SSC0640 - Sistemas Operacionais I

Trabalho 3
Simulador de gerenciamento de memória virtual com paginação.

Grupo: gso03
Higor Tessari - nºUSP: 10345251
Lucas Tavares dos Santos - nºUSP: 10295180
Renata Oliveira Brito - nºUSP: 10373663

Link dos códigos no gitHub

https://github.com/lucast98/Sistemas-Operacionais-gso3/tree/master/Projeto%203

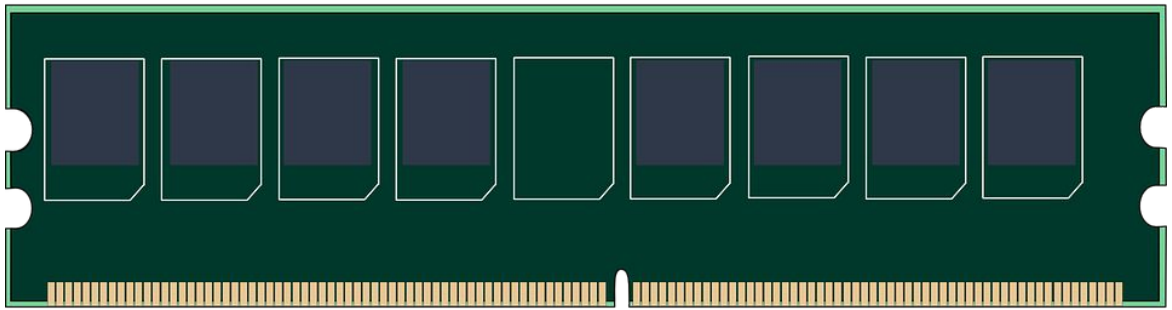
Link da explicação em vídeo

https://www.youtube.com/watch?v=7sQq0rkbLXI&t=2750s

Gerenciamento de Memória o que é?

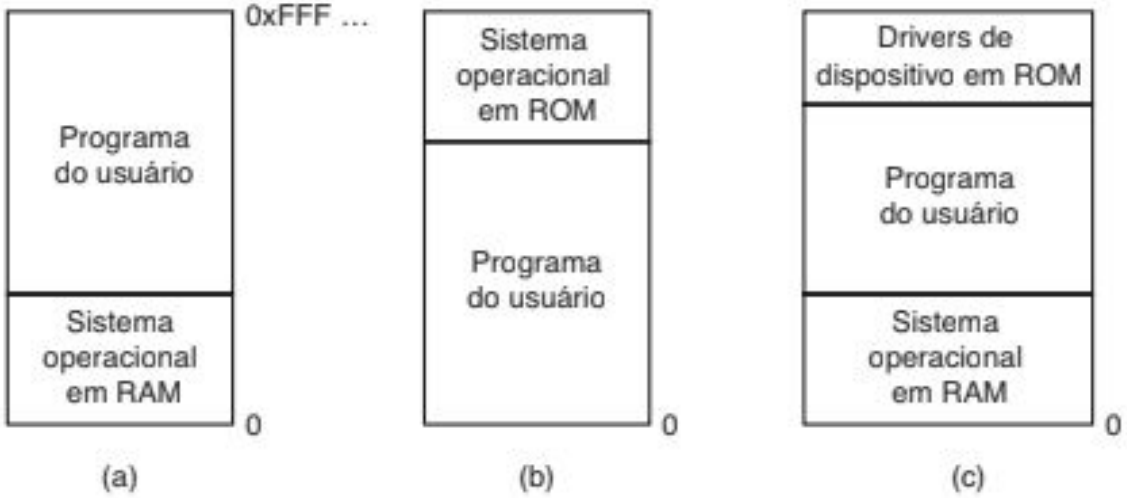
O gerenciamento de memória de memória é a parte do sistema operacional responsável pela gerência da hierarquia de memórias, que inclui a distribuição variável de bytes na execução de processos e armazenamento.

Ao longo do tempo os computadores realizaram essas operações de diferentes formas movendo conteúdo entre as células de memórias física, mas a dificuldade de execução múltipla de programas, fez o gerenciamento de memória evoluir até o que conhecemos hoje.



Gerenciamento de Memória

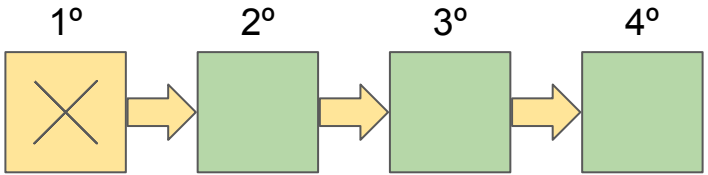
Dentre as diversas possibilidades existentes temos a seguir três maneiras simples de organizar a memória em um sistema operacional:



Trabalho sugerido

Criação de um simulador para ler um arquivo que contenha comandos de processos, além de informar o tamanho das páginas e quadro de páginas, tamanho do endereço lógico em bits, tamanho da memória física, o limite da memória secundária, o tamanho da imagem dos processos, um algoritmo de substituição de páginas como LRU e FIFO.

