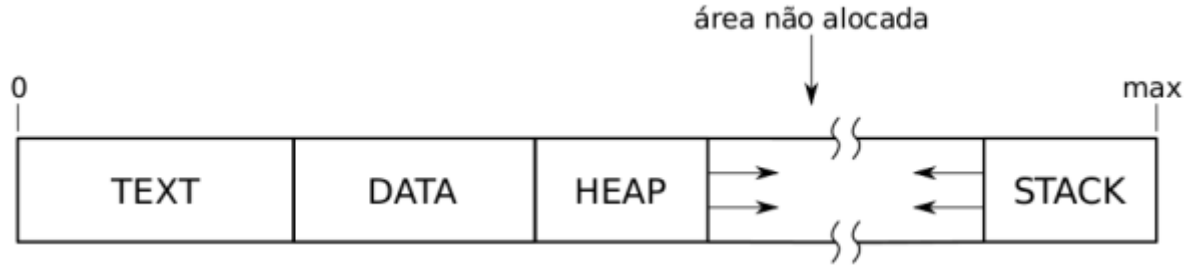
Gerência de memória

O espaço de endereços de um processo é dividido em várias áreas distintas. As mais importantes são:

- *Text area*: contém o código do programa e suas constantes. Esta área é alocada durante a chamada exec e permanece do mesmo tamanho durante toda a vida do processo.
- *Data area*: é a memória de trabalho do processo, onde ele armazena suas variáveis globais e estáticas. Tem tamanho fixo ao longo da execução do processo.
- *Stack area*: contém a pilha de execução, onde são armazenadas os parâmetros, endereços de retorno e variáveis locais de funções. Pode variar de tamanho durante a execução do processo.
- *Heap area*: contém áreas de memória alocadas a pedido do processo, durante sua execução. Varia de tamanho durante a vida do processo.



Um programa em C suporta três tipos de alocação de memória:

- A **alocação estática** ocorre quando são declaradas variáveis globais ou estáticas; geralmente usa a área *Data*.
- A **alocação automática** ocorre quando são declaradas variáveis locais e parâmetros de funções. O espaço para a alocação dessas variáveis é reservado quando a função é invocada, e liberado quando a função termina. Geralmente é usada a pilha (*stack*).
- A **alocação dinâmica**, quando o processo requisita explicitamente um bloco de memória para armazenar dados; o controle das áreas alocadas dinamicamente é manual ou semi-automático: o programador é responsável por liberar as áreas alocadas dinamicamente. A alocação dinâmica geralmente usa a área de *heap*.

Alocação dinâmica de memória

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

Esta função aloca uma nova região com `size` bytes de tamanho e retorna um pointeiro para o início da mesma (ou 0 em caso de erro). O conteúdo da nova área é indefinido. Eis um exemplo de uso:

```
struct mystruct *ptr;
...
ptr = (struct mystruct *) malloc (sizeof (struct mystruct));
if (ptr == 0) abort ();
```

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

Esta função libera um bloco de memória previamente alocado, apontado por `ptr`.

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

Esta função redimensiona o bloco previamente alocado apontado por `ptr` para o novo tamanho `newsize`. Retorna o novo endereço do bloco, que pode ser diferente do anterior, caso tenha sido necessário mudá-lo de lugar (o conteúdo original do bloco é preservado nesse caso ou em caso de erro).

```
#include <stdlib.h>
void * calloc (size_t count, size_t eltsize)
```

Esta função aloca um bloco de tamanho suficiente para conter um vetor com `count` elementos de tamanho `eltsize` cada um. O conteúdo do bloco alocado é preenchido por zeros.

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

Esta função provê um mecanismo de alocação dinâmica semi-automática: o bloco alocado será liberado automaticamente ao encerrar a função onde o bloco foi alocado. O valor de retorno da chamada é o endereço de um bloco de tamanho `size` bytes, alocado na pilha da função atual (como se fosse uma variável local).

