

## GERENCIAMENTO DE MEMÓRIA

Explicação sobre as três syscalls utilizadas diretamente em memory.c.

**mmap**: Esta syscall mapeia uma região de memória para o processo que a chama.

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

Parâmetros:

- void\* addr, é o endereço de memória aonde o mapeamento ocorrerá, caso seja nulo, o kernel escolherá um endereço, e caso o endereço passado como parâmetro esteja indisponível, o kernel também escolherá um endereço.
- size\_t length, é o tamanho da região de memória a ser alocada.
- int prot, define a proteção de memória, ou seja, define se a memória pode ser lida, escrita ou executada.
- Int flags, esse parâmetro define o tipo de mapeamento que ocorrerá, no caso do exemplo utilizamos “MAP\_ANONYMOUS”, para que o mapeamento ocorra sem um arquivo.
- Os parâmetros fd e offset não foram utilizados no exemplo, por conta da flag utilizada, visto que não é necessário passar um descritor de arquivo nem o offset.

A syscall retorna o endereço para a região alocada.

**mumap**: Esta syscall deleta o mapeamento de memória de alguma região.

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

Parâmetros:

- void \*addr, é o endereço da região mapeada em memória.
- size\_t length, é o tamanho da região de memória.

A syscall retorna 0 caso sua execução tenha sido um sucesso, ou -1 caso tenha ocorrido um erro.

**mprotect**: Esta syscall define as permissões de acesso a uma região de memória.

**`int mprotect(void *addr, size_t len, int prot);`**

Parâmetros:

- void \*addr, é o endereço da região de memória.
- size\_t len, é o tamanho da região de memória.

- int prot, define o tipo de proteção, os utilizados no exemplo são:
  - PROT\_NONE, a memória não pode ser acessada.
  - PROT\_WRITE, a memória pode ser modificada.
  - PROT\_READ, a memória pode ser lida.

A syscall retorna 0 caso seja executada com sucesso, e retorna -1 caso tenha encontrado algum erro.

## **E/S E ARQUIVOS**

**Access**: system call que checa se um processo pode acessar um determinado diretório.

Faz parte da biblioteca <unistd.h> em C e deve ser usado da seguinte forma:

```
access(const char *pathname, int mode);
```

onde pathname é o diretório a ser consultado e mode o tipo de acesso, podendo ser F\_OK (checa se o diretório existe), R\_OK (verifica a permissão para leitura do diretório), W\_OK (verifica a permissão para escrita no diretório) ou X\_OK (verifica e executa permissões no diretório).

Essa verificação é feita utilizando o UID real (ID real do usuário) e o GID (ID do grupo) do processo que chama essa system call, ao invés do UID efetivo, como outras system calls utilizam.

Caso tudo ocorra com sucesso, access() retorna zero; caso contrário, retorna -1, e errno é setado. Para saber o erro exato que pode ter ocorrido ao chamar a função, utiliza-se a biblioteca errno.h. Essa biblioteca nos indicará alguns erros, dentre eles:

EACCESS: Processo não possui permissão para acessar o diretório desejado!

ENOENT: pathname não existe.

EROFS: Tentativa de escrita em um diretório para apenas leitura (“read-only”).

EIO: Um erro de I/O ocorreu.

**Fstat**: system call que te fornece todas as informações relevantes de um arquivo ou diretório.

Para explicar melhor o fstat, citarei primeiro o comando stat. Esse comando é usado passando como parâmetro o caminho (path) desejado e uma struct do tipo stat que é usada para retornar as informações do path informado.

A struct é definida da seguinte forma:

```
struct stat {  
    dev_t    st_dev;        /* ID do arquivo ou diretório */  
    ino_t    st_ino;        /* Inode */  
    mode_t   st_mode;       /* Tipo de arquivo */  
    nlink_t  st_nlink;      /* Número de hard links */  
    uid_t    st_uid;        /* User ID */  
    gid_t    st_gid;        /* Group ID */  
    dev_t    st_rdev;       /* Device ID (se special file) */  
    off_t    st_size;       /* Tamanho total, em bytes */  
    blksize_t st_blksize;   /* Tamanho do bloco para o sistema de arquivos I/O */  
    blkcnt_t st_blocks;     /* Número de 512B blocos alocados */  
  
    /* Desde o Linux 2.6 há suporte para precisão em nanossegundos */  
  
    struct timespec st_atim; /* Último acesso */  
    struct timespec st_mtim; /* Última modificação */  
    struct timespec st_ctim; /* Última alteração de status */  
  
    #define st_atime st_atim.tv_sec    /* compatibilidade para versões de Linux  
anteriores */  
  
    #define st_mtime st_mtim.tv_sec
```

```
#define st_ctime st_ctim.tv_sec
```

```
};
```

Se tudo ocorrer com sucesso, a função retorna zero; caso contrário, retorna -1 e `errno` é setado. Utilizando a biblioteca `<errno.h>` conseguimos saber qual erro ocorreu.

Podemos executar o `stat` também apenas passando, no prompt de comandos, o path desejado, por exemplo: “`stat Desktop/`”.

O `fstat` funciona da mesma forma que o `stat`, com uma única diferença: ao invés de passar o path como parâmetro, a função `fstat` recebe o descritor do arquivo (número inteiro).

```
fstat(int fd, struct stat *buf)
```

Onde `fd` é o descritor do arquivo e `buf` é a struct do tipo `stat` passada por referência.