O algoritmo LRU (Last Recently Used) retira a página que não tem sido referenciada por uma maior fatia de tempo. Enquanto o FIFO (First-In First-Out) aloca as páginas de forma circular, retirando a página de acordo com sua alocação.



Main - Bibliotecas

<stdlib.h> que contém funções de alocação de memória e controle de processos.

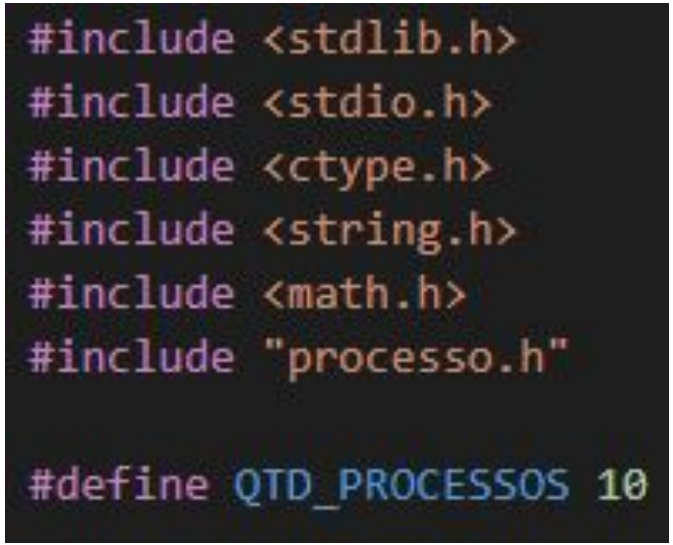
<stdio.h> que contém várias funções de entrada e saída, como exemplo as funções de printf do nosso programa.

<ctype.h> que contém várias funções para mapeamento e testes de caracteres.

<string.h> que contém várias funções para manipulação de strings.

<math.h> que contém funções matemáticas básicas.

“processo.h” que contém as estruturas e funções utilizadas neste projeto.



Main - Pressione para continuar

- A quantidade máxima de processos foi definida como 100.
- A função pressEnter é responsável por limpar o buffer do teclado e verificar se a tecla “ENTER” foi pressionada.

```c
#define QTD_PROCESSOS 100 //quantidade maxima de processos

/** Executar com ./nome_exec arquivoSimular.txt */

/** Limpa o buffer do teclado e verifica se enter foi apertado */
void pressEnter(FILE* in, int *first){
    int c;
    while((c = fgetc(in)) != EOF && c != '\n');
    if(*first == 1){
        *first = 0;
        while((c = fgetc(in)) != EOF && c != '\n');
    }
}
```

Main - Obtenção do número decimal

- Função para obter valores decimais do modelo de arquivo utilizado neste projeto, que estão entre parênteses.
- Pode retornar:
  - Endereços lógicos, para acessos de leitura e escrita.
  - Tamanho, para criação de processos.
  - Operando, para instruções a serem executadas pela CPU.
  - Dispositivo, para instruções de I/O.

```c
/** Obtem o valor entre parenteses no arquivo */
int getDec(char mode, char *dec){
    int valor = 0;
    int i, j=0;
    if(mode == 'R')
        i=1;
    else
        i=2;

    while(dec[i] != ')'){
        i++;
    }
    i--;
    while(i > 0){
        valor += (dec[i]-48)*pow(10, j);
        i--;
        j++;
    }
    return valor;
}
```

Main - Divisão das linhas de comando

- Função que divide as linhas de comando de acordo com sua finalidade.
- Comandos são divididos da seguinte forma:
  - PID - Tipo de acesso - Endereço/Tamanho/etc…
- Função strtok.

```c
/** Divide as linhas de comandos em cada uma de suas funcoes */
void splitString(char *str, char *pNumber, char *mode, char *op){
    char *split;
    split = strtok(str, " "); //pega o numero do processo do arquivo
    strcpy(pNumber, split);
    split = strtok(NULL, " ");
    *mode = *split;
    split = strtok(NULL, " ");
    strcpy(op, split);
}
```

Main - Obtenção do PID

- Função para obter o PID que está especificado na linha de comando.
- Como o PID está representado por uma string “P+número de processo”, deve-se extrair esse número dessa cadeia de caracteres.

```
/** Obtem o PID a partir do comando */
int getPID(char *pNumber){
    int pid = 0, i, j;
    if(pNumber[0] != 'P')
        return -1; //erro
    i = strlen(pNumber) - 1;
    j = i;
    while(i != 0){
        pid += (pNumber[i] - 48) * pow(10,(j-i));
        i--;
    }
    return pid;
}
```

