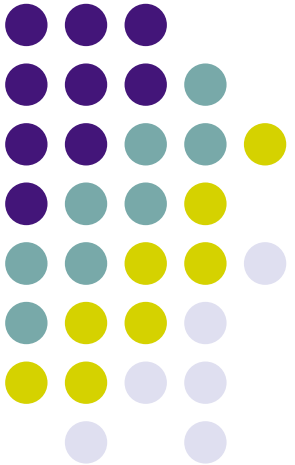
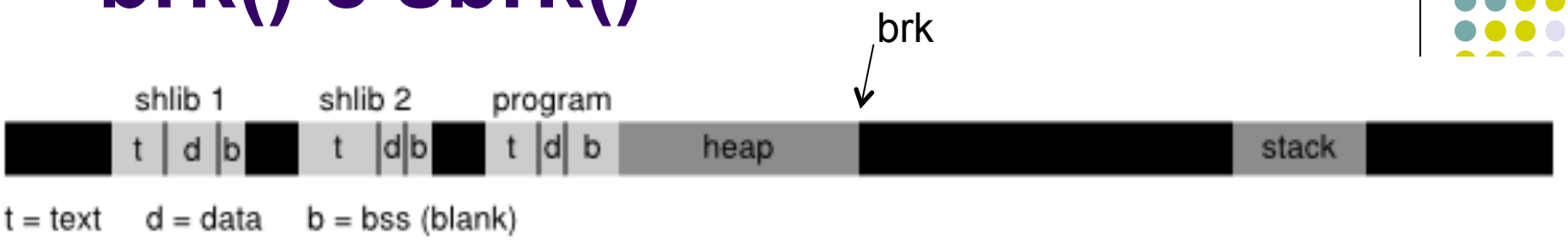
Sistemas de Computação

Manipulando a Memória



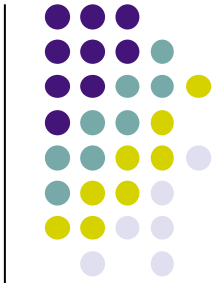
brk() e sbrk()




```
#include <unistd.h>

        void *sbrk(intptr_t increment n)
        int* brk (void *addr)
```

- Estas chamadas permitem obter o limite da memória alocada para o heap (sbrk) e redefinir este limite (brk)
- `malloc` internamente chama `brk()`

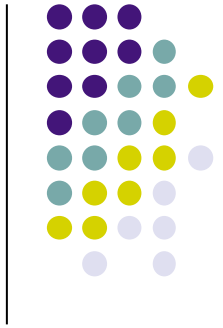
Obtendo o limite do heap



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

int main(void) {
    void *b = sbrk(0);  /* obtém o primeiro endereço além do
limite do heap */
    int *p = (int *)b;
    /* Segmentation Fault because it is outside of the heap. */
    *p = 1;
    *(p+100) = 2;
    return 0;
}
```

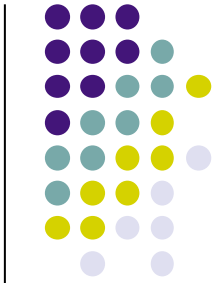
- Grandes chances de dar um erro “Segmentation Fault”

Movendo o limite do heap



```
#define _GNU_SOURCE
#include <assert.h>
#include <unistd.h>

int main(void) {
    void *b = sbrk(0);
    int *p = (int *)b;

    brk(p + 100); /* Move it 100 ints forward */
    *p = 1;
    *(p + 100) = 2;
    assert(*p == 1);
    assert(*(p + 1) == 2);

     brk(b); /* Desaloca voltando ao endereço anterior b
*/
     return 0;
}
```

Alocando e Desalocando porções do heap

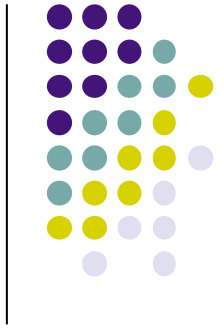


myallocate.c