**Open**: abre ou cria um arquivo.

Pertence às bibliotecas `<sys/types.h>`, `<sys/stat.h>` e `<fcntl.h>`. Sua estrutura pode ser:

```
open(const char* pathname, int flags) ou open(const char* pathname, int flags,  
                                             mode_t mode)
```

Ela abre um arquivo especificado em `pathname`, podendo criar um novo arquivo caso este não exista (modo de escrita) ou retornar um erro (modo de leitura). Caso não haja erro, `open()` retorna um número inteiro não-negativo referente ao descritor do arquivo.

O parâmetro `pathname` é o caminho onde está localizado o arquivo que será aberto (ou criado), o parâmetro `flags` deverá ser um dos três a seguir: `O_RDONLY`, `O_WRONLY`, `O_RDWR` (apenas leitura, apenas escrita ou leitura e escrita, respectivamente).

A partir da versão 2.26 da glibc, é utilizada a system call `openat` ao invés de `open`. A `openat` difere da `open` no tratamento do `pathname`, com a adição, na estrutura da função, do descritor de arquivo do diretório (`dirfd`).

`openat(int dirfd, const char* pathname, int flags)`

Se `pathname` for um caminho relativo, ele será interpretado como relativo ao `dirfd` informado, e não relativo à mesma pasta que se encontra o programa (como ocorre com `open`).

Se `dirfd` for igual a `AT_FDCWD`, então `pathname` será tratado como relativo à mesma pasta que se encontra o programa. Já se `pathname` se tratar de um caminho absoluto, `dirfd` será ignorado.

## ***PROCESSOS***

**Execve**: executa um programa específico.

Faz parte da biblioteca `<unistd.h>` e tem sua estrutura definida da seguinte forma:

```
int execve(const char *pathname, const char *argv[], const char *envp[])
```

Esta `systemcall` executará o programa localizado em `pathname`, onde este `pathname` deverá ser um executável binário ou um script que contém em sua primeira linha o seguinte código:

```
#!/interpreter [optional-arg]
```

O parâmetro `argv` deverá ser um array de argumentos, onde, por convenção, o primeiro índice (`argv[0]`) deverá ser o nome do executável. E `envp` é um array de strings da forma `chave=valor` que será o ambiente de execução do programa.

**Geteuid**: retorna o ID efetivo do usuário que chamou o processo.

Esta `system call` faz parte das bibliotecas `<unistd.h>` e `<sys/types.h>`. Possui a seguinte estrutura:

```
uid_t geteuid(void);
```

No Linux, cada usuário do SO possui um número inteiro relacionado a ele, chamado de `userID`, que se divide em três tipos: `userID` real, `userID` efetivo e `userID` salvo. O primeiro é o próprio número inteiro referente a um usuário. O segundo é utilizado quando um usuário sem privilégios deseja acessar um recurso que necessita de privilégio (`root`). O último, por sua vez, é usado quando geralmente um processo está rodando com privilégios `root` e precisa realizar alguma tarefa que não necessita de tal privilégio. Ele então é alternado para um usuário não-privilegiado, salvando

seu userID efetivo para retornar a seu acesso privilegiado quando o processo terminar a execução.

**Getuid**: De modo similar ao `geteuid`, a system call `getuid()` retorna o userID real do usuário que o está chamando. Pertence às bibliotecas `<unistd.h>` e `<sys/types.h>`, e sua estrutura também é similar ao `geteuid`:

```
uid_t getuid(void);
```

## **Outras Syscalls Utilizadas**

**brk**: Essa syscall define o tamanho do segmento de memória disponível para o programa armazenar os dados.

**arch\_prctl**: Define ou modifica a arquitetura utilizada em uma thread.

**exit\_group**: Finaliza todas as threads do processo.

**close**: Encerra um descritor de arquivo, removendo toda referência do descritor a um arquivo, tornando esse descritor um número reutilizável.

**read**: Lê determinado número de bytes de um arquivo a partir de um descritor de arquivo fornecido por parâmetro.

**write**: Escreve determinado número de bytes em um arquivo a partir de um descritor de arquivo fornecido por parâmetro.

## REFERÊNCIAS:

- [https://www.tutorialspoint.com/unix\\_system\\_calls/mprotect.htm](https://www.tutorialspoint.com/unix_system_calls/mprotect.htm)
- <http://man7.org/linux/man-pages/man2/mprotect.2.html>
- [https://www.tutorialspoint.com/unix\\_system\\_calls/mmap.htm](https://www.tutorialspoint.com/unix_system_calls/mmap.htm)
- <http://man7.org/linux/man-pages/man2/mmap.2.html>
- <http://man7.org/linux/man-pages/man2/access.2.html>
- <http://man7.org/linux/man-pages/man2/stat.2.html>
- <https://linux.die.net/man/2/fstat>
- <https://linux.die.net/man/2/access>
- <https://linux.die.net/man/2/stat>
- <https://linux.die.net/man/2/open>
- <http://man7.org/linux/man-pages/man2/open.2.html>
- <http://man7.org/linux/man-pages/man2/execve.2.html>
- <https://linux.die.net/man/2/getuid>
- <https://linux.die.net/man/2/geteuid>
- <http://man7.org/linux>