

Biblioteka języka C dla Linuksa dostarcza interfejsu, który pozwala na wykonywanie trywialnej operacji kodowania bajtów:

```
#define _GNU_SOURCE
#include <string.h>

void * memfrob (void *s, size_t n);
```

Wywołanie funkcji memfrob() koduje pierwszych n bajtów z obszaru pamięci wskazywanego przez s. Polega to na przeprowadzeniu dla każdego bajta operacji binarnej różnicy symetrycznej (XOR) z liczbą 42. Funkcja zwraca wskaźnik do zmodyfikowanego obszaru s.

Aby przywrócić pierwotną zawartość zmodyfikowanego obszaru pamięci, należy dla niego ponownie wywołać funkcję memfrob(). Dlatego też wykonanie poniższego fragmentu kodu nie powoduje żadnych zmian w obszarze secret:

```
memfrob (memfrob (secret, len), len);
```

Funkcja ta nie jest jednak żadną prawdziwą (ani nawet okrojoną) namiastką operacji szyfrowania; ograniczona jest jedynie do wykonania trywialnego zaciemnienia bajtów. Jest specyficzna dla GNU.

Blokowanie pamięci

W Linuksie zaimplementowano operację *stronicowania na żądanie*, która polega na tym, że strony pobierane są z dysku w razie potrzeby, natomiast zapisywane na dysku, gdy nie są już używane. Dzięki temu nie istnieje bezpośrednie powiązanie wirtualnych przestrzeni adresowych dla procesów w systemie z całkowitą ilością pamięci fizycznej, gdyż istnienie obszaru wymiany na dysku dostarcza wrażenia posiadania prawie nieskończonej ilości tejże pamięci.

Wymiana stron wykonywana jest w sposób przezroczysty, a aplikacje w zasadzie nie muszą „interesować się" (ani nawet znać) sposobem działania stronicowania, przeprowadzanym przez jądro Linuksa. Istnieją jednak dwie sytuacje, podczas których aplikacje mogą wpływać na sposób działania stronicowania systemowego:

Determinizm

Aplikacje, posiadające ograniczenia czasowe, wymagają deterministycznego zachowania. Jeśli pewne operacje dostępu do pamięci kończą się błędami stron (co wywołuje powstawanie kosztownych operacji wejścia i wyjścia), wówczas aplikacje te mogą przekraczać swoje parametry ograniczeń czasowych. Aby zapewnić, że wymagane strony będą zawsze znajdować się w pamięci fizycznej i nigdy nie zostaną wyrzucone na dysk, można dla danej aplikacji zagwarantować, że dostęp do pamięci nie zakończy się błędem, co pozwoli na spełnienie warunków spójności i determinizmu, a również na poprawę jej wydajności.

Bezpieczeństwo

Jeśli w pamięci przechowywane są tajne dane prywatne, wówczas poziom bezpieczeństwa może zostać naruszony po wykonaniu operacji stronicowania i zapisaniu tych danych w postaci niezaszyfrowanej na dysku. Na przykład, jeśli prywatny klucz użytkownika jest zwykle przechowywany na dysku w postaci zaszyfrowanej, wówczas jego odszyfrowana kopia, znajdująca się w pamięci, może zostać wyrzucona do pliku wymiany. W przypadku środowiska o wysokim poziomie bezpieczeństwa, zachowanie to może być niedopuszczalne. Dla aplikacji wymagających zapewnienia dużego poziomu bezpieczeństwa, można zdefiniować, że obszar, w którym znajduje się odszyfrowany klucz, będzie istniał wyłącznie w pamięci fizycznej.

Oczywiście zmiana zachowania jądra może spowodować pogorszenie ogólnej sprawności systemu. Dla danej aplikacji nastąpi poprawa determinizmu oraz bezpieczeństwa, natomiast gdy jej strony będą zablokowane w pamięci, strony innej aplikacji będą wyrzucane na dysk. Jądro (jeśli można ufać metodzie jego zaprojektowania) zawsze optymalnie wybiera taką stronę, która powinna zostać wyrzucona na dysk (to znaczy stronę, która najprawdopodobniej nie będzie używana w przyszłości), dlatego też po zmianie jego zachowania wybór ten nie będzie już optymalny.