```
void *memorypool;
void *myallocate( int n){
    return sbrk(n);
}
void initmemorypool(void) {
      memorypool = sbrk(0);
}
void resetmemorypool(void){
    brk(memorypool);
}
```

- Não é a mesma coisa do que malloc(), pois não é possível liberar (desalocar) alocações individuais
- Precisa liberar do mais recente para o mais antigo (como uma pilha)
- Atenção: essa sua alocação pode conflitar com chamadas que fazem uso do malloc()!!

Descobrindo o tamanho da página



```
#include <unistd.h>
long sysconf (int name)
```

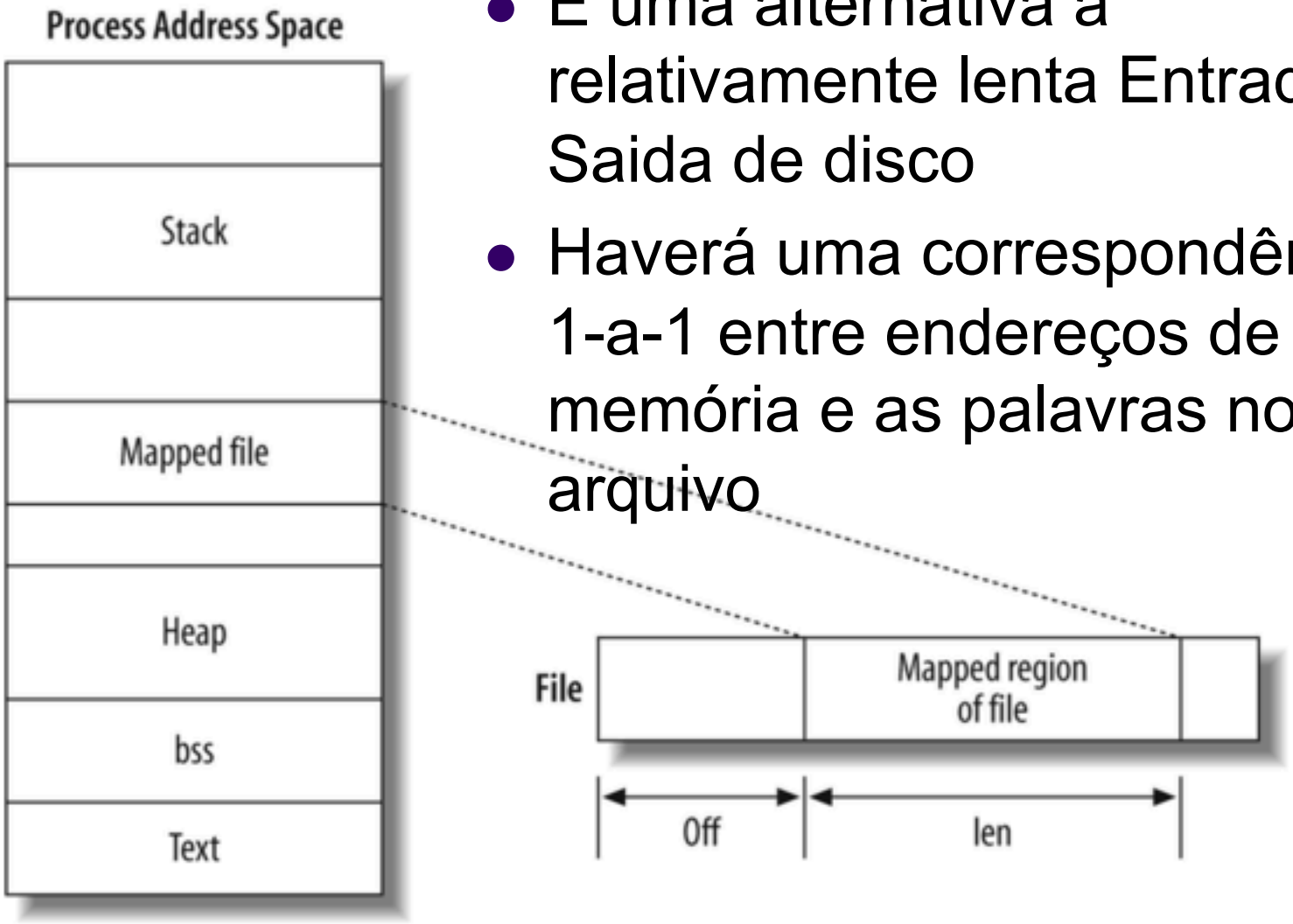
- POSIX define constante: `_SC_PAGESIZE`
- Uso:

```
long pagesize = sysconf(_SC_PAGESIZE);
```

- Em Linux:

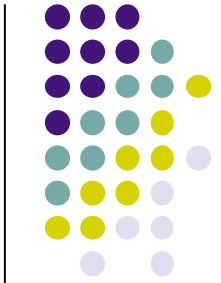
```
int getpagesize(void)
```

Arquivos Mapeados na Memória



- É uma alternativa à relativamente lenta Entrada/ Saida de disco
- Haverá uma correspondência 1-a-1 entre endereços de memória e as palavras no arquivo

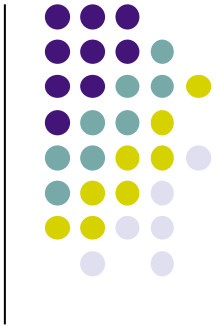
mmap()



```
#include <sys/mman.h>
void *mmap (void *addr,
            size_t len,
            int prot,
            int flags,
            int fd,
            off_t offset);
```

- Retorna o endereço na memória para onde o arquivo foi mapeado
- `addr` : sugestão ao kernel do endereço para onde mapear o arquivo
- `len`: quantidade de bytes mapeados
- `prot`: PROT_READ/ PROT_WRITE/ PROT_EXEC
- `fd:` descritor do arquivo previamente aberto
- `offset`: a partir de qual posição no arquivo

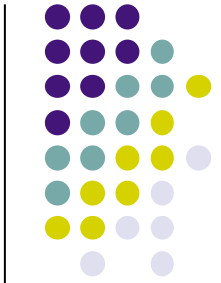
mmap() – cont.



```
#include <sys/mman.h>
void *mmap (void *addr,
            size_t len,
            int prot,
            int flags,
            int fd,
            off_t offset);
```

- flag: como o mapeamento deve ser feito. Os principais são:
  - MAP_FIXED – o **`addr`** era mandarório (não apenas uma sugestão)
  - MAP_PRIVATE – mapeamento não é compartilhado com outros processos, e qualquer escrita no arquivo mapeado não são refletidas no arquivo original no disco e nem nas cópias mapeadas para outros processos
  - MAP_SHARED – compartilha o mapeamento com todos os outros processos que também tenham esse mapeamento. Escritas são refletidas instanteamente em todos os demais processos

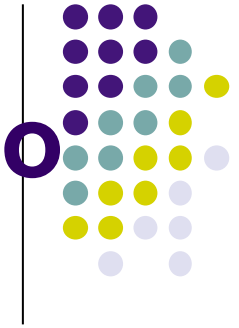
Exemplo de chamada



