# PODSTAWY OBSŁUGI SYGNAŁÓW I ZARZĄDZANIA PAMIĘCIĄ

# Sygnały

- **Sygnały**
- Pamięć

# Sygnały

- **Sygnał**: wiadomość, którą proces może przesłać do procesu lub grupy procesów, jeśli tylko ma odpowiednie uprawnienia (zdarzenia asynchroniczne i wyjątki).
- Typ wiadomości reprezentowany jest przez **nazwę symboliczną**
- Dla każdego sygnału otrzymujący go proces może:
  1. Jawnie **zignorować** sygnał (bit w masce sygnałów w tablicy procesów nie jest ustawiany – brak śladu odebrania sygnału)
  2. Realizować specjalne działania za pomocą tzw. ***signal handler*** (np. funkcja użytkownika)
  3. W przeciwnym przypadku realizowane jest **działanie domyślne** (zwykle proces kończony)
- Wybrane sygnały mogą być również **blokowane**, tzn. ich nadejście jednorazowo jest odznaczane w masce sygnałów w tablicy procesów, ale obsługa jest odkładana do momentu zdjęcia blokady.

# Przykład sygnałów 1/3

- Po zakończeniu potomka wysyłany jest sygnał **SIGCHLD** do rodzica (wartość 20,17 lub 18 zależna od architektury – dla **i386** i **ppc** 17).
- Jeśli rodzic chce czekać na zakończenie potomka, to powiadamia system, że chce przechwycić sygnał **SIGCHLD**
- Jeśli nie zasygnalizuje oczekiwania, to sygnał **SIGCHLD** jest przez niego ignorowany (standardowa obsługa)
- W większości systemów dokładny opis obsługiwanych sygnałów wraz z ich domyślną obsługą i dodatkowymi informacjami umieszczany jest w **7 manualu signal** (**`man 7 signal`**)

# Przykład sygnałów 2/3

- **SIGINT** Sygnał przerwania generowany zwykle, gdy użytkownik naciśnie klawisz **przerwania** na terminalu.
- **SIGQUIT** Sygnał przerwania generowany gdy użytkownik naciśnie na terminalu klawisz zakończenia pracy. Sygnał **SIGQUIT** jest podobny do sygnału **SIGINT**, ale dodatkowo generuje **obraz pamięci**.
- **SIGKILL** Bezwarunkowe zakończenie procesu (proces odbierający ten sygnał **nie może** go ani zignorować, ani przechwycić).
- **SIGTSTP** Sygnał wysyłany do procesu po naciśnięciu klawisza zawieszenia (na ogół CRTL+Z) lub klawisza zawieszenia z opóźnieniem (CTRL+Y). Proces zawieszony (zatrzymany), można wznowić sygnałem SIGCONT.
- **SIGILL** Sygnał ten jest generowany po wystąpieniu wykrywanej sprzętowo sytuacji wyjątkowej, spowodowanej przez niewłaściwą implementację systemu.
- **SIGSTOP** Sygnał ten zatrzymuje proces. Podobnie jak sygnał **SIGKILL nie może** zostać zignorowany lub przechwycony. Działanie zatrzymanego procesu można wznowić sygnałem **SIGCONT**.