Main - Variáveis

- Variáveis presentes na main:
  - `int pid` => Identificador de processo.
  - `int qtdPag` => Quantidade de páginas.
  - `int maxEnd` => Quantidade máxima de endereços possíveis.
  - `int memLivre` => Memória secundária livre.
  - `int qtdProc` => Quantidade atual de processos.
  - `int first` => Indica que o primeiro comando foi utilizado.

```c
int main(int argc, char const *argv[]){

    int pid;
    int qtdPag;
    int maxEnd; //quantidade maxima de enderecos possiveis
    int memLivre; //memoria secundaria livre
    int qtdProc; //quantidade atual de processos
    int first = 1; //indica que é o primeiro comando

    FILE *fp;
    char str[60]; //armazena a linha do arquivo
    char pNumber[5]; //numero do processo
    char mode; //tipo de operação
    char op[10]; //tamanho/operando/dispositivo
    const char *fileName = argv[1]; //nome do arquivo de entrada

    /** Variaveis de configuracao de mecanismos associados à memoria virtual */
    int page_size; //representa o tamanho das paginas e quadros de paginas
    int logic_size; //representa o tamanho em bits do endereço logico
    int real_size; //representa o tamanho da memoria fisica
    int sec_size; //representa o tamanho maximo da memoria secundaria
    char subs_alg; //representa o algoritmo de substituicao a ser utilizado
```

Main - Variáveis

- Variáveis presentes na main:
  - `FILE *fp` => Ponteiro para o arquivo com os comandos.
  - `char str[60]` => Armazena cada uma das linhas do arquivo.
  - `char pNumber[5]` => Armazena o identificador de processo.
  - `char mode` => Armazena o tipo de operação.
  - `char op[10]` => Armazena o endereço/tamanho/etc...
  - `const char *fileName` => Recebe o nome do arquivo de entrada.

```c
int main(int argc, char const *argv[]){

    int pid;
    int qtdPag;
    int maxEnd; //quantidade maxima de enderecos possiveis
    int memLivre; //memoria secundaria livre
    int qtdProc; //quantidade atual de processos
    int first = 1; //indica que é o primeiro comando

    FILE *fp;
    char str[60]; //armazena a linha do arquivo
    char pNumber[5]; //numero do processo
    char mode; //tipo de operação
    char op[10]; //tamanho/operando/dispositivo
    const char *fileName = argv[1]; //nome do arquivo de entrada

    /** Variaveis de configuracao de mecanismos associados à memoria virtual */
    int page_size; //representa o tamanho das paginas e quadros de paginas
    int logic_size; //representa o tamanho em bits do endereço logico
    int real_size; //representa o tamanho da memoria fisica
    int sec_size; //representa o tamanho maximo da memoria secundaria
    char subs_alg; //representa o algoritmo de substituicao a ser utilizado
```

Main - Variáveis

- Variáveis presentes na main:
  - `int page_size` => Tamanho das páginas e quadros.
  - `int logic_size` => Tamanho em bits do endereço lógico.
  - `int real_size` => Tamanho da memória física.
  - `int sec_size` => Tamanho máximo da memória secundária.
  - `char subs_alg` => Representa o algoritmo de substituição que será utilizado.

```c
int main(int argc, char const *argv[]){

    int pid;
    int qtdPag;
    int maxEnd; //quantidade maxima de enderecos possiveis
    int memLivre; //memoria secundaria livre
    int qtdProc; //quantidade atual de processos
    int first = 1; //indica que é o primeiro comando

    FILE *fp;
    char str[60]; //armazena a linha do arquivo
    char pNumber[5]; //numero do processo
    char mode; //tipo de operação
    char op[10]; //tamanho/operando/dispositivo
    const char *fileName = argv[1]; //nome do arquivo de entrada

    /** Variaveis de configuracao de mecanismos associados à memoria virtual */
    int page_size; //representa o tamanho das paginas e quadros de paginas
    int logic_size; //representa o tamanho em bits do endereço logico
    int real_size; //representa o tamanho da memoria fisica
    int sec_size; //representa o tamanho maximo da memoria secundaria
    char subs_alg; //representa o algoritmo de substituicao a ser utilizado
```

Main - Informações desejadas

- São requeridas as seguintes informações:
  - Tamanho das páginas e quadros.
  - Tamanho em bits do endereço lógico.

```c
/** Abre o arquivo e verifica se está OK */
fp = fopen(fileName, "r");
if(fp == NULL){
    printf("Erro\n");
    return -1;
}

/** Opções de configuração de mecanismos associados à memória virtual */
do{
    printf("Digite o tamanho das paginas e quadros de pagina: ");
    scanf("%d", &page_size);
    if(page_size <= 0)
        printf("Digite um tamanho de pagina maior que zero.\n");
}while(page_size <= 0);

do{
    printf("Digite o tamanho em bits do endereco logico: ");
    scanf("%d", &logic_size);
    if(logic_size <= 0)
        printf("Digite um tamanho em bits maior que zero.\n");
}while(logic_size <= 0);
```