Na GLibC, os blocos alocados pelas funções acima sempre iniciam em um endereço múltiplo de 8 em plataformas de 32 bits. Caso seja necessário obter blocos iniciando em múltiplos de 16, 32, 64, etc, as funções a seguir estão disponíveis:

```
#include <malloc.h>
void * memalign (size_t boundary, size_t size)
```

Aloca um bloco de tamanho `size` cujo endereço inicial é um múltiplo de `boundary` (que deve ser 2^n). Retorna o endereço do bloco alocado, que pode ser liberado mais tarde através da função `free`. Ver também a função `posix_memalign`.

Arquivos mapeados em memória

Sistemas operacionais modernos permitem mapear um arquivo em uma região de memória. Isso torna possível acessar o arquivo como se fosse um vetor de bytes em memória. Esse procedimento é mais eficiente que os tradicionais `read`/`write`, pois somente as regiões acessadas do arquivo são carregadas em memória. Como o mecanismo de mapeamento de arquivo faz uso dos mecanismos de memória virtual, é possivel mapear arquivos muito grandes na memória (o limite é o espaço de endereçamento).

```
#include <sys/mman.h>
void * mmap (void *address, size_t length,int protect, int flags, int filedes,
off_t offset)
```

Cria um novo mapeamento em memória relacionado aos bytes `offset` ... `offset+length-1` no arquivo aberto indicado pelo descritor `filedes`. O mapeamento não é removido ao fechar o descritor, ele deve ser desfeito explicitamente.

O campo `address` indica um endereço preferencial para mapear o arquivo na memória (ou NULL). O campo `protect` contém flags indicando o tipo de acesso permitido àquela região: leitura (PROT_READ), escrita (PROT_WRITE) ou execução (PROT_EXEC). Acessos inválidos resultarão em um sinal SIGSEGV. Essas proteções podem ou não ser suportadas pelo hardware.

O campo `flags` contém flags que controlam o tipo de mapeamento:

- MAP_PRIVATE: escritas na área mapeada em memória não devem ser escritas de volta no disco, ou seja, o mapeamento é privado do processo que o fez.
- MAP_SHARED: escritas na área mapeada em memória serão imediatamente visíveis por outros processos mapeando esse mesmo arquivo em memória (caso seja necessário atualizar o conteúdo físico em disco rapidamente, deve-se utilizar a chamada msync).
- MAP_FIXED: força o sistema a usar o endereço especificado em address (ou falhar se não for possível).
- MAP_ANONYMOUS: cria um mapeamento anônimo, ou seja, não conectado a nenhum arquivo. Pode ser útil para comunicação entre processos sem a necessidade de se criar um arquivo em disco.

Deve-se observar que nem todos os descritores de arquivo podem ser mapeados em memória. Normalmente, somente arquivos normais e dispositivos orientados a blocos podem ser mapeados em memória. Além disso, essa funcionalidade não está disponível em sistemas mais antigos.

```
#include <sys/mman.h>
int munmap (void *addr, size_t length)
```

Remove mapeamentos efetuados entre (addr) e (addr + length), onde length deve ser o tamanho do mapeamento. Mais de um mapeamento pode ser removido em uma só operação.

```
#include <unistd.h>
#include <sys/mman.h>
int msync (void *address, size_t length, int flags)
```

Permite sincronizar um mapeamento de arquivo em memória com a imagem do arquivo em disco. A região é indicada pelos campos address e length. O campo flags pode conter as seguintes opções:

- MS_SYNC : força os dados a serem efetivamente escritos no disco (a operação default apenas garante que outros processo com o arquivo aberto de forma convencional verão as mudanças recentes).
- MS_ASYNC : indica a msync para iniciar a sincronização, mas não aguardar sua conclusão.