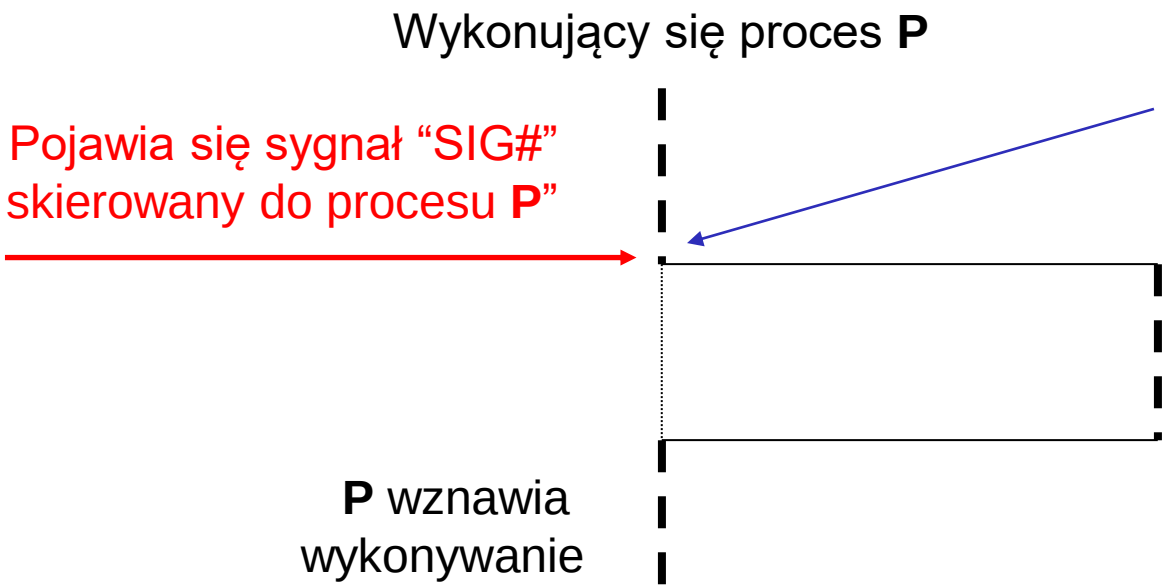
# Przykład sygnałów 3/3

- **SIGSEGV** Sygnał generowany po wystąpieniu **błędu sprzętowego** spowodowanego przez niewłaściwą implementację systemu. Sygnał naruszenia segmentacji pojawia się na ogół wtedy, kiedy proces odnosi się do takiego adresu w pamięci, do którego nie ma dostępu.
- **SIGALARM** Sygnał budzika generowany przez f-cję **`unsigned int alarm(unsigned int sec);`**
- **SIGCHLD** Sygnał wysyłany do procesu, kiedy jego potomny proces się skończył.
- **SIGUSR1, SIGUSR2** Sygnały definiowane przez użytkownika, których można używać do komunikacji między procesami.
- **SIGTERM** Domyślny sygnał wysyłany przez komendę **kill**. Wymusza zawieszenie procesu.
- **SIGHUP** Jest wysyłany przy zerwaniu łączności z terminalem do wszystkich procesów, dla których jest on terminalem sterującym. Domyślnie powoduje zakończenie procesu.

# Obsługa sygnałów

```
/* kod procesu P */
. . .
signal(SIG#, sig_hndlr);
. . .
/* DOWOLNY KOD */
```

```
void sig_hndlr(...) {
        /* DOWOLNY KOD */
}
```




tzw. **wolne funkcje** systemowe (w odróżnieniu od f-cji szybkich) mogą zostać przerwane i zwrócić błąd **EINTR**

**sig_hndlr** wykonuje się w przestrzeni adresowej procesu **P**

# Wolne funkcje systemowe

- „*Funkcje systemowe podzielono na funkcje wolne i szybkie. Z grubsza wolne funkcje systemowe to te, które mogą długo wstrzymywać proces (np. read z konsoli), a szybkie to te, które nie będą wstrzymywać procesów długo (read z pliku).*”
- Przykłady wolnych funkcji systemowych:
  - blokujące wywołanie **`read`** na pustym potoku
  - **`pause`**
  - **`wait`**
- Wolne funkcje systemowe są **przerywane** przez sygnały. W zależności od wersji systemu:
  - kończą się one wówczas zwracając **-1** i przekazując w wyniku błąd **EINTR**
  - lub po obsłudze sygnału są automatycznie wznawiane.
- POSIX nie określa, które z powyższych zachowań jest słuszne. W systemie Linux domyślne jest zachowanie pierwsze, możemy jednak zmienić je na drugie używając flagi **`SA_RESTART`** w funkcji **`sigaction`**.

# Wysyłanie sygnałów: `kill` 1/2

```
#include <sys/types.h>
#include <signal.h>

int kill(pid_t pid, int sig);
```

Funkcja systemowa kill może służyć do przesłania dowolnego sygnału do dowolnego procesu lub do dowolnej grupy procesów.

Jeśli ***pid*** ma **wartość dodatnią**, to sygnał ***sig*** jest przesyłany do procesu ***pid***.

Jeśli ***pid*** jest **równy 0**, to ***sig*** jest przesyłany do wszystkich procesów należących do tej samej grupy, co proces bieżący.

Jeśli ***pid*** jest **równy -1**, to sygnał jest przesyłany do wszystkich procesów, oprócz procesu nr 1 (init).

Jeśli ***pid*** jest **mniejszy niż -1**, to sygnał jest przesyłany do wszystkich procesów należących do grupy procesów o numerze ***-pid***.

# Wysyłanie sygnałów: `kill` 2/2

Linux pozwala procesowi wysłać sygnał **do samego siebie**, ale wywołanie **`kill(-1,sig)`** pod Linuksem nie powoduje wysłania sygnału do bieżącego procesu.

Aby proces miał prawo wysłać sygnał do procesu *pid* :

- Proces wysyłający musi mieć uprawnienia roota, albo
- **rzeczywisty lub efektywny ID użytkownika** procesu wysyłającego musi być równy rzeczywistemu ID lub zachowanemu set-UID procesu otrzymującego sygnał.