Main - Informações desejadas

- São requeridas as seguintes informações:
  - Tamanho da memória física.
    - Deve ser múltiplo do tamanho do quadro.

```
do{
    printf("Digite o tamanho da memoria fisica que deve ser multiplo de tamanho da moldura (quadro): ");
    scanf("%d", &real_size);
    if(real_size <= 0)
        printf("Digite um tamanho de memoria fisica que seja maior que zero.\n");
    else{
        if(real_size % page_size != 0)
            printf("Digite um tamanho de memoria que seja multiplo do quadro.\n");
    }
}while(real_size % page_size != 0 || real_size <= 0);
```

Main - Informações desejadas

- São requeridas as seguintes informações:
  - Tamanho máxima da memória secundária.
    - Deve ser maior ou igual a memória principal.

```
do{
    printf("Digite o tamanho maximo da memoria secundaria: ");
    scanf("%d", &sec_size); //tamanho da memoria virtual
    if(sec_size <= 0)
        printf("Digite um tamanho de memoria secundaria maior que zero.\n");
    else{
        if(sec_size < real_size)
            printf("Recomenda-se que a memoria secundaria seja maior ou igual a principal.\n");
    }
}while(sec_size < real_size || sec_size <= 0);
```

Main - Informações desejadas

- São requeridas as seguintes informações:
  - Algoritmo de substituição a ser utilizado.
    - Deve ser escolhido entre LRU e FIFO, com L e F, respectivamente.

```
do{
    printf("Digite o algoritmo de substituicao a ser utilizado (L para LRU ou F para FIFO): ");
    scanf(" %c", &subs_alg);
    subs_alg = toupper(subs_alg); //deixa sempre maiusculo
    if(subs_alg != 'L' && subs_alg != 'F')
        printf("Digite um algoritmo de substituicao valido.\n");
}while(subs_alg != 'L' && subs_alg != 'F');
```

Main - Criação da simulação

- A quantidade de páginas é dada pela divisão entre o tamanho da memória secundária e o tamanho das páginas.
- Maior endereço possível é dado por 2^n-1, sendo n a quantidade de bits do endereço lógico.
- São criadas duas memórias:
  - Virtual.
  - Principal.
- É criado um vetor de processos, limitado pela quantidade de processos pré-definida.

```
qtdPag = sec_size/page_size; //unidades de tamanho fixo no dispositivo secundario
maxEnd = pow(2, logic_size); //endereco maximo permitido (por causa dos bits)
qtdProc = 0; //nao tem processo nenhum

/** Cria memoria virtual e principal */
Memoria *memVirtual = criaMemoria(sec_size, page_size);
Memoria *memPrincipal = criaMemoria(real_size, page_size);
memLivre = real_size;

/** Processos criados */
Processo **pro = (Processo**) malloc(QTD_PROCESSOS * sizeof(Processo *));

printf("Iniciando simulador de gerenciamento de memoria virtual.\n");
printf("Arquivo de entrada: %s\n", fileName);
```

Memoria.h - Struct da memória

A struct quadroMemoria define o quadro da memória, contendo o identificador do processo (PID), o número da página (numPag) e algum elemento que colocamos para que no momento que escrever ou ler ter esse elemento, ele deve ser um inteiro (elemento).

Na struct Memoria (diferenciaremos entre principal e virtual durante a criação dos processos, as duas terão os mesmos atributos) temos um vetor de quadros (quadros), o tamanho da memória (tam), um vetor em binário dizendo quantos quadros livres temos (quadrosLivres) e o seu tamanho (qtd_quadrosLivres).

```
/** Struct do quadro de memoria
typedef struct quadroMemoria{
    int PID; //identificador de
    int numPag; //numero de pagi
    int elemento; //elemento da
} quadroMemoria;

/** Struct da memoria virtual */
typedef struct memoria{
    quadroMemoria *quadros; //qu
    int tam;
    int *quadrosLivres; //vetor
    int qtd_quadrosLivres; //qtd
} Memoria;
```

Memória - Criar Memória

A função criarMemoria tem como objetivo retomar a memória criada com as informações passadas, como seu tamanho (tam) e o tamanho de suas páginas (tamPag).

Alocaremos então o espaço na memória do computador para a memória do simulador (mem), para os quadros e atualizaremos os valores do conteúdo da memória, iniciando os quadros livres com 1.

```
/** Funcao para criar uma nova estrutura de memoria */
Memoria* criaMemoria(int tam, int tamPag){
    int qtdQuad = tam/tamPag;
    Memoria *mem = (Memoria*) malloc(sizeof(Memoria));
    mem->quadros = (quadroMemoria*) calloc(tam, sizeof(quadroMemoria));
    mem->tam = tam;
    mem->quadrosLivres = (int*) malloc(tam*sizeof(int));
    for(int i = 0; i < tam; i++){
        mem->quadrosLivres[i] = 1;
        mem->quadros[i].numPag = -1; //nao tem nada
    }
    mem->qtd_quadrosLivres = qtdQuad;
    return mem;
}
```