```
void *p;
p = mmap(0,len, PROT_READ, MAP_SHARED,fd,0);
if (p == MAP_FAILED) perror ("mmap");
```

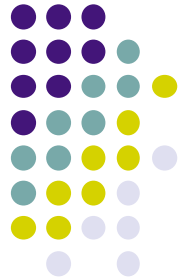
- Mapeamento do arquivo com descritor `fd`, a partir do seu começo (`offset ==0`) e de `len` bytes, para somente leitura
- A página de memória é a menor unidade que pode ter direitos de acesso específicos. Por isso, cada arquivo é mapeado para um conjunto inteiro de páginas, independente do seu tamanho.
- Os espaços restantes no final da última página são preenchidos com zeros (0)

Possíveis erros no mapeamento



- Em caso de falha (`MAP_FAILED`), `mmap()` seta o valor de `errno`, que podem ser:
  - EACCESS – o arquivo não era um arquivo regular
  - EAGAIN – o arquivo estava seguro com um file lock
  - EINVAL – um ou mais dos argumentos de mmap estão inválidos
  - ENODEV – o sistema de arquivos no qual o arquivo reside não supota mmap()
  - EOVERFLOW – add+len excederam o tamanho do espaço de endereçamento

Desfazendo o mapeamento

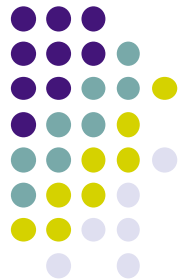


```
int munmap (void *addr, size_t len);
```

- Remove qualquer mapeamento páginas a partir do endereço `addr` e nos `len` bytes consecutivos
- Normalmente, chama-se `munmap()` com o endereço retornado por `mmap()` anterior e o seu len correspondente.

map-exemplo.c

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



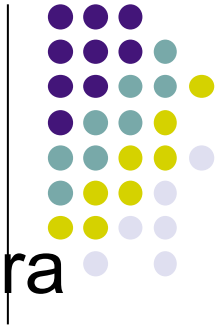
```
# include varios…
int main (int argc, char *argv[])
{
   struct stat sb;
   off_t len;
   char *p;   int fd;

   if (argc < 2) { fprintf (stderr, "usage: %s <file>\n",
                      argv[0]); exit(1); }
   fd = open (argv[1], O_RDONLY);
   if (fd == -1) { perror ("open"); exit(1); }
   if (fstat(fd, &sb)== -1) { perror ("fstat"); exit(1); }
   if (!S_ISREG (sb.st_mode)) { fprintf (stderr, "%s is not
a file\n", argv[1]); exit(1);}

   p = mmap (0, sb.st_size, PROT_READ, MAP_SHARED, fd, 0);
   if (p == MAP_FAILED)  { perror ("mmap"); exit(1);  }

   if (close (fd) == -1) { perror ("close"); exit(1); }

   for (len = 0; len < sb.st_size; len++) putchar (p[len]);

   if (munmap  (p, sb.st_size) == -1) {perror ("munmap");
exit(1); }
   return 0;
}
```

Algumas observações

- Mapeamentos para memória são sempre um número interiro de páginas. Ou seja, para arquivos pequenos (poucos bytes), e páginas de 4KB voce estará desperdiçando muita memória!
- Os mapeamentos precisam caber no espaço de endereçamento do processo. Se forem mapeados muitos arquivos, isso pode levar à fragmentacão de memória, o que pode tornar a execução do seu programa mais lento.
- Existe um overhead do núcleo para manter os mapeamentos de arquivo em memória. Portanto, só compensa para arquivos muito frequentemente acessados.

Perguntas?

Exercícios!



1. Faça um programa que usa `myallocate` para alocar e desalocar memória dinâmiamente.

2. Execute map-exemplo.c passando o nome de um arquivo como parámetro. Qual foi a novidade?

3. Modifique **`map-exemplo.c`** para:
   a. Verificar quantas páginas de memória o seu arquivo precisou
   b. Medir o quão mais rápido foi a escrita do arquivo mapeado quando comparado com a E/S em disco. Dica: use gettimeofday()

Exercícios!

4. Escreva dois programas prog1.c e prog2.c que trocam dados através de um mesmo arquivo mapeado para memória (de cada um).

5. Escreva um programa que faz um mapeamento de arquivo para memória e depois escreve novos caracteres no final do arquivo. O que voce precisa fazer para evitar um SegFault?

- O Linux também suporta uma função:

```
void *mremap(void *addr, size_t old_sz, size_t
new_sz, unsigned long flags) // addr precisa
estar alinhado com o início da página
```

6. Re-escreva o seu programa do item anterior usando mremap()