Inne funkcje: **raise**, **alarm**

# Zmiana obsługi sygnału: `signal` 1/2

```
#include <signal.h>

typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

Funkcja instaluje nową obsługę sygnału ***signum***. Obsługa sygnału ustawiana jest na ***handler***, który może być funkcją podaną przez użytkownika lub **SIG_IGN** albo **SIG_DFL**.

Po przyjściu sygnału do procesu:

- jeśli obsługa odpowiedniego sygnału została ustawiona na **SIG_IGN**, to sygnał jest ignorowany.
- jeśli obsługa została ustawiona na **SIG_DFL**, to podejmowana jest domyślna akcja skojarzona z sygnałem.
- jeśli jako obsługa sygnału została ustawiona funkcja ***sighandler*** to wywoływana jest funkcja sighandler z argumentem signum.

Sygnały **SIGKILL** i **SIGSTOP** nie mogą być ani przechwycone, ani zignorowane.

Funkcja zwraca poprzednią wartość obsługi sygnału, lub **SIG_ERR** w przypadku błędu.

# Zmiana obsługi sygnału: **signal** 2/2

test-signal-1.c

```
...
void (*f)( int);
f=signal(SIGINT,SIG_IGN); /* ignorowanie sygnału sigint*/
signal(SIGINT,f); /*przywrócenie poprzedniej reakcji na syg.*/
signal(SIGINT,SIG_DFL); /*ustaw. standardowej reakcji na syg.*/
...
```

test-signal-2.c

```
void moja_funkcja(int s) {
printf("Został przechwycony sygnał %d\n", s); return 0; }

main(){
  signal(SIGINT, moja_funkcja); /* przechwycenie sygnału */
  ...
}
```

# Zmiana obsługi sygnału: `sigaction` 1/4

```
#include <signal.h>
int sigaction(int signum, const struct sigaction *act, struct
sigaction *oldact);
```

Wywołanie systemowe używane do zmieniania akcji, którą obiera proces po odebraniu określonego sygnału. ***signum*** określa sygnał i może być dowolnym prawidłowym sygnałem poza **SIGKILL** i **SIGSTOP**. Jeśli ***act*** jest niezerowe, to nowa akcja dla sygnału ***signum*** jest brana z ***act***. Jeśli ***oldact*** też jest niezerowe, to poprzednia akcja jest w nim zachowywana.

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

# Zmiana obsługi sygnału: `sigaction` 2/4

**`sa_handler`** podaje akcję, związaną z sygnałem ***signum*** i może to być m.in **SIG_DFL** dla akcji domyślnej, **SIG_IGN** dla akcji ignorowania lub wskaźnik do funkcji obsługującej sygnał. Funkcja ta ma tylko jeden argument, w którym będzie przekazany numer sygnału.

**`sa_sigaction`** podaje akcję zamiast ***sa_handler*** jeżeli w ***sa_flags*** ustawiono **SA_SIGINFO**. Funkcja ta otrzymuje numer sygnału jako pierwszy argument, wskaźnik do ***siginfo_t*** jako drugi argument oraz wskaźnik do ***ucontext_t*** (zrzutowany na void *) jako jej trzeci argument.

**`sa_mask`** podaje maskę sygnałów, które powinny być blokowane podczas wywoływania handlera sygnałów. Dodatkowo, sygnał, który wywołał handler będzie zablokowany, chyba że użyto flagi **SA_NODEFER**.

**`sa_flags`** podaje zbiór flag, które modyfikują zachowanie procesu obsługi sygnałów. Jest to zbiór wartości połączonych bitowym OR (np. flaga **SA_RESETHAND** odtwórz akcję sygnałową do stanu domyślnego po wywołaniu handlera sygnałów a **SA_SIGINFO** określa, że handler sygnałów pobiera 3 argumenty, a nie jeden, natomiast **SA_RESTART** pozwala na automatyczne wznowienie przerwanej sygnałem **wolnej funkcji** systemowej)

# Zmiana obsługi sygnału: `sigaction` 3/4

Parametr ***siginfo_t*** z ***sa_sigaction*** jest strukturą zawierającą następujące elementy ( zależności od przechwyconego sygnału pola są różnie interpretowane oraz nie wszystkie zawierają dla każdego sygnału sensowne informacje):

```
siginfo_t {
            int      si_signo;     /* Signal number */
            int      si_errno;     /* An errno value */
            int      si_code;      /* Signal code */
            pid_t    si_pid;       /* Sending process ID */
            uid_t    si_uid;       /* Real user ID of sending process */
            int      si_status;    /* Exit value or signal */
            clock_t  si_utime;     /* User time consumed */
            clock_t  si_stime;     /* System time consumed */
            union sigval si_value; /* Signal value */
            void    *si_addr;      /* Memory location which caused fault */
            ...
}
```

Rozszerzona informacja o tym skąd i dlaczego dostaliśmy ten sygnał. Interpretuje się ją różnie dla różnych sygnałów, np. dla **SIGILL**:
- ILL_ILLOPN (niepoprawny operand),
- ILL_ILLADR (niepoprawny tryb adresowania),
- …

Przykładowo dla **SIGCHLD**:
- ustawiane są pola standardowe **si_signo**, **si_errno**, **si_code**
- ustawiane są pola charakterystyczne dla tego sygnału: **si_pid**, **si_uid**, **si_status**, **si_utime**, **si_stime**

Żeby korzystać z rozszerzonej funkcji obsługi sygnału związanej z polem sa_sigaction musimy podczas wywoływania funkcji sigaction umieścić w polu sa_flags flagę **SASIGINFO.**

# Zmiana obsługi sygnału: `sigaction` 4/4

**test-sigaction-1.c**