Main - Leitura de comandos

- Lê cada linha do arquivo:
  - Se começar por um ‘#’, pula para a próxima.
  - Caso contrário, divide a string com a função splitString e obtém o PID com o auxílio da função getPID.

```
/** Parte em que o arquivo será lido e realizará a simulação */
while(fgets(str, 60, fp) != NULL) {
    /* lendo arquivo */
    if(str[0] == '#')
        continue; //evita comentario
    printf("\n--> Comando: %s", str);
    splitString(str, pNumber, &mode, op); //divide linha em tres partes (numero de processo-modo-tamanho/operando)
    pid = getPID(pNumber);
```

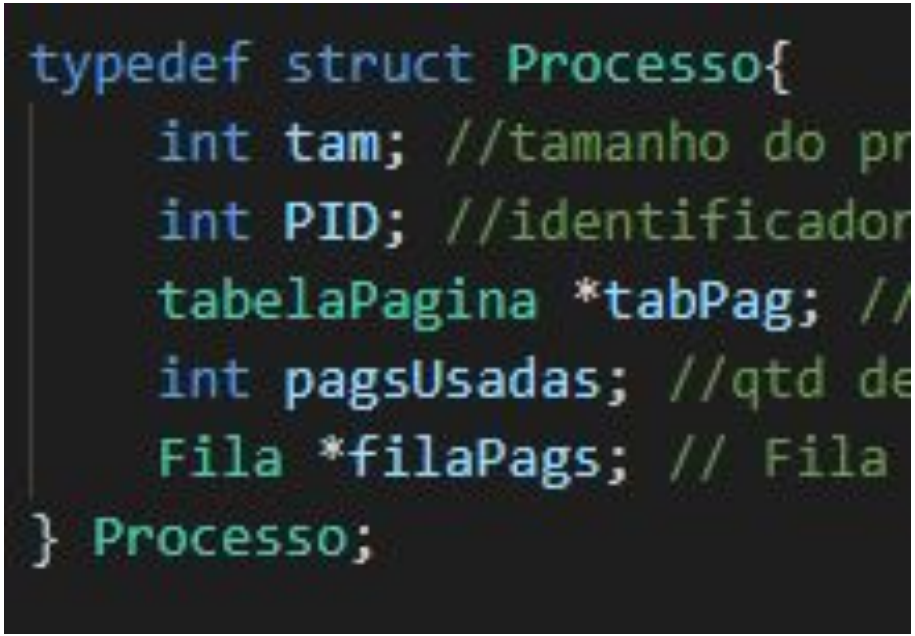
Main - Menu de ações possíveis

- É utilizado um Switch/Case para escolher entre os tipos ‘C’, ‘R’, ‘W’, ‘P’ e ‘I’.
  - Esse processo é repetido até o fim do arquivo.

```
switch (mode){
    case 'C':
        // Criar o processo lido antes desse do tamanho especificado logo em seguida em binário
        if(memLivre >= page_size && atoi(op)-page_size <= sec_size){
            if(qtdProc <= QTD_PROCESSOS){ //verifica se nao ultrapassou a qtd de processo permitida
                pro[pid-1] = criaProcesso(pid, memPrincipal, memVirtual, qtdPag, atoi(op), page_size, &memLivre, subs_alg);
                qtdProc++;
            }
            else
                printf("Nao e possivel criar um processo, pois a quantidade foi excedida.\n");
        }
        else
            printf("Nao e possivel criar um processo, pois nao ha espaco na memoria.\n");
        break;
```

Processo.h - Struct ‘Processo’

- Foi definida a struct Processo:
  - Possui as seguintes variáveis:
    - `int tam` => Tamanho do processo.
    - `int PID` => Identificador do processo.
    - `tabelaPagina *tabPag` => Tabela de páginas do processo.
    - `int pagsUsadas` => Quantidade de páginas utilizadas.
    - `Fila *filaPags` => Fila utilizada pelos algoritmos de substituição.



Processo - Criar Processos

- Função responsável pela criação de novos processos.
  - Utiliza a função iniciaTabela para criar uma tabela de páginas para o processo.
  - Utiliza a função criaFila para criar uma fila para o processo.

```
/** Funcao para criar um novo processo */
Processo* criaProcesso(int pid, Memoria *memPrincipal, Memoria *memVirtual, int qtdPag, int tamProcesso, int tamPag,
                                                                                int *memLivre, char alg){
    int pag, quadro;
    Processo *p = (Processo*) malloc(sizeof(Processo));
    if(p == NULL || tamProcesso <= 0){
        printf("Erro ao criar processo\n");
        return NULL;
    }
    p->tam = tamProcesso;
    p->PID = pid;
    p->pagsUsadas = 0; //processo ainda nao utiliza nenhuma pagina
    p->tabPag = iniciaTabela(tamProcesso/tamPag); //cria tabela de paginas do processo
    p->filaPags = criaFila(); //cria fila para LRU e FIFO

    pag = 0;
```

Pagina.h - Structs da página

Definimos uma variável do tipo enum (bit_pres_aus) que é um conjunto de valores inteiros representados por identificadores, representando o bit presente e ausente controlando quais páginas estão fisicamente presentes na memória, nesse caso temos que ausente = 0, e presente = 1.