Blokowanie fragmentu przestrzeni adresowej

POSIX 1003.1b-1993 definiuje dwa interfejsy pozwalające na „zamknięcie" jednej lub więcej stron w pamięci fizycznej, dzięki czemu można zapewnić, że nie zostaną one nigdy wyrzucone na dysk. Pierwsza funkcja blokuje pamięć dla danego przedziału adresów:

```
#include <sys/mman.h>

int mlock (const void *addr, size_t len);
```

Wywołanie funkcji mlock() blokuje w pamięci fizycznej obszar pamięci wirtualnej, rozpoczynający się od adresu addr i posiadający wielkość len bajtów. W przypadku sukcesu, funkcja zwraca wartość 0. W przypadku błędu zwraca −1 oraz odpowiednio ustawia zmienną errno.

Poprawne wywołanie funkcji blokuje w pamięci wszystkie strony fizyczne, których adresy zawierają się w zakresie [addr, addr + len). Na przykład, jeśli funkcja chce zablokować tylko jeden bajt, wówczas w pamięci zostanie zablokowana cała strona, w której on się znajduje. Standard POSIX definiuje, że adres addr powinien być wyrównany do wielkości strony. Linux nie wymusza tego zachowania i w razie potrzeby niejawnie zaokrągla adres addr w dół do najbliższej strony. W przypadku programów, dla których wymagane jest zachowanie warunku przenośności do innych systemów, należy jednak upewnić się, że addr jest wyrównany do granicy strony.

Poprawne wartości zmiennej errno obejmują poniższe kody błędów:

EINVAL
: Parametr len ma wartość ujemną.

ENOMEM
: Proces wywołujący zamierzał zablokować więcej stron, niż wynosi ograniczenie zasobów RLIMIT_MEMLOCK (szczegóły w podrozdziale Ograniczenia blokowania).

EPERM
: Wartość ograniczenia zasobów RLIMIT_MEMLOCK była równa zeru, lecz proces nie posiadał uprawnienia CAP_IPC_LOCK (podobnie, szczegóły w podrozdziale Ograniczenia blokowania).

Podczas wykonywania funkcji fork(), proces potomny nie dziedziczy pamięci zablokowanej. Dzięki istnieniu mechanizmu kopiowania podczas zapisu, używanego dla przestrzeni adresowych w Linuksie, strony procesu potomnego są skutecznie zablokowane w pamięci, dopóki potomek nie wykona dla nich operacji zapisu.

Załóżmy przykładowo, że pewien program przechowuje w pamięci odszyfrowany łańcuch znaków. Proces może za pomocą kodu, podobnego do poniżej przedstawionego, zablokować stronę zawierającą dany łańcuch:

```
int ret;

/* zablokuj łańcuch znaków 'secret' w pamięci */
ret = mlock (secret, strlen (secret));
if (ret)
   perror ("mlock");
```

Blokowanie całej przestrzeni adresowej

Jeśli proces wymaga zablokowania całej przestrzeni adresowej w pamięci fizycznej, wówczas użycie funkcji mlock() staje się niewygodne. Aby zrealizować to zadanie — powszechnie wykonywane w przypadku aplikacji czasu rzeczywistego — standard POSIX definiuje funkcję systemową, która blokuje całą przestrzeń adresową:

```
#include <sys/mman.h>

int mlockall (int flags);
```

Wywołanie funkcji mlockall() blokuje w pamięci fizycznej wszystkie strony przestrzeni adresowej dla aktualnego procesu. Parametr flags steruje zachowaniem funkcji i jest równy sumie bitowej poniższych znaczników:

MCL_CURRENT
: Jeśli znacznik jest ustawiony, powoduje to, że funkcja mlockall() blokuje wszystkie aktualnie odwzorowane strony w przestrzeni adresowej procesu. Stronami takimi może być stos, segment danych, pliki odwzorowane itd.

MCL_FUTURE
: Jeśli znacznik jest ustawiony, wówczas wykonanie funkcji mlockall() zapewnia, iż wszystkie strony, które w przyszłości zostaną odwzorowane w przestrzeni adresowej, będą również zablokowane w pamięci.

Większość aplikacji używa obu tych znaczników jednocześnie.

