Sistemas Operativos

Resumo

Rafael Rodrigues

LEIC
Instituto Superior Técnico
2023/2024

Contents

1 Organização dos Sistema Operativos

1.1 Conceitos

TODO

1.2 System Calls

TODO

1.3 Estruturas do Sistema Operativo

TODO

- Sistemas Monolíticos
- Sistemas em Camadas
- Micro-Núcleos

1.4 Boot

Sequência de inicialização de um computador:

1. A máquina recebe energia, o PC (Program Counter) aponta para um programa (firmware) na Boot ROM, normalmente uma BIOS ou UEFI.
2. Este programa faz algumas verificações sobre o computador (se está em condições de ser iniciado) e, de seguida, determina a localização do bootloader.
3. Copia o bootloader para RAM, e passa-lhe o controlo (salta).
4. O bootloader, por sua vez, carrega o programa do núcleo em RAM e salta para a rotina de inicialização do núcleo:
    - inicializar as suas estruturas de dados
    - copiar rotinas de tratamento de cada interrupção para RAM
    - preencher a tabela de interrupções em RAM
    - lançar os processos inicias do sistema, incluindo o processo de login

1.5 Gestor de Processos

TODO

1.6 Despacho

TODO

- Se o núcleo não utilizasse uma pilha (stack) diferente da usada pelas aplicações, aplicações maliciosas poderiam manipular o estado interno do kernel.
- A última instrução executada pelo despacho é Return from Interrupt (RTI).
- A comutação entre processos implica custos maiores do que a comutação entre tarefas do mesmo processo.
- Um processo no estado executável executa sempre em modo núcleo antes de executar em modo utilizador.
- Durante a execução da chamada de sistema execl o núcleo copia os argumentos de input da execl da pilha utilizador para a pilha núcleo
- A escolha da min_granularity pode afetar a latência de comutação.
- O processo quando é acordado por um evento necessita de verificar se a condição que levou a acordar ainda é válida senão tem de bloquear-se. Se se bloqueou com `wait_interruptible` os `signals` desbloqueiam o processo e não verifica se a condição de bloqueio se tornou válida

1.7 Escalonamento (Scheduling)

Scheduler - escolhe porque ordem devem correr os processos executáveis, e por quanto tempo.

Métricas

Para o nosso escalonamento ser o melhor possível, é necessário definir métricas:

- **Throughput**: número de trabalhos por hora
- **Turn around time**: tempo entre a submissão do trabalho e a obtenção do resultado
- **Utilização de CPU**: percentagem de tempo de uso do processador
- **Responsividade**: responder o mais rapidamente possível aos eventos desencadeados por utilizadores
- **Previsibilidade**: garantir que os conteúdos são carregados pelo menos a uma dada velocidade. Importante, por exemplo, para conteúdos de multimédia.

Um conceito também importante é o da prioridade de um processo, que dita a sua importância. Um processo prioritário corre com maior probabilidade que um outro. A prioridade pode ser fixa ou dinâmica.

Podemos ainda categorizar os processos em duas classes, do ponto de vista do scheduler:

- **I/O bound**: fazem uso intensivo de dispositivos de entrada e saída - são interativos.

  Não costumam utilizar todo o quantum que lhes é disponibilizado.
- **CPU bound**: fazem uso intensivo do processador.

  São normalmente penalizados pelos algoritmos de escalonamento que utilizam prioridades dinâmicas

Políticas

- **Round-Robin**

  Nesta política, define-se um quantum (ou time-slice) - um intervalo de tempo onde um processo está em execução. Ao fim de um quantum, chama-se o dispatcher. A implementação é trivial: uma FIFO de processos, o dispatcher faz push do processo que saiu de execução (ou de um novo que chegou) e pop daquele que vai colocar em execução. A grande desvantagem é que pode levar a tempos de resposta elevados em situações de muita carga.

- **Multi-lista**

  Uma forma de contornar a falha do round-robin é gerir os processos em várias FIFOs - cada uma correspondente a um nível de prioridade. Faz-se pop da lista com maior prioridade primeiro, permitindo, por exemplo, que processos intensivos em IO sejam mais responsivos. Uma adição importante a este sistema é ter um quantum variável para cada lista, permitindo assim que os com maior prioridade sejam mais responsivos.

- **Preempção**

  Retirar o CPU ao processo em execução logo que haja um mais prioritário, permitindo que os processos prioritários sejam mais responsivos. Isto é difícil de implementar, por isso, relaxa-se a definição. É então aplicada pseudo-preempção: o processo perde o CPU ao fim de um tempo mínimo.