Criamos uma struct que representa o um conteúdo que pode conter na tabela (item_tabelaPagina), possuindo o bit presente e ausente (bpa) e o número do quadro em que se encontra (quadro).

E a struct que representa a tabela em si (tabelaPagina), possuindo a variável inteira que representa seu tamanho (tam) e um vetor de itens da página (paginas).

```c
/** bit presente/ausente */
typedef enum pres_aus{
    ausente, presente
} bit_pres_aus;

typedef struct item_tabelaPagina{
    bit_pres_aus bpa;
    int quadro;
}item_tabelaPagina;

/** Struct que representa uma tabela
typedef struct tabelaPagina{
    int tam; //tamanho da tabela
    item_tabelaPagina *paginas;
}tabelaPagina;
```

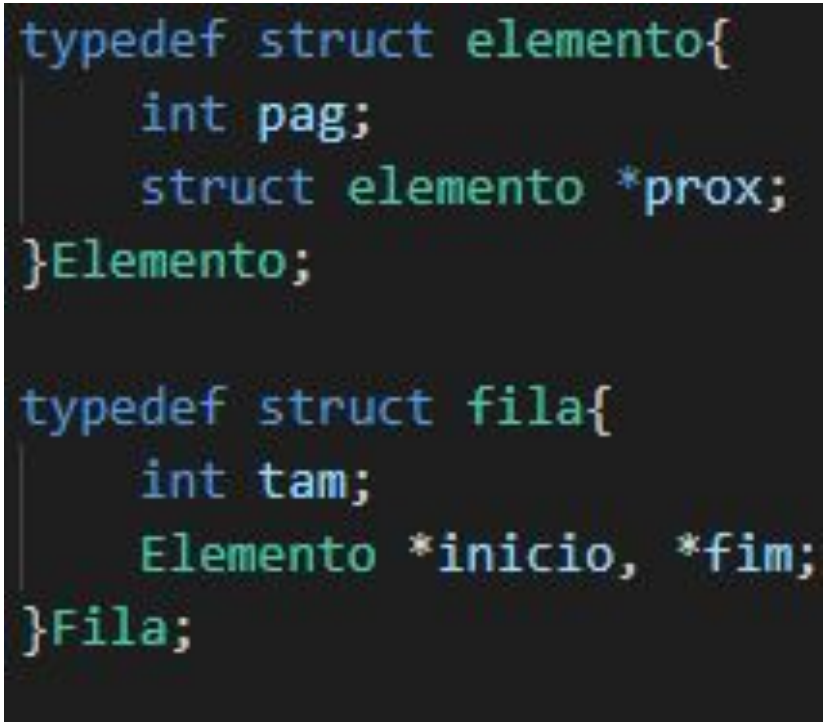
Pagina - Iniciar Tabela

```c
/** Cria uma nova tabela de paginas */
tabelaPagina* iniciaTabela(int tam){
    tabelaPagina *tabPag = (tabelaPagina*) malloc(sizeof(tabelaPagina));
    tabPag->tam = tam;
    tabPag->paginas = (item_tabelaPagina*) calloc(tam, sizeof(item_tabelaPagina));
    for(int i = 0; i < tam; i++){
        tabPag->paginas[i].quadro = -1;
    }
    return tabPag;
}
```

A função iniciaTabela tem como objetivo retornar a tabela de páginas do processo, para isso é necessário passar como parâmetro o tamanho da tabela (divisão do tamanho do processo pelo tamanho da página), alocamos então espaço de memória que será utilizado (tabPag) , atualizaremos o tamanho da tabela, alocamos o espaço para o vetor de páginas (conteúdo da tabela) colocando o valor do quadro como -1, pois aqui ainda não foi definido seu quadro e retornamos a tabela

Fila.h - Structs da Fila

- Foi definida duas estruturas na Fila:
  - Elemento possui as variáveis:
    - `int pag` => Valor da pagina.
    - `struct elemento *prox` => Referência o próximo elemento
  - Fila possui as variáveis:
    - `int tam` => Tamanho da fila.
    - `Elemento *inicio` => Elemento que marca o início da fila.
    - `Elemento *fim` => Elemento que marca o fim da fila.



Fila - Cria Fila

- Função responsável pela criação de uma fila.
  - Alocar espaços na fila.
  - Define o tamanho da fila igual a zero.
  - A fila portanto não possui elementos

```c
/** Cria uma nova fila */
Fila *criaFila(){
    Fila *fila = (Fila *)malloc(sizeof(Fila));
    fila->tam = 0; //nao tem nada ainda
    fila->inicio = NULL; //vazia
    fila->fim = NULL; //vazia
    return fila;
}
```