W przypadku sukcesu funkcja zwraca wartość 0. W przypadku błędu zwraca −1 oraz odpowiednio ustawia zmienną errno na jedną z poniższych wartości:

EINVAL
: Parametr flags ma wartość ujemną.

ENOMEM
: Proces wywołujący zamierzał zablokować więcej stron, niż wynosi ograniczenie zasobów RLIMIT_MEMLOCK (szczegóły w podrozdziale Ograniczenia blokowania).

EPERM
: Wartość ograniczenia zasobów RLIMIT_MEMLOCK była równa zeru, lecz proces nie posiadał uprawnienia CAP_IPC_LOCK (podobnie, szczegóły w podrozdziale Ograniczenia blokowania).

Odblokowywanie pamięci

Aby umożliwić odblokowanie stron z pamięci fizycznej, pozwalając jądru w razie potrzeby ponownie wyrzucać je na dysk, POSIX definiuje dwa dodatkowe interfejsy:

```
#include <sys/mman.h>

int munlock (const void *addr, size_t len);
int munlockall (void);
```

Funkcja systemowa munlock() odblokowuje strony, które rozpoczynają się od adresu addr i zajmują obszar len bajtów. Jest ona przeciwieństwem funkcji mlock(). Funkcja systemowa munlockall() jest przeciwieństwem mlockall(). Obie funkcje zwracają zero w przypadku sukcesu, natomiast w przypadku niepowodzenia zwracają −1 oraz ustawiają zmienną errno na jedną z poniższych wartości:

EINVAL
: Parametr len jest nieprawidłowy (tylko dla munlock()).

ENOMEM
: Niektóre z podanych stron są nieprawidłowe.

EPERM
: Wartość ograniczenia zasobów RLIMIT_MEMLOCK była równa zeru, lecz proces nie posiadał uprawnienia CAP_IPC_LOCK (szczegóły w następnym podrozdziale Ograniczenia blokowania).

Blokady pamięci nie zagnieżdżają się. Dlatego też, bez względu na to, ile razy dana strona została zablokowana za pomocą funkcji mlock() lub mlockall(), pojedyncze wywołanie funkcji munlock() lub munlockall() spowoduje jej odblokowanie.

Ograniczenia blokowania

Ponieważ blokowanie pamięci może spowodować spadek wydajności systemu (faktycznie, jeśli zbyt wiele stron zostanie zablokowanych, operacje przydziału pamięci mogą się nie powieść), dlatego też w systemie Linux zdefiniowano ograniczenia, które określają, ile stron może zostać zablokowanych przez jeden proces.

Proces, który posiada uprawnienie CAP_IPC_LOCK, może zablokować dowolną liczbę stron w pamięci. Procesy nieposiadające takiego uprawnienia, mogą zablokować wyłącznie tyle bajtów pamięci, ile wynosi ograniczenie RLIMIT_MEMLOCK. Domyślnie, ograniczenie to wynosi 32 kB — jest ono wystarczające, aby zablokować jeden lub dwa tajne klucze w pamięci, lecz nie tak duże, aby skutecznie wpłynąć na wydajność systemu (w rozdziale 6. omówiono ograniczenia zasobów oraz metody pozwalające na pobieranie i ustawianie tych parametrów).

Czy strona znajduje się w pamięci fizycznej?

Aby ułatwić uruchamianie programów oraz usprawnić diagnostykę, Linux udostępnia funkcję mincore(), która może zostać użyta, by ustalić, czy obszar danych znajduje się w pamięci fizycznej lub w pliku wymiany na dysku:

```
#include <unistd.h>
#include <sys/mman.h>

int mincore (void *start, size_t length, unsigned char *vec);
```

Wywołanie funkcji mincore() zwraca wektor bajtów, który opisuje, jakie strony odwzorowania znajdują się w pamięci fizycznej w czasie jej użycia. Funkcja zwraca wektor poprzez parametr vec oraz opisuje strony rozpoczynające się od adresu start (który musi być wyrównany do granicy strony) i obejmujące obszar o wielkości length bajtów (który nie musi być wyrównany do granicy strony). Każdy element w wektorze vec odpowiada jednej stronie z dostarczonego zakresu adresów, poczynając od pierwszego bajta opisującego pierwszą stronę i następnie przechodząc w sposób liniowy do kolejnych stron. Zgodnie z tym, wektor vec musi być na tyle duży, aby przechować odpowiednią liczbę bajtów, równą wyrażeniu (length - 1 + *rozmiar strony*)/*rozmiar strony*. Najmniej znaczący bit w każdym bajcie wektora równy jest 1, gdy strona znajduje się w pamięci fizycznej lub 0, gdy jej tam nie ma. Inne bity są obecnie niezdefiniowane i zarezerwowane do przyszłego wykorzystania.

W przypadku sukcesu funkcja zwraca 0. W przypadku błędu zwraca −1 oraz odpowiednio ustawia zmienną errno na jedną z poniższych wartości:

EAGAIN
: Brak wystarczających zasobów jądra, aby zakończyć tę operację.

EFAULT
: Parametr vec wskazuje na błędny adres.

EINVAL
: Parametr start nie jest wyrównany do granicy strony.

`ENOMEM`
: Obszar `[start, start + length)` zawiera pamięć, która nie jest częścią odwzorowania opartego na pliku.

Ta funkcja systemowa działa obecnie poprawnie jedynie dla odwzorowań opartych na plikach i utworzonych za pomocą opcji `MAN_SHARED`. Bardzo ogranicza to jej zakres użycia.

Przydział oportunistyczny

W systemie Linux używana jest strategia *przydziału oportunistycznego*. Gdy proces żąda przydzielenia mu dodatkowej pamięci z jądra — na przykład, poprzez zwiększenie segmentu danych lub stworzenie nowego odwzorowania w pamięci — wówczas jądro *zatwierdza* przyjęcie zlecenia na przydział pamięci, ale bez rzeczywistego dostarczenia dodatkowego fizycznego miejsca. Dopiero wówczas, gdy proces wykonuje operację zapisu dla nowo przydzielonej pamięci, jądro *realizuje* przydział poprzez zamianę zlecenia na fizyczne udostępnienie pamięci. Strategia ta jest zaimplementowana dla każdej strony z osobna, a jądro wykonuje wymagane operacje stronicowania oraz kopiowania podczas zapisu tylko w razie potrzeby.

Zachowanie to ma dużo zalet. Po pierwsze, strategia leniwego przydziału pamięci pozwala jądru przesuwać wykonywanie czynności na ostatni dopuszczalny moment, jeśli w ogóle zaistnieje potrzeba realizacji operacji przydziału. Po drugie, ponieważ żądania realizowane są dla każdej strony z osobna i wyłącznie w razie potrzeby, dlatego też tylko ta pamięć, która jest rzeczywiście używana, wykorzystuje zasoby fizyczne. Wreszcie, ilość pamięci zatwierdzonej może być dużo większa od ilości pamięci fizycznej, a nawet od dostępnego obszaru wymiany. Ta ostatnia cecha zwana jest *przekroczeniem zakresu zatwierdzenia* (ang. *overcommitment*).

Przekroczenie zakresu zatwierdzenia oraz stan braku pamięci (OOM)

Przekroczenie zakresu zatwierdzenia pozwala systemom na uruchamianie dużo większej liczby obszerniejszych aplikacji, niż byłoby to możliwe, gdyby każda żądana strona pamięci otrzymywała odwzorowanie w zasobie fizycznym w momencie jej przydziału zamiast w momencie użycia. Bez mechanizmu przekraczania zakresu zatwierdzenia, wykonanie odwzorowania pliku o wielkości 2 GB przy użyciu kopiowania podczas zapisu, wymagałoby od jądra przydzielenia 2 GB pamięci fizycznej. Dzięki mechanizmowi przekraczania zakresu zatwierdzenia, odwzorowanie pliku 2 GB wymaga przydzielenia pamięci fizycznej jedynie dla poszczególnych stron z danymi, które są faktycznie zapisywane przez proces. Ponadto, bez użycia mechanizmu przekraczania zakresu zatwierdzenia, każde wywołanie funkcji `fork()` wymagałoby dostarczenia odpowiednio dużego obszaru wolnej pamięci, aby móc skopiować przestrzeń adresową, nawet gdyby większość stron nie zostało poddanych operacji kopiowania podczas zapisu.

Co stanie się jednak, gdy procesy przystąpią do realizacji zaległych przydziałów, których sumaryczna wielkość przekroczy rozmiar pamięci fizycznej i obszaru wymiany? W tym przypadku jedna lub więcej realizacji przydziału zakończy się niepowodzeniem. Ponieważ jądro zrealizowało już przydział pamięci — wykonanie funkcji systemowej, żądającej przeprowadzenia tej operacji, zakończyło się sukcesem — a proces właśnie przystępuje do użycia udostępnionej pamięci, dlatego też jedyną dostępną opcją jądra jest przerwanie działania tego procesu i zwolnienie zajętej przez niego pamięci.

Gdy przekroczenie zakresu zatwierdzenia powoduje pojawienie się niewystarczającej ilości pamięci, aby zatwierdzić zrealizowane żądanie, wówczas sytuację taką nazywa się *stanem braku pamięci* (ang. *out of memory*, w skrócie *OOM*). W odpowiedzi na taką sytuację jądro uruchamia *zabójcę stanu braku pamięci* (ang. *OOM killer*), aby wybrał proces, który „nadaje się" do usunięcia. W tym celu jądro próbuje odnaleźć najmniej ważny proces, zużywający największą ilość pamięci.

Stany braku pamięci występują rzadko, dlatego też odnosi się duże korzyści z zezwolenia na przekraczanie zakresu zatwierdzenia. Na pewno jednak pojawienie się takiego stanu nie jest mile widziane, a niedeterministyczne przerwanie działania procesu przez zabójcę OOM jest często nie do zaakceptowania.

W systemach, których to dotyczy, jądro pozwala na zablokowanie przekraczania zakresu zatwierdzenia przy użyciu pliku */proc/sys/vm/overcommit_memory* oraz analogicznego parametru *sysctl* o nazwie `vm.overcommit_memory`.

Domyślna wartość tego parametru równa jest zeru i nakazuje ona jądru, aby realizował heurystyczną strategię przekraczania zakresu zatwierdzenia, która pozwala na zatwierdzanie przydziałów pamięci w granicach rozsądku, nie dopuszczając jednak do realizacji wyjątkowo złych żądań. Wartość równa `1` pozwala na wykonanie wszystkich zatwierdzeń, podejmując ryzyko powstania stanu braku pamięci. Pewne aplikacje, wykorzystujące zasoby pamięci w intensywny sposób, takie jak programy naukowe, próbują wysyłać tak dużo żądań przydziału pamięci, które i tak nigdy nie będą musiały zostać zrealizowane, że użycie tej opcji ma w tym przypadku sens.

Wartość równa `2` całkowicie uniemożliwia przekraczanie zakresu zatwierdzenia i aktywuje *rozliczanie ścisłe* (ang. *strict accounting*). W tym trybie zatwierdzenia przyjęcia zleceń przydziału pamięci ograniczone są do wielkości obszaru wymiany oraz pewnego fragmentu pamięci fizycznej, którego względny rozmiar jest konfigurowalny. Rozmiar ten można ustawić za pomocą pliku */proc/sys/vm/overcommit_ratio* lub analogicznego parametru *sysctl*, zwanego `vm.overcommit_` `↪ratio`. Domyślną wartością jest liczba `50`, ograniczająca zatwierdzenia przyjęcia zleceń przydziału pamięci do wielkości obszaru wymiany i połowy pamięci fizycznej. Ponieważ pamięć fizyczna zawiera jądro, tablice stron, strony zarezerwowane przez system, strony zablokowane itd., dlatego też tylko jej fragment może być w rzeczywistości wyrzucany na dysk i realizować przydziały pamięci.

Rozliczenia ścisłego należy używać z rozwagą! Wielu projektantów systemowych, którzy są zniechęceni opiniami o zabójcy OOM, uważa, że użycie rozliczenia ścisłego spowoduje rozwiązanie ich problemów. Aplikacje często jednak wykonują mnóstwo niepotrzebnych przydziałów pamięci, które sięgają daleko poza obszar przekraczania zakresu zatwierdzenia. Akceptacja takiego zachowania była jednym z argumentów przemawiających za implementacją pamięci wirtualnej.