1.7.1 Gestor de Processos no Unix

O Unix suporta dois tipos de prioridade:

- Prioridades para processos em modo núcleo:
  - vão de 0 a N (quanto mais pequeno, mais prioritário)
  - são calculadas dinamicamente em função do tempo de processador utilizado
  - escalonamento (quase) preemptivo
- Prioridades para processos em modo utilizador:
  - têm valores negativos (quanto mais negativo, mais prioritário)
  - são fixas, consoante o acontecimento que o processo está a tratar;
  - são sempre mais prioritárias que os processos em modo utilizador.

As prioridades do utilizador seguem o seguinte algoritmo:

- O CPU é sempre atribuído ao processo mais prioritário durante um quantum de 100ms (5 ticks)
- Round-Robin entre os processos mais prioritários
- A cada segundo (50 ticks) as prioridades são recalculadas de acordo com a seguinte fórmula:

$$prioridade = prioridade_{base} + \frac{CPUtime}{2}$$

$$CPUtime = \frac{CPUtime}{2}$$

Chamadas de sistema importantes relacionadas com o scheduling:

- `nice(int val)` : Muda o valor `nice` de um processo
  - Adiciona o valor `val` ao `nice` atual do processo, tornando-o menos prioritário se `val` for positivo.

  - Apenas superuser pode invocar com `val` negativo, tornando o processo mais prioritário
- `getpriority(int which, int id)`: retorna prioridade de um processo ou grupo de processos
- `setpriority(int which, int id, int prio)`: altera prioridade do processo ou grupo de processos

O Gestor de Processos no Unix recalcula a prioridade de todos os processos a cada segundo, ou seja, ao aumentar do número de processos o algoritmo de escalonamento Unix torna-se pouco eficiente.

1.7.2 Gestor de Processos no Linux

No Linux, o tempo é dividido em épocas. Uma época acaba quando todos os processos usaram o seu quantum disponível ou estão bloqueados.

O quantum e prioridade são atribuídos no início de cada época por:

$$quantum = quantum_{base} + \frac{quantum_{por\ usar\ epoca\ anterior}}{2}$$

$$prioridade = prioridade_{base} + quantum_{por\ usar\ epoca\ anterior} - nice$$

O valor do quantum pode ser mudado com chamadas de sistema.

As prioridades mais importantes são as com valor mais elevado.

1.7.3 Completely Fair Scheduler (CFS)

O CFS é o scheduler atualmente utilizado pelo Linux. Adiciona a cada processo um novo atributo `vruntime` que representa o seu tempo acumulado de execução em modo utilizador.

Um novo processo inicia com `vruntime` igual ao mínimo entre o `vruntime` dos processos ativos. Quando o processo perde CPU, o seu `vruntime` é incrementado com o tempo executado nesse quantum.

Os processos executáveis são guardados numa red-black tree ordenada por `vruntime`, que permite encontrar o processo mais prioritário em O(log n). O processo mais prioritário é o com menor `vruntime`.

2 Sistemas de Ficheiros

2.1 Ficheiros

Definimos um ficheiro como uma coleção de dados persistentes, geralmente relacionados, identificados por um nome.

Os vários ficheiros de um certo sistema estão normalmente organizados num sistema de ficheiros.

2.1.1 Nomes

Para aceder a um ficheiro temos de saber como referir ao SO a qual ficheiro estamos a querer aceder.

- Nomes Absolutos
  - Caminho de acesso desde a raiz (root, normalmente denominado /)
- Nomes Relativos
  - Caminho de acesso a partir do diretório corrente
  - O diretório corrente é mantido para cada processo como parte do seu contexto

2.2 Links

Um ficheiro pode ser conhecido por vários nomes, ou seja, é possível querermos associar um dado conjunto de dados a mais que um nome (eventualmente em diretorias diferentes).

- Hard link
  - Corresponde ao conceito de cópia de um ficheiro (sem cópia real dos dados).
  - Se apagarmos um ficheiro com vários hard links, o ficheiro continua a existir. Só será removido quando o ultimo hard link for apagado.
- Symbolic link
  - É um ficheiro (de tipo diferente) que contem o caminho de acesso para o ficheiro original.
  - Quando se apaga um symbolic link para um ficheiro, o ficheiro nunca é apagado.
  - Se o ficheiro original for apagado o link fica quebrado.

2.2.1 Programar com Ficheiros

TODO

2.3 Organização do Disco

Um disco está dividido em partições (e um sector de 512B, o MBR). Para qualquer partição do disco, existe sempre um `boot block`, que contém código (instruções) que vai ser carregado para RAM.

O grande desafio do sistema de ficheiros é conseguir organizar o volume para (de uma forma eficiente) armazenar os dados.