Processo - Criação de Processos

- Como o processo deve ser dividido em páginas, deve-se determinar quais irão para a memória principal e quais irão para virtual:
  - A primeira página irá para a memória principal.
  - As demais irão para a memória virtual.
- Para isso, deve-se inserir quadros nas memórias, com a função insereQuadro.
- Deve-se inserir também essas páginas na tabela de páginas do processo (com a função inserePagina), indicando se está presente ou ausente na memória principal.
- Para a página enviada para a memória principal, deve-se armazenar o seu índice em uma fila, com a função push.

```
while(tamProcesso > 0){
    if(pag == 0){ //coloca primeira pagina na memoria princi
        quadro = insereQuadro(memPrincipal, p->PID, pag); //
        inserePagina(p->tabPag, presente, pag, quadro); //in
        push(p->filaPags, pag); //insere paginas na fila

        *memLivre -= tamPag; //diminui quantidade de memoria
    }
    else{ //coloca demais paginas na memoria virtual
        quadro = insereQuadro(memVirtual, p->PID, pag); //in
        inserePagina(p->tabPag, ausente, pag, quadro+1); //i
    }
    pag++;
    tamProcesso -= tamPag;
}
p->pagsUsadas += pag;
printf("Processo %d criado!\n", pid);
printMemoria(memPrincipal, memVirtual, tamPag);
printProcesso(p, alg);
return p;
```

Memória - Inserir Quadro

A função insereQuadro tem como objetivo retornar o quadro inserido na memória, para isso é necessário passar como parâmetro a memória, o identificador de processos e o numero da pagina, feito isso, verificamos os quadros livres com a função quadroLivre, caso ele seja, possui o valor 1, e então atualizamos os seus valores.

```c
/** Insere um quadro na memoria */
int insereQuadro(Memoria *mem, int pid, int numPag){
    int quadro = quadroLivre(mem);
    mem->quadros[quadro].PID = pid;
    mem->quadros[quadro].numPag = numPag;
    mem->quadros[quadro].elemento = 0;
    mem->quadrosLivres[quadro] = 0;
    mem->qtd_quadrosLivres--; //decrementa o numero to

    return quadro;
}
```

```c
/** retorna o quadro livre */
int quadroLivre(Memoria *mem){
    int i;
    for(i = 0; i < mem->tam; i++){
        if(mem->quadrosLivres[i] == 1)
            return i;
    }
    return -1;
}
```

Memória - Remove Quadro

A função removeQuadro tem como objetivo remover um quadro da memória, para isso é necessário passar como parâmetro a memória e o quadro, para assim atualizarmos seus valores, como removido.

```
/** Remove um quadro da memoria */
void removeQuadro(Memoria *mem, int quadro){
    //retorna o pid e o numPag para o estado i
    mem->quadros[quadro].PID = 0;
    mem->quadros[quadro].numPag = -1;
    mem->quadrosLivres[quadro] = 1; //o quadro
    mem->qtd_quadrosLivres++; //incrementa o n
}
```

Página - Inserir Página

```c
/** Insere elemento na tabela de paginas */
void inserePagina(tabelaPagina *tabPag, bit_pres_aus bpa, int pagina, int quadro){
    tabPag->paginas[pagina].bpa = bpa;
    tabPag->paginas[pagina].quadro = quadro;
}
```

A função inserePagina tem como objetivo inserir uma página na tabela de páginas, para isso é necessário passar como parâmetro o vetor da tabela de página, o bit presente/ausente para ter o controle se está presente na memória ou não, e o quadro, então atualizaremos os valores da página.

Página - Remover Página

A função removerPagina tem como objetivo remover a página da tabela de páginas, é necessário passar como parâmetro a tabela de páginas e a página, para então removermos o conteúdo da página (atualização dos valores).

```
/** Remove elemento na tabela de paginas */
void removePagina(tabelaPagina *tabPag, int pagina){
    tabPag->paginas[pagina].bpa = 0;
    tabPag->paginas[pagina].quadro = -1;
}
```