Proteção de páginas em memória

É possível informar ao sistema operacional que certas páginas de memória virtual nunca devem ser enviadas para o disco (*swapping* ou *paging*). Isso pode ser necessário em algumas circunstâncias:

- *Velocidade*: buscar uma página faltante em disco toma muito mais tempo que acessar a mesma página em memória. Esse tempo adicional de paginação pode prejudicar a execução de processos críticos.
- *Segurança*: informação temporária sensível escrita na memória (como senhas) pode ir para o disco através de *swapping* e ficar lá por muito tempo, podendo ser capturada mais facilmente.

Por causa de seus possíveis efeitos sobre outros processos (reduzindo a disponibilidade de memória do sistema), normalmente só processos do administrador podem travar páginas em memória (mais detalhes na página de manual de mlock).

```
#include <sys/mman.h>
int mlock (const void *addr, size_t len)
```

Permite travar na memória um conjunto de páginas do processo. A faixa de memória a travar inicia em addr e tem len bytes de tamanho. Todas as páginas atingidas por essa faixa são travadas.

```
#include <sys/mman.h>
int munlock (const void *addr, size_t len)
```

Faz o inverso de mlock.

```
#include <sys/mman.h>
int mlockall (int flags)
```

`mlockall` trava todas as páginas usadas pelo processo (ou que virão a ser usadas no futuro). `flags` indica o tipo de travamento a realizar: `MCL_CURRENT` (trava as páginas atuais) e/ou `MCL_FUTURE` (trava as páginas que vierem a fazer parte do espaço de endereços do processo no futuro).

```
#include <sys/mman.h>
int munlockall (void)
```

Faz o contrário de `mlockall`, destravando todas as páginas presentes e futuras do processo.

```
#include <sys/mman.h>
int mprotect(const void *addr, size_t len, int prot);
```

Controla a forma como uma região de memória pode ser acessada. Caso a forma de acesso definida seja violada, o processo recebe um sinal `SIGSEGV`. A região de memória a proteger inicia em `addr` e possui `len` bytes de tamanho (`addr` deve ser um múltiplo de `PAGESIZE`, pois a unidade básica de proteção é a página); o campo `prot` indica o tipo de proteção a aplicar, que pode ser uma combinação (OU binário) das seguintes macros:

- `PROT_NONE` : a memória não pode ser acessada.
- `PROT_READ` : a memória pode ser lida.
- `PROT_WRITE` : a memória pode ser escrita.
- `PROT_EXEC` : a memória pode conter código executável.

Operações em blocos de memória

As funções aqui indicadas são úteis para operar com blocos de memória alocados estatica ou dinamicamente.

```
#include <string.h>
void * memcpy (void *restrict to, const void *restrict from, size_t size)
```

Copia `size` bytes do bloco iniciando no endereço from para o bloco iniciando no endereço to. O comportamento dessa função é imprevisível caso haja sobreposição das áreas de memória indicadas.

```
#include <string.h>
void * memmove (void *to, const void *from, size_t size)
```

Copia `size` bytes do bloco iniciando no endereço `from` para o bloco iniciando no endereço `to`, mesmo que haja sobreposição entre os blocos.

```
#include <string.h>
void * memccpy (void *restrict to, const void *restrict from, int test, size_t
size)
```

Copia até `size` bytes do bloco iniciando em `from` para o bloco iniciando em `to`, parando ao copiar todos os bytes ou ao encontrar um byte cujo valor seja igual a `test`.

```
#include <string.h>
void * memset (void *addr, int value, size_t size)
```

Copia o valor de `value` (convertido para `unsigned char`) nos primeiros `size` bytes do bloco de memória

iniciando em `addr`.

Depurando problemas de memória

As operações envolvendo ponteiros e alocação dinâmica de memória costumam levar a bugs complexos e muitas vezes difíceis de resolver. Existem várias ferramentas para auxiliar o programador a localizar e resolver problemas relacionados à alocação dinâmica de memória e uso inadequado de ponteiros em C/C++:

- Dmalloc
- GNU Checker
- Memory Supervision System
- LeakTracer
- Valgrind
- Parasoft Insure++
- Rational Purify

Atividades

A definir...