2.3.1 Organização em Lista e Diretório

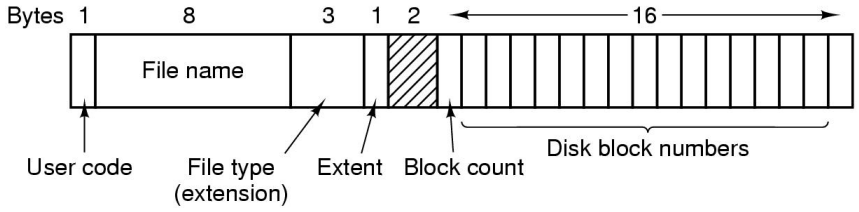
- **Organização em Lista**

  Neste modelo cada ficheiro é um nó numa lista ligada (os campos sendo o nome, o tamanho, o ficheiro seguinte e os dados)
  - **Vantagens**: forma simples e compacta de organizar um sistema de ficheiros
  - **Desvantagens**: tempo necessário para localizar um ficheiro, fragmentação de memória

- **Organização em Diretório**

  Neste modelo os metadados (nome e dimensão) são compactados numa diretoria, mantendo referências para a localização dos dados.
  - **Vantagens**: aumenta a eficiência da procura de um ficheiro pelo seu nome
  - **Desvantagens**: perda de funcionalidade para diminuir a fragmentação externa usando blocos

2.3.2 Sistema de Ficheiros do CP/M e MS-DOS



O CP/M é baseado na organização em diretório, cada entrada do diretório do sistema de ficheiros tinha 32B. Como cada bloco tinha 1 KB, os ficheiros tinham, no máximo, 16 KB.

Para aumentar a dimensão máxima do ficheiro podemos:

- aumentar o mapa de blocos, ou seja, tamanho das entradas (ineficiente para ficheiros pequenos)
- aumentar o tamanho dos blocos (maior fragmentação interna)

O MS-DOS possui uma estrutura de sistema de ficheiros semelhante ao CP/M, mas em vez de um mapa de blocos por ficheiro, no MS-DOS existe uma tabela de blocos global partilhada por todos os ficheiros.

2.3.3 Sistema de File Allocation Table

Num sistema de ficheiros FAT (designado FAT-Y, para n=Y), a partição contém três secções distintas:

- A tabela de alocação: um vetor com, no máximo, 2^n inteiros de n bits. Cada entrada contém:
  - `0` se o bloco com o esse índice está livre
  - `max` se esse é o último bloco de um cadeia de blocos de um ficheiro
  - `x` indica qual é o índice do próximo bloco com dados relativos um ficheiro
- Um diretório com os nome do ficheiro e um inteiro correspondente a um índice da FAT, para cada ficheiro presente no sistema.
- uma secção com o espaço restante dividido em blocos, de igual dimensão, para conter os dados dos ficheiros.

O tamanho máximo de um ficheiro em FAT-Y é de 2^{Y-1} blocos.

Desvantagens:

- Elevada dimensão da FAT quando os discos têm dimensões muito grandes.
- Não é possível manter tabelas desta dimensão em RAM, sendo preciso ler a FAT do disco, o que prejudica muito o acesso à cadeia de blocos de um ficheiro.

2.3.4 Organização com i-nodes

Este sistema cria uma estrutura de dados, um i-node, com informação relevante sobre o ficheiro (tipo de ficheiro, dono, datas de últimos acessos, permissões, dimensão, localizações dos blocos de dados).

Isto permite organizar o sistema de ficheiros como uma árvore ou hash table de i-nodes, e que várias entradas numa diretoria apontem para o mesmo ficheiro (referenciam o mesmo i-node).

Em cada partição, cada ficheiro é identificado univocamente pelo i-number (número do inode). Assim, as diretorias têm que ser apenas um mapeamento entre nomes e i-numbers.

Num descritor do volume, podemos encontrar a localização de tabela de i-nodes e a tabela de alocação. Dada a importância do descritor de volume (ou super-bloco), este tipicamente está replicado.

A tabela de alocação é um bitmap que indica se um bloco está livre ou não.

2.3.5 Sistema de Ficheiros EXT

TODO

- No Ext3 quer os ficheiros quer os diretórios têm um i-node associado que descreve os respetivos metadados.
- No Ext3 os diretórios guardam a associação entre os nomes dos ficheiros e os respetivos i-nodes (i-number)
- O número de ficheiros guardados num sistema de ficheiros Ext3 não pode ultrapassar o número de entradas na tabela de i-nodes

A dimensão máxima de um ficheiro é dado por:

$$t_{max} = B \times \left[12 + \frac{B}{R} + \left(\frac{B}{R}\right)^2 + \left(\frac{B}{R}\right)^3 \right]$$

B - tamanho do bloco em bytes
R - tamanho da referência para um bloco em bytes

2.3.6 Estruturas de Suporte

TODO

3 Processos e Tarefas

3.1 Processos

Um processo é uma entidade ativa controlada por um programa e que necessita de um processador para poder executar-se. Cada processo tem:

- Espaço de Endereçamento
  - Código do programa
  - Heap - Zona para os dados, variáveis globais e alocação dinâmica de estruturas de dados
  - Stack
- Reportório de Instruções
  - Instruction set da máquina do processador
  - Instruções do sistema operativo
- Estado interno
  - Guardam components como program counter, stack pointer, status register, ...

Os processos relacionam-se de forma hierárquica, um novo processo herda grande parte do contexto do processo pai.

Se um processo pai termina, os seus sub-processos continuam a executar-se e são adotados pelo processo de inicialização (`pid = 1`).

3.1.1 Criação do Processo

1. Reserva-se uma entrada na tabela `proc` (verificando se o utilizador não excedeu o número máximo de processos).
2. Atribui-se um `pid`.
3. Copia-se a imagem do processo pai (com copy-on-write, para aumentar a performance).

`pid_t fork()`
`Retorna 0 ao processo filho`
`Retorna pid do processo filho ao processo pai`
`Retorna -1 em caso de erro`

3.1.2 Terminação do Processo

1. Executar funções registadas pelo `atexit`.
2. Libertar recursos (ficheiros, diretoria, memória). **Não** elimina a `task_struct`.
3. Atualizar o ficheiro que regista a utilização do processador, memória e IO.
4. Enviar o sinal `SIGCHLD` ao pai.

`void exit(int status)`
`status: valor que é retornado ao processo pai`

Entre `exit` e `wait`, processo filho diz-se zombie, só depois de `wait` o processo é totalmente esquecido. Enquanto procura filhos zombies o processo pai bloqueia.

`int wait(int *status)`
`status: estado que é retornado ao processo pai pelo filho no exit`
`Retorna o pid do processo terminado`

3.1.3 Substituição do Processo

O `exec()` carrega um novo programa num processo já existente, em que:

1. Verifica que o processo existe e é executável
2. Copia os argumentos do exec para a pilha do núcleo (a pilha do utilizador vai ser destruída)
3. Liberta os dados referentes ao programa antigo
4. Reserva dados para o novo programa
5. Carrega o executável.
6. Copia os argumentos para a pilha do utilizador

O código a seguir a chamada à `exec()` só é executado caso a chamada sistema falhar. A área de dados e a pilha do programa atual são libertados caso a `exec()` tenha sucesso.

3.1.4 Exemplo

```
main () {
    int pid, status;
    pid = fork();
    if(pid == 0) {
        /* algoritmo do processo filho */
        exit(0);
    } else {
        /* o processo pai bloqueia à espera da terminação do processo filho */
        pid = wait(&status);
    }
}
```

3.2 Tarefas

Mecanismo simples para criar fluxos de execução independentes, partilhando um contexto comum.

Tarefas do mesmo processo partilham o código, o heap (variáveis globais e variáveis dinamicamente alocadas) e atributos do processo. **Não partilham** a stack, o estado dos registos do processador e os seus atributos específicos.

Vantagens:

- Criação e comutação entre tarefas do mesmo processo é mais leve.
- Tarefas podem comunicar através de memória partilhada.

Desvantagens:

- Não podem executar diferentes binários em paralelo.
- Não permitem o isolamento de bugs.

3.2.1 Interface POSIX

Criar uma tarefa
`int pthread_create(pthread_t *tid, const pthread_attr_t *attr, void *(*routine)(void*), void *arg)`

`pid: apontador para o identificador da tarefa`
`attr: atributos da tarefa`
`routine: função a executar`
`arg: ponteiro para argumentos da função`

`Retorna 0 em caso de sucesso` `Retorna -1 em caso de erro`

Termina a tarefa chamadora
`void pthread_exit(void *retval)`
`retval: ponteiro para valor a ser retornado`

Tarefa chamadora espera até a tarefa indicada ter terminado
`int pthread_join(pthread_t thread, void **retval)`
`thread: tarefa pela qual tarefa chamadora espera` `retval: ponteiro retornado pela tarefa terminada`

3.3 Memória Partilhada

3.3.1 Secção Crítica

Uma **secção crítica** é um bloco que deve ser executado de forma **indivisível** ou **atómica**.

Em programação concorrente, sempre que atividades concorrentes acedem a recursos partilhados, **é necessário efetuá-lo dentro de uma secção crítica**.

As secções críticas são delimitadas por um fecho e uma correspondente libertação para outras tarefas. Dizemos, para estas ações, que são o **lock** e **unlock** da secção crítica. Para este fim, utilizamos os **trincos lógicos**.

3.3.2 Mutex

Uma secção crítica fechada por um mutex só pode acedida quando este for aberto.

O uso de trincos diferentes para secções críticas independentes maximiza o paralelismo do programa.

Inicializar um trinco
`int pthread_mutex_init(pthread_mutex_t *mutex, pthread_mutexattr_t *attr)`
`mutex: trinco a ser inicializado` `attr: atributos do trinco`
`Retorna 0 em caso de sucesso` `Retorna -1 em caso de erro`

Fechar ou abrir um trinco
`int pthread_mutex_[un]lock(pthread_mutex_t *mutex)`
`mutex: trinco que serálocked ou unlocked`
`Retorna 0 em caso de sucesso` `Retorna -1 em caso de erro`

Tentar bloquear um trinco
`int pthread_mutex_trylock(pthread_mutex_t *mutex)`
`mutex: trinco que se tentarábloquear`
`Retorna 0 em caso de sucesso` `Retorna -1 em caso de erro`

Bloquear uma trinco por um dado tempo
`int pthread_mutex_timedlock(pthread_mutex_t *mutex, const struct timespec *timeout)`
`mutex: trinco a ser inicializado` `timeout: estrutura que indica o tempo a fazer lock`
`Retorna 0 em caso de sucesso` `Retorna -1 em caso de erro`

Destruir um trinco
`int pthread_mutex_destroy(pthread_mutex_t *mutex)`
`mutex: trinco a ser destruído`
`Retorna 0 em caso de sucesso` `Retorna -1 em caso de erro`

3.3.3 Trinco de Leitura-Escrita

Permitem fechar a secção crítica para ler ou para escrever.

- Os escritores só podem aceder em exclusão mútua.
- Os leitores podem aceder simultaneamente com outros leitores, mas em exclusão mútua com os escritores.

Vantajoso quando acessos de leitura às secções críticas são dominantes.

Fechar um trinco para leitura
`int pthread_rwlock_rdlock(pthread_rwlock_t *lock)`
`lock: trinco que seráfechado`
`Retorna 0 em caso de sucesso` `Retorna -1 em caso de erro`

Fechar um trinco para escrita
`int pthread_rwlock_wrlock(pthread_rwlock_t *lock)`
`lock: trinco que seráfechado`
`Retorna 0 em caso de sucesso` `Retorna -1 em caso de erro`

Abrir um trinco
`int pthread_rwlock_unlock(pthread_rwlock_t *lock)`
`lock: trinco que seráaberto`
`Retorna 0 em caso de sucesso` `Retorna -1 em caso de erro`

As funções de inicialização (`pthread_rwlock_init`) e de destruição (`pthread_rwlock_destroy`) são semelhantes às dos mutexes.

3.3.4 Semáforos

Um semáforo é uma primitiva de sincronização mais genérica, que pode ser visto como um contador atómico. É inicializado com o número de threads que podem entrar numa secção crítica (ao entrarem fazem `sem_wait` e decrementam o contador) e ao sair fazem `sem_post` (incrementando o contador).

- Para bloquear uma tarefa T1 até o acontecimento de um evento detetado pela tarefa T2, pode-se usar um semáforo inicializado com 0, onde T1 espera e T2 assinala.

TODO

3.3.5 Variáveis de Condição

TODO

3.4 Troca de Mensagem

3.4.1 Pipes

Um pipe é um canal byte stream, unidirecional, que liga dois processos. Os descritores de um pipe são semelhantes aos de um ficheiro, mas são internos ao processo e podem ser transmitidos aos processos filhos através do mecanismo de herança.

As duas extremidades de um pipe criado pelo pai ficam automaticamente abertas no processo filho quando se efetuar o fork.

Criar um Pipe
`int pipe(int *fds)`
`fds[0]: descritor aberto para leitura` `fds[1]: descritor aberto para escrita`
`Retorna 0 em caso de sucesso` `Retorna -1 em caso de erro`

3.4.2 Named Pipes

Um named pipe é, como o nome indica, um pipe identificado por um nome. Permite que dois processos que não têm qualquer relação hierárquica comuniquem. Um named pipe comporta-se externamente como um ficheiro, existindo uma entrada na diretoria correspondente, mas **não é persistente**.

Um processo que abra uma extremidade do canal **bloqueia** até que pelo menos 1 processo tenha aberto a outra extremidade.

Criar um Named Pipe
`int mkfifo(const char *pathname, mode_t mode)`
`pathname: caminho do pipe` `mode: permissões do ficheiro`
`Retorna 0 em caso de sucesso` `Retorna -1 em caso de erro`

Eliminar um Named Pipe
`int unlink(const char *pathname)`
`pathname: caminho do pipe`
`Retorna 0 em caso de sucesso` `Retorna -1 em caso de erro`

Exemplo

```
/* Cliente */

main() {
    int fcli, fsrv;
    char buf[BUFSIZE];

    if((fsrv = open("/tmp/server",
       O_WRONLY)) < 0) exit(1);
    if((fcli = open("/tmp/client",
       O_RDONLY)) < 0) exit(1);

    /* produce message */
    write(fsrv, buf, strlen(buf));

    read(fcli, buf, BUFSIZE);
    /* process message */

    close(fsrv);
    close(fcli);
}
```

```
/* Servidor */

main () {
    int fcli, fsrv;
    char buf[BUFSIZE];

    unlink("/tmp/servidor");
    unlink("/tmp/cliente");

    if(mkfifo("/tmp/server", 0777) <
       0) exit (1);
    if(mkfifo("/tmp/client", 0777) <
       0) exit (1);

    if((fsrv = open("/tmp/server",
       O_RDONLY)) < 0) exit(1);
    if((fcli = open("/tmp/client",
       O_WRONLY)) < 0) exit(1);

    for (;;) {
        if(read(fsrv, buf, BUFSIZE) <=
            0) break;
        /* process message */

        /* produce message */
        write (fcli, buf, strlen(buf))
           ;
    }

    close(fsrv);
    close(fcli);

    unlink("/tmp/server");
    unlink("/tmp/client");
}
```

3.4.3 Sockets

Um socket é uma extremidade de um canal de comunicação, um socket pode ter um nome (socket address) associado para permitir que outros processos o referenciem. Num exemplo com dois processos a comunicar, existirão (pelo menos) dois sockets.

Criar um socket
`int socket(int domain, int type, int protocol)`
`domain: domínio da comunicação (AF_UNIX, AF_INET)` `type: tipo de socket (SOCK_STREAM, SOCK_DGRAM)`
`Retorna o identificador do socket em caso de sucesso` `Retorna -1 em caso de erro`

Associar um nome (endereço de comunicação) a um socket já criado
`int bind(int sockfd, struct sockaddr *addr, socklen_t addrlen)`

`sockfd: file descriptor do socket`
`addr: endereço a associar ao socket`
`addrlen: tamanho da estrutura apontada por addr`

`Retorna 0 em caso de sucesso` `Retorna -1 em caso de erro`

Sockets sem Ligação (datagram)

- Modelo de comunicação tipo correio.
- Canal sem ligação, bidirecional, não fiável, interface tipo mensagem.

Envia uma mensagem para o endereço especificado
`int sendto(int sockfd, char *msg, int length, int flag, struct sockaddr *dest, int addrlen)`

`sockfd: file descriptor do socket`
`msg: mensagem a enviar`
`length: tamanho da mensagem`
`dest: endereço do socket de destino`
`addrlen: tamanho da estrutura apontada por dest`

`Retorna o número de caracteres enviados` `Retorna -1 em caso de erro`

Recebe uma mensagem e devolve o endereço do emissor
`int recvfrom(int sockfd, char *msg, int length, int flag, struct sockaddr *orig, int * addrlen)`

`sockfd: file descriptor do socket`
`msg: mensagem a receber`
`length: tamanho da mensagem`
`orig: endereço do socket de origem`
`addrlen: tamanho da estrutura apontada por orig`

`Retorna o número de caracteres recebidos` `Retorna -1 em caso de erro`

Sockets com Ligação (stream)

- Modelo de comunicação tipo diálogo.
- Canal com ligação, bidirecional, fiável, interface tipo sequência de octetos.

É estabelecido um canal de comunicação entre o processo cliente e o servidor. O servidor pode gerir múltiplos clientes, mas dedica a cada um deles uma atividade independente. O servidor pode ter uma política própria para atender os clientes.

Indica que se vão receber ligações neste socket (Server)
`int listen (int sockfd, int backlog)`
`sockfd: file descriptor do socket` `backlog: número máximo de conexões em espera`
`Retorna 0 em caso de sucesso` `Retorna -1 em caso de erro`

Estabelece uma ligação (Client)
`int connect(int sockfd, struct sockaddr *addr, socklen_t addrlen)`

`sockfd: file descriptor do socket`
`addr: endereço do socket a conectar`
`addrlen: tamanho da estrutura apontada por addr`

`Retorna 0 em caso de sucesso` `Retorna -1 em caso de erro`

Aceita uma ligação (Server)
`int accept(int sockfd, struct sockaddr *addr, int *addrlen)`

`sockfd: file descriptor do socket`
`addr: endereço do socket que se conecta`
`addrlen: tamanho da estrutura apontada por addr`

`Retorna o identificador do socket aceite em caso de sucesso` `Retorna -1 em caso de erro`

É possível um servidor fazer espera múltipla usando o `select`. Esta função bloqueia o processo até que um descritor tenha um evento associado ou expire o alarme.

`int select(int nfds, fd_set *read, fd_set *write, fd_set *error, struct timeval *timeout)`

`nfds: número de file descriptors`
`read: conjunto de file descriptors de leitura`
`write: conjunto de file descriptors de escrita`
`error: conjunto de file descriptors de erro`
`timeout: intervalo de espera`

`Retorna número de bits em caso de sucesso` `Retorna -1 em caso de erro`

Exemplo

```
int main() {
    int server_fd, client_fd;
    struct sockaddr_in addr;
    socklen_t addrlen = sizeof(addr);

    if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        exit(0);
    }

    /* set socket options and address... */

    if (bind(server_fd, (struct sockaddr*) &addr, sizeof(addr)) < 0) {
        exit(0);
    }

    if (listen(server_fd, 3) < 0) {
        exit(0);
    }

    client_fd = accept(server_fd, (struct sockaddr*) &addr, &addrlen)
    if (client_fd < 0) {
        exit(0);
    }

    int n = read(client_fd, buffer, BUFSIZE);
    /* do something with buffer */

    /* write response to buffer*/
    write(client_fd, buffer, strlen(buffer));

    close(client_fd);   /* closing the connected socket */
    close(server_fd);   /* closing the listening socket */

    return 0;
}
```

3.5 Sinais

TODO

Diferentes semânticas dos signals:

- System V
  - A associação de uma rotina a um signal é apenas efetiva para uma ativação
  - Depois de receber o signal, o tratamento passa a ser novamente o por omissão
  - Necessário restabelecer a associação na primeira linha da rotina de tratamento (problema se houver receção sucessiva de signals)
- BSD
  - Associação signal-rotina **não** é desfeita após ativação
  - A receção de um novo signal é inibida durante a execução da rotina de tratamento

- É uma rotina normal que fica referenciada na tabela de tratamento de signals do processo. Quando o processo vai passar de modo núcleo a modo utilizador, é testado se existe um signal pendente e por manipulação dos stacks núcleo e utilizador a rotina é posta a correr em modo utilizador
- Um processo pode ter associado um tratamento por omissão para todos os signals

4 Gestão de Memória

O objetivo da gestão de memória é gerir o espaço de endereçamento dos processos, assegurando que cada processo dispõe da memória que precisa, garantir que cada processo só acede à memória a que tem direito e otimizar desempenho dos acessos.

4.1 Gestor de Memória

Espaço de Endereçamento - Conjunto de endereços de memória disponíveis para um processo.

O processo endereça a memória física. Esta abordagem é feita tipicamente em sistemas embebidos (e em arquiteturas antigas). No entanto, coloca problemas de portabilidade e de concorrência.

4.2 Endereçamento Virtual

Neste modelo, o espaço de endereçamento é virtual, logo é feita uma tradução para os endereços reais.

4.2.1 Segmentação

A segmentação consiste na divisão dos programas em segmentos lógicos que refletem a sua estrutura funcional (rotinas, módulos, código, dados, pilha, etc).

Através da segmentação, podemos carregar segmentos mais eficientemente e proteger os segmentos. Existe uma tabela de segmentos que faz a tradução para a memória real.

Em sistemas segmentados, um endereço virtual é um par (segmento, deslocamento) em que segmento define uma entrada na tabela de segmentos. Uma entrada da tabela de segmentos é constituída por:

P	Indica se o segmento correspondente a esta entrada está presente na memória principal
Prot	Definem as proteções do segmento em causa
Limite	Indica o número de endereços que constituem este segmento
Base	Endereço na memória principal em que está a informação relativa ao segmento

4.2.2 Páginação

A paginação consiste em dividir a memória em blocos de tamanho fixo, chamados páginas, que são a unidade de carregamento entre a memória primária e secundária.

Em sistemas paginados, um endereço virtual será então um par (página, deslocamento) em que página define uma entrada na tabela de páginas. Uma entrada da tabela de páginas é constituída por:

P	Indica se a página correspondente a esta entrada está presente na memória principal
R	Indica se a página foi lida recentemente
M	Indica se a página foi escrita recentemente
Prot	Definem as proteções da página em causa
Base	Endereço na memória principal em que está a informação relativa à página

A dimensão das páginas influencia:

- A fragmentação interna (não existe fragmentação externa com páginas)
- A dimensão das tabelas de páginas (páginas maiores, tabelas menores)
- O número de faltas de páginas
- Tempo de transferência de páginas

4.2.2.1 Otimização de Tradução de Endereços

TODO

- Cada processo tem uma tabela de páginas distinta.
- O TLB deve ser limpo pelo SO quando é efetuada a comutação entre processos diferentes.
- Usar um quantum menor implica "resets" mais frequentes do TLB para processos "CPU-bound"
- Quando existe uma comutação de processo, o núcleo altera os registos do Memory Management Unit para apontar para a tabela de páginas do processo
- Sempre que houver um Page fault (página não presente em RAM), o núcleo é acordado.
- Os processos podem ter o mesmo endereço virtual. As bases das respetivas entradas na tabela de páginas somadas ao deslocamento é que definem o endereço real

4.2.2.2 Tabelas de Páginas Multi-Nível

TODO

É então necessária uma forma de endereçar as páginas sem consumir tanta memória. É usada uma tabela de páginas multi-nível. Existe uma tabela de páginas de nível 1, que endereça páginas que, elas próprias, consistem em tabelas de páginas. Esta solução resolve o problema apresentado, garantindo que só estão em memória tabelas de páginas correspondentes às páginas que estão de facto a ser utilizadas pelo processo.

4.3 Partilha de Memória entre Processos

Ao criar um processo novo, diz-se que se duplicam os recursos para o processo filho. Na verdade, duplica-se a página de tabelas, e partilham-se as páginas. Utiliza-se o copy-on-write para manter a semântica.

Copy-on-Write - Técnica para preguiçosamente duplicar dados. Permite acelerar a execução do `fork()`.

1. Aloca uma nova tabela de páginas para o processo filho e copia o conteúdo da tabela do pai.

2. Nas entradas da tabela (tanto do filho como do pai) retira permissão de escrita e ativa o bit C-o-W.
3. Quando o pai ou o filho tentam escrever é levantada uma exceção de violação de permissões de acesso à pagina.
4. Ao detetar a exceção e o bit CoW com valor 1, o OS aloca uma nova página, para onde copia o conteúdo da página partilhada.
5. Atualiza a entrada da tabela do processo onde ocorreu a exceção com a base da nova página e novas permissões (escrita ativada, CoW desativado);

4.4 Algoritmos de Gestão de Memória

O sistema operativo tem de tomar decisões sobre várias operações sobre a memória:

- **Alocação**: onde colocar um bloco na memória primária
- **Transferência**: quando transferir um bloco de memória secundária para memória primária e vice-versa
- **Substituição**: qual o bloco a retirar da memória primária

4.4.1 Alocação

TODO

4.4.2 Transferência

TODO

Há três abordagens para a transferência de segmentos:

- **on request**: o programa ou o sistema operativo determinam quando se deve carregar o bloco em memória principal.
- **on demand**: o bloco é acedido e gera-se uma falta (de segmento ou de página), sendo necessário carregá-lo para a memória principal.
- **prefetching**: o bloco é carregado na memória principal pelo sistema operativo porque este considera fortemente provável que ele venha a ser acedido nos próximos instantes. Isto é normalmente feito de acordo com o princípio da localidade de referência.

4.4.3 Substituição

A heurística para o algoritmo de substituição ótimo é que devemos retirar a página cujo próximo pedido seja mais distante no tempo. Para estimar isto, vamos medir o uso recente das páginas. Para isto podemos usar vários sistemas:

- FIFO (eficiente, mas não atende ao grau de utilização das páginas)
  - associar a cada entrada da tabela de páginas um timestamp de quando foi colocada em RAM.

  - tirar a entrada que está em RAM há mais tempo
- NRU (Not Recently Used)
  - a UGM coloca `R = 1` quando há leitura na página e `M = 1` quando há escrita.
  - o paginador percorre regularmente as tabelas de páginas e coloca o bit `R = 0`.
  - obtemos assim 4 grupos de páginas
    * `0 (R = 0, M = 0)`: Não referenciada, não modificada.
    * `1 (R = 0, M = 1)`: Não referenciada, modificada.
    * `2 (R = 1, M = 0)`: Referenciada, não modificada.
    * `3 (R = 1, M = 1)`: Referenciada, modificada.
  - libertam-se primeiro as páginas dos grupos de número mais baixo.
- LRU (Least Recently Used)
  - em cada entrada na tabela de páginas é mantido um bit R e um valor de Idade.
  - o paginador faz uma ronda regular pelas páginas, aumentando a idade daquelas com o bit `R = 0`.
  - sempre que uma página é acedida (leitura ou escrita), a UGM coloca `R = 1` e a sua idade é anulada.
  - quando uma página atinge uma idade determinada, esta passa para a lista de páginas que podem ser transferidas.

4.5 Comparação entre Segmentação e Paginação

	Segmentação	Paginação
Vantagens	• adapta-se à estrutura lógica dos programas. • permite a realização de sistemas simples sobre hardware simples. • permite realizar eficientemente as operações que agem sobre segmentos inteiros.	• o programador não tem que se preocupar com a gestão de memória. • os algoritmos de reserva, substituição e transferência são mais simples e eficientes. • o tempo de leitura de uma página de disco é razoavelmente pequeno. • a dimensão dos programas é virtualmente ilimitada.
Desvantagens	• o programador tem de ter sempre algum conhecimento dos segmentos subjacentes. • os algoritmos tornam-se bastante complicados em sistemas mais sofisticados. • o tempo de transferência de segmentos em memória principal e disco torna-se incomportável para segmentos muito grandes. • a dimensão máxima dos segmentos é limitada.	• o hardware é mais complexo (por exemplo, instruções precisam de ser recomeçáveis). • operações sobre segmentos lógicos são mais complexos e menos elegantes, pois têm de ser realizadas sobre um conjunto de páginas. • o tratamento das faltas de páginas representa uma sobrecarga adicional de processamento. • tamanho potencial das tabelas de páginas.

4.6 Gestão de Memória em Unix/Linux

TODO

4.6.1 Unix

TODO

4.6.2 Linux

TODO