Fila - Inserção de página

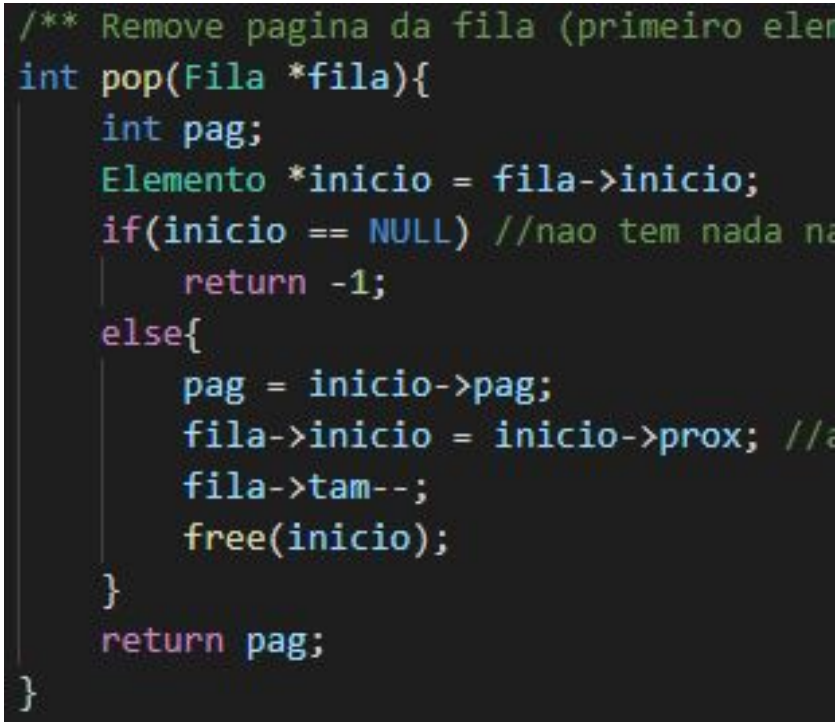
- A função push é aquela que ficará responsável pela inserção da página ao final da fila:
  - É alocado espaço para a inserção da nova página.
  - Inserimos o valor da página atual no elemento atual da fila.
  - Igualamos o elemento seguinte a NULL para indicar o novo fim da fila e atualizamos o tamanho.
  - Se a fila não possuir elementos o elemento atual será o início e fim da fila.
  - Todos os demais elementos inseridos serão sucessores do último elemento da fila e portanto o novo final da fila.

```
void push(Fila *fila, int pag){
    Elemento *elem = (Elemento *)malloc(sizeof(Elemento));
    elem->pag = pag;
    elem->prox = NULL; //indica que é o fim
    fila->tam++;

    if(fila->inicio == NULL){
        fila->inicio = elem;
        fila->fim = fila->inicio;
    }
    else{
        fila->fim->prox = elem; //novo elemento será o prox
        fila->fim = elem; //novo elemento é o novo ultimo
    }
}
```

Fila - Remoção de página

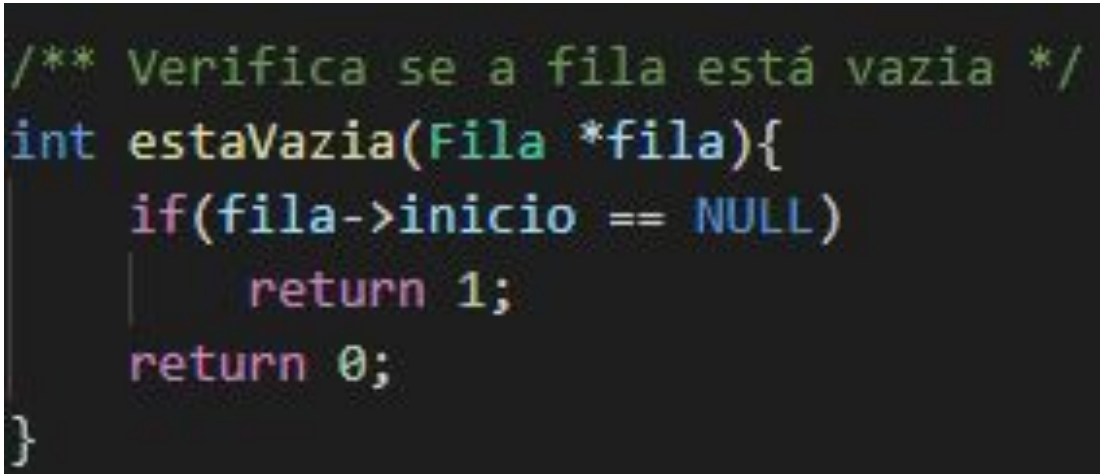
- A função pop é aquela que ficará responsável pela remoção a primeira página da fila:
  - Primeiramente verificamos se a fila possui pelo menos um elemento, comparando se o início é igual a NULL. Pois não terão elementos a serem removidos caso a fila estiver vazia.
  - Se a fila tiver elementos colocamos o valor da pagina inicial que será removida em uma variável auxiliar.
  - Colocamos o elemento sucessor como o novo início da fila e atualizamos o tamanho da fila.
  - Após a remoção liberamos este espaço e retornamos o valor da página removida, para informar a remoção.



```
/** Remove pagina da fila (primeiro elem
int pop(Fila *fila){
    int pag;
    Elemento *inicio = fila->inicio;
    if(inicio == NULL) //nao tem nada na
        return -1;
    else{
        pag = inicio->pag;
        fila->inicio = inicio->prox; //a
        fila->tam--;
        free(inicio);
    }
    return pag;
}
```

Fila - Verificação na Fila

- A função estaVazia apenas verifica se existem elementos na fila, para isso comparamos se o início é igual a NULL:
  - Se for igual a função retorna 1 para indicar que a fila não possui elementos e portanto está vazia.
  - Se for diferente a função retorna 0 para indicar que a fila possui elementos e não está vazia.



Processo - Criação de processos

- Depois de atualizar a quantidade de páginas utilizadas, é exibida uma mensagem que afirma que a criação do processo foi bem-sucedida.
- Com a função printMemoria, é possível ver a situação das memórias.
- Com a função printProcesso, é possível ver o estado e as atividades do processo.

```
while(tamProcesso > 0){
    if(pag == 0){ //coloca primeira pagina na memoria princi
        quadro = insereQuadro(