```
void obslugaint(int s) { printf("... nie przerwiesz!\n"); }

int main(void)
{
  int x = 1;
  sigset_t iset;
  struct sigaction act;

  sigemptyset(&iset);
  act.sa_handler = &obslugaint;
  act.sa_mask = iset;
  act.sa_flags = 0;
  sigaction(SIGINT, &act, NULL);

  while (x != 0)
  {
    printf("Skoncze sie dopiero kiedy wprowadzisz 0\n");
    scanf("%d", &x);
  }
  return 0;
}
```

# Zmiana obsługi sygnału: `sigaction` 4/5

**test-sigaction-2.c**

```
...

int a = 1;

void handler(int no, siginfo_t *info, void *ucontext) {
    printf("signo: %d, code:%d\n", info->si_signo, info->si_code);
    a = 0;
}

int main() {
    struct sigaction sa;
    sa.sa_sigaction = handler;
    sigemptyset(&(sa.sa_mask));
    sa.sa_flags = SA_SIGINFO;

    int x = sigaction(SIGINT, &sa, NULL);

    while(a);
    return 0;
}
```

```
$ ./siginfo

^C
signo: 2, code:128
```

# Blokowanie sygnałów: `sigprocmask` 1/2

```
#include <signal.h>

int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

Funkcja zmienia maskę blokowanych sygnałów procesu. Jej zachowanie zależy od ustawionej opcji ***how***:

**SIG_BLOCK** dodanie do aktualnej maski sygnałów z zestawu ***set***

**SIG_UNBLOCK** usunięcie z aktualnej maski sygnałów z zestawu ***set***

**SIG_SETMASK** ustawienie aktualnej maski na zbór sygnałów z zestawu ***set***

Jeżeli ***oldset*** nie jest ustawione na **NULL**, to jest pod nim zapisywany zestaw sygnałów sprzed zmiany.

Jeżeli ***set*** jest ustawione na **NULL**, to maska sygnałów pozostaje niezmieniona, ale ustawienia aktualnej maski są zapisywane w ***oldset*** (o ile różne od **NULL**).

# Blokowanie sygnałów: `sigprocmask` 2/2

**test-sigprocmask.c**

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

int main(void)
{
  int i = 0;
  sigset_t iset;

  sigemptyset(&iset);
  sigaddset(&iset, SIGALRM);
  sigaddset(&iset, SIGINT);
  sigprocmask(SIG_BLOCK, &iset, NULL);
  alarm(5);
  while (1)
    printf("%d\n", i++);
  return 0;
}
```

# Zbiory sygnałów 1/2

Uwaga! Poniższe funkcje służą tylko do grupowania zestawów sygnałów. Ich wywołanie nie powoduje żadnych zmian w obsłudze sygnałów, wysłania sygnałów czy ich blokowania.

```
int sigemptyset(sigset_t *set);
```
Funkcja ustawia pusty zbiór sygnałów ***set***

```
int sigfillset(sigset_t *set);
```
Funkcja ustawia kompletny zbiór sygnałów ***set***

```
int sigaddset(sigset_t *set, int signum);
```
Funkcja dodaje do zbioru sygnałów ***set*** sygnał ***signum***

```
int sigdelset(sigset_t *set, int signum);
```
Funkcja usuwa ze zbioru sygnałów ***set*** sygnał ***signum***

```
int sigismember(const sigset_t *set, int signum);
```
Funkcja sprawdza czy sygnał ***signum*** jest zawarty we wskazanym zbiorze ***set***.

```
int sigpending(const sigset_t *set);
```
Funkcja zwraca do zmiennej ***set*** zbiór sygnałów, które zostały wysłane do procesu i zablokowane.

# Zbiory sygnałów 2/2

**test-block-sig.c**

```
...

sigset_t empty, set1, set2, set3;
sigemptyset(&empt);
sigemptyset(&set1); sigaddset(&set1, SIGUSR1);

sigpending(&set2);
sigprocmask(SIG_BLOCK, &empty, &set3);
printf("%d %d\n", sigismember(&set2, SIGUSR1), sigismember(&set3, SIGUSR1));

sigprocmask(SIG_SETMASK, &set1, NULL);

sigpending(&set2);
sigprocmask(SIG_BLOCK, &empty, &set3);
printf("%d %d\n", sigismember(&set2, SIGUSR1), sigismember(&set3, SIGUSR1));

kill(getpid(), SIGUSR1);
sigpending(&set2);
sigprocmask(SIG_BLOCK, &empty, &set3);
printf("%d %d\n", sigismember(&set2, SIGUSR1), sigismember(&set3, SIGUSR1));

...
```

```
$./test-block-sig
0 0
0 1
1 1
```

# Pamięć

- Sygnały
- **Pamięć**

# Pamięć procesu

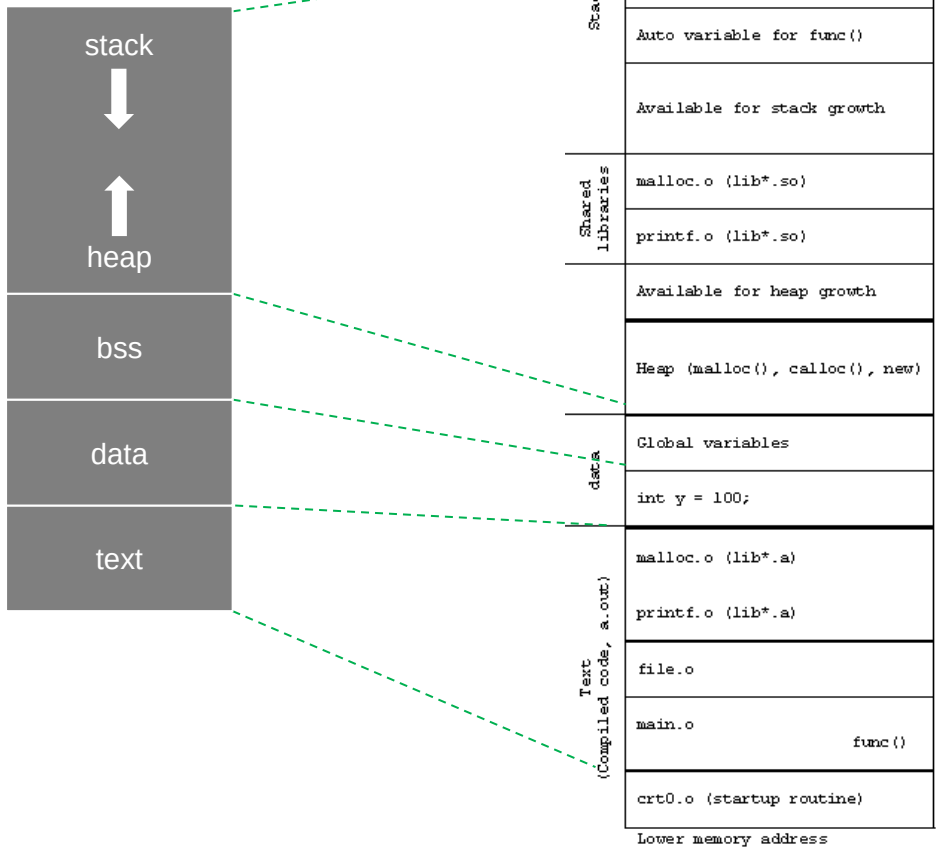
stos (ramki stosu)

sterta (alokacja dynamiczna)

dane globalne niezainicjowane

dane globalne zainicjowane

instrukcje / kod






*COMPILER, ASSEMBLER, LINKER AND LOADER:A BRIEF STORY* (http://www.tenouk.com/ModuleW.html)

# Alokacja na stercie

```
#include <stdlib.h>
void* malloc (size_t size);
```

Funkcja alokuje obszar pamięci o rozmiarze **size** bajtów. W przypadku powodzenia zwracany jest wskaźnik do początku zaalokowanego regionu (w przeciwnym przypadku NULL). Zawartość zaalokowanego obszaru jest nieokreślona (nie musi być wyzerowana).

```
#include <stdlib.h>
void* calloc (size_t nr, size_t size);
```

Funkcja alokuje obszar pamięci, który pomieści **nr** elementów o rozmiarze **size**. W przypadku powodzenia zwracany jest wskaźnik do początku zaalokowanego regionu (w przeciwnym przypadku **NULL**). Zaalokowana pamięć jest wyzerowana.

```
#include <stdlib.h>
void* realloc (void *ptr, size_t size);
```

Funkcja zamienia wielkość regionu pamięci wskazanego przez **ptr**, ustawiając go na **size** bajtów. W przypadku powodzenia zwraca nowy wskaźnik do regionu pamięci. Dotychczasowa zawartość regionu jest zachowana. Podanie wartości 0 jako drugiego argumentu powoduje, że f-cja zachowuje się tak jak **free**.

```
#include <stdlib.h>
void free (void *ptr);
```

Funkcja zwalnia pamięć wskazaną przez **ptr**, zaalokowaną wcześniej np. przez **malloc**. Funkcja nie daje możliwości zwolnienia fragmentu pamięci.

# Anonimowe odwzorowanie w pamięci

- **malloc** w bibliotece ***glibc*** używa sterty przy alokacji niewielkich obszarów pamięci (standardowo do **128kB**)
- Przy alokacji większych obszarów wykorzystywany jest mechanizm **anonimowego odwzorowania w pamięci** (alokacja poza stertą, rozmiar jest całkowitą wielokrotnością strony systemowej, alokacja bardziej czasochłonna niż na stercie)
- Anonimowe odwzorowanie można wymusić za pomocą f-cji **mmap** (z parametrem **MAP_ANONYMOUS** bądź mapując plik **/dev/zero**) i zwolnić za pomocą **unmap**

```
#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);
```

**anonymous-memory.c**

```
…
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… */
…
```

# „Poniżej” funkcji bibliotecznych …malloc

```c
#include <unistd.h>
#include <stdio.h>

int main() {
    void *_brk, *_new;
    int *a = NULL;

    _brk = sbrk(0);
    printf("_brk: %p\n", _brk);

    _brk = sbrk(0);
    printf("_brk: %p\n", _brk);

    _new = sbrk(sizeof(int));
    _brk = sbrk(0);
    printf("_new: %p, _brk: %p\n", _new, _brk);
    if( (void *)-1 != _new) {
        a = (int*)_new;
        *a = 10;
        printf("_new: %p, _brk: %p, a: %p,  *a: %d\n",
                        _new, _brk, a, *a);
    }

    brk(_brk); // sbrk(-sizeof(int))
    return 0;
}
```

```
$./sbrkTest

_brk: 0x5605288d0000
_brk: 0x5605288f1000
_new: 0x5605288f1000, _brk: 0x5605288f1004
_new: 0x5605288f1000, _brk: 0x5605288f1004, a: 0x5605288f1000,  *a: 10
```

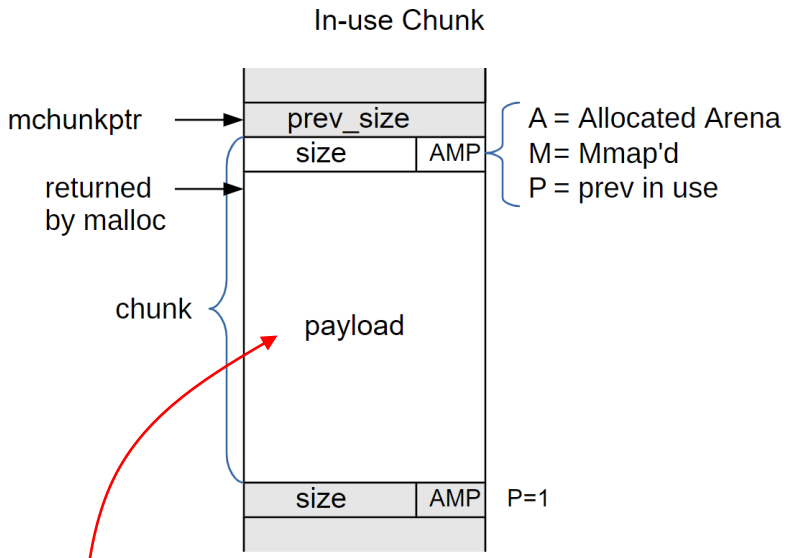
Za pomocą funkcji systemowych **brk** i **srbk** możemy ustalać nowe położenie **program break**, czyli szczytu sterty (najwyższego adresu obszaru alokowanego dynamicznie). Do funkcji **brk** podajemy adres nowego szczytu, do **sbrk** wartość określającą o ile bajtów szczyt ma by przesunięty (można podać również wartość ujemną). Funkcja **sbrk** zwraca adres poprzedniego szczytu stosu – czyli po zalokowaniu pamięci zwróci nam jej początek.

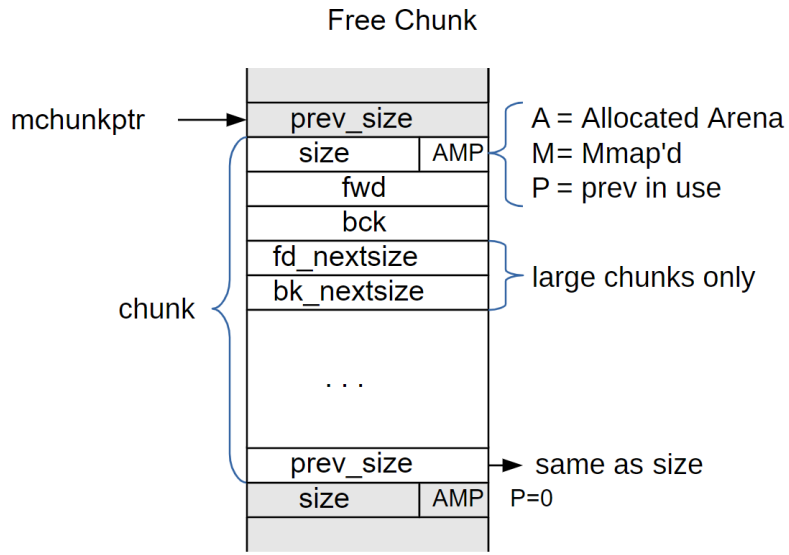
Wywołanie **sbrk(0)** zwróci nam aktualny szczyt sterty bez zmiany jego położenia.

Nie łączymy użycia funkcji **brk** i **sbrk** z funkcjami bibliotecznymi zarządzania pamięci! (zachowanie może być nieprzewidywalne)

Zwróć uwagę, że pierwsze wywołanie f-cji **printf** zmieniło adres szczytu sterty (printf alokował tymczasowo pamięć bez naszej kontroli).

# Zarządzanie pamięcią przez …malloc




```c
#include <unistd.h>
#include <malloc.h>

int main() {
    char *buf1 = malloc(3);
    char *buf2 = malloc(13);

    int size1 = malloc_usable_size(buf1); // payload
    int size2 = malloc_usable_size(buf2);

    printf("%d %d\n", size1, size2);

    return 0;
}
```

```
$ ./meminfo
24 24
```

Funkcja **malloc** w rzeczywistości rezerwuje większy obszar od żądanego, ponieważ musi zachować dodatkowe informacje do zarządzania a sam „payload” musi być wyrównany (funkcja malloc_usable_size zwraca wielkość bloku „payload”).

Inne funkcje: **mallopt**, **mall_trim**, **mallinfo**, …

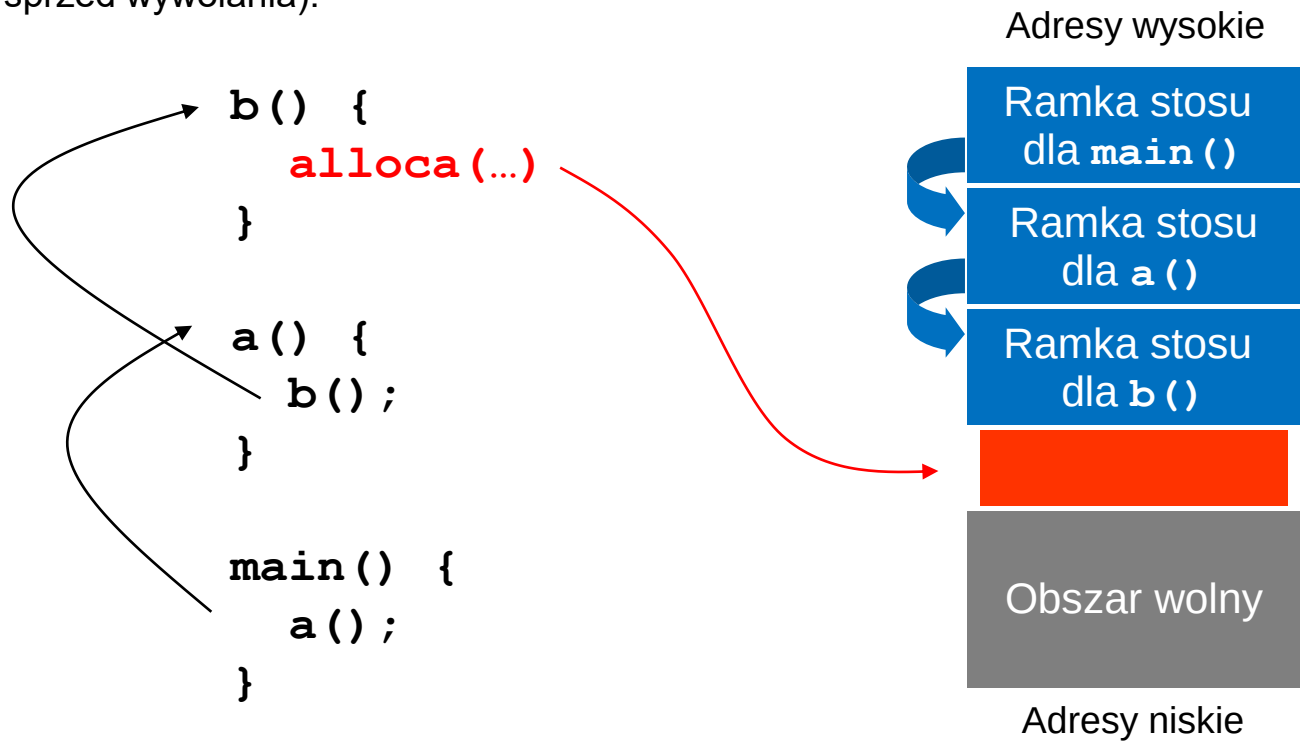


https://sourceware.org/glibc/wiki/MallocInternals

# Alokacja na stosie

```
#include <stdlib.h>
void* alloca (size_t size);
```

Funkcja alokuje obszar pamięci o rozmiarze **size** bajtów, ale nie na stercie a na stosie. **Pamięci zaalokowanej w ten sposób nie zwalniamy manualnie!** Zostaje ona zwolniona automatycznie po zakończeniu wykonywania się funkcji, w której wywołano **alloca** (podczas zdejmowania ze stosu ramki tej funkcji i powrotu do funkcji wywołującej wskaźnik wierzchołka stosu przesuwa się do pozycji sprzed wywołania).




Ramka stosu tworzona jest dla każdego wywołania funkcji i niszczona po jej zakończeniu

# Tablice o zmiennej długości (VLA)

- Zgodnie ze standardem C99 można w języku C używać tablic o zmiennej długości, dla których pamięć alokowana jest automatycznie (na **stosie**).
- Pamięć zaalokowana w ten sposób istnieje do momentu, gdy **zmienna**, która ją reprezentuje, znajdzie się **poza zakresem widoczności** (czyli potencjalnie wcześniej niż przy alokacji za pomocą **alloca**)

**VLA.c**

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

}
…
```

# Alokacja pamięci a funkcje …printf

| | |
|---|---|
| `int printf(const char *format, ...);` | zapis na standardowe wyjście (stdout) |
| `int fprintf(FILE *stream, const char *format, ...);` | zapis do wskazanego strumienia |
| `int sprintf(char *str, const char *format, ...);` | zapis do przygotowanego bufora (wcześniej zaalokowany) |
| `int snprintf(char *str, size_t size, const char *format, ...);` | zapis do przygotowanego bufora z kontrolą ilości danych |
| **`int asprintf(char **strp, const char *fmt, ...);`** | zapis do nieprzygotowanego bufora (automatyczna alokacja, bufor trzeba zwolnić samodzielnie) |

```c
#define _GNU_SOURCE
#include <stdlib.h>
#include <stdio.h>

int main() {
    char *buf;
    asprintf( &buf, "test: (%d)", 100);

    puts( buf);

    free( buf);
    return 0;
}
```

```
$ ./asprintf

test: (100)
```

# Blokady pamięci

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

int main() {
    char *buf1 = malloc(3);

    int retVal1 = mlock(buf1, 3);
    int retVal2 = munlock(buf1, 3);
    printf("%d %d\n", retVal1, retVal2);


    int retVal3 = mlockall(MCL_FUTURE); //MCL_CURRENT, MCL_FUTURE
    int retVal4 = munlockall();
    printf("%d %d\n", retVal3, retVal4);

    return 0;
}
```

- Funkcje: **mlock**, **mlockall**, **munlock**, **munlockall**
- Blokada przeciwdziała wymianie (swap)
- Blokowane są całe strony
- Adres i długość nie muszą być podane do granicy stron (blokowane są strony które znajdują się w podanym zakresie)
- Blokada pamięci nie jest dziedziczona przez dziecko (fork)

```
$ ./mlockTest

0 0
0 0
```

# Alokacja 0 bajtów

- Zachowanie **malloc(0)** jest zależne od implementacji, najczęściej jedno z dwóch:
  - AIX, Solaris i Tru64 UNIX powinny zwrócić **wskaźnik NULL** podczas próby alokacji 0 bajtów
  - Darwin, FreeBSD, IRIX, Linux (z domyślnymi ustawieniami) i Windows zwrócą **wskaźnik różny od zera**, wskazujący na bufor zerowej długości

- … ale jeszcze większe zamieszanie powoduje **realloc(…,0)**

| | returns | ptr | errno |
|---|---|---|---|
| **AIX** | | | |
| `realloc(NULL,0)` | Always `NULL` | | unchanged |
| `realloc(ptr,0)` | Always `NULL` | freed | unchanged |
| **BSD** | | | |
| `realloc(NULL,0)` | only gives `NULL` on alloc failure | | `ENOMEM` |
| `realloc(ptr,0)` | only gives `NULL` on alloc failure | unchanged | `ENOMEM` |
| **glibc** | | | |
| `realloc(NULL,0)` | only gives `NULL` on alloc failure | | `ENOMEM` |
| `realloc(ptr,0)` | always returns `NULL` | freed | unchanged |

```
char *p2;
char *p = malloc(100);
...
if ((p2 = realloc(p, 0)) == NULL) {
  if (p)
    free(p);
  p = NULL;
  return NULL;
}
p = p2;
```

… nie powinno się zwalniać pamięci za pomocą **`realloc`**

# Pamięć jako strumień

```c
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
    FILE *stream;
    char *buf;
    size_t len;

    stream = open_memstream (&buf, &len);
    if (stream == NULL)
        return 1;

    fprintf (stream, "hello my world");
    fflush (stream);
    printf ("buf=%s, len=%zu\n", buf, len);

    fprintf (stream, "!!!");
    printf ("buf=%s, len=%zu\n", buf, len);

    fseeko (stream, 0, SEEK_SET);
    fprintf (stream, "good-bye");
    fclose (stream);
    printf ("buf=%s, len=%zu\n", buf, len);

    free (buf);
    return 0;
}
```

```
$ ./memStream

buf=hello my world, len=14
buf=hello my world!!!, len=14
buf=good-bye, len=8
```

Strumień pamięci otwieramy za pomocą **`open_memstream`**.
Dalej używamy standardowe funkcje biblioteczne do obsługi strumieni (**`fprintf`**, **`fgets`**, **`fflush`**, **`fseeko`**, **`fclose`** itd.).
Bufor pamięci związany ze strumieniem jest automatycznie alokowany i realokowany (w razie potrzeby zmiana wielkości).

Bufor nie jest automatycznie zwalniany po zamknięciu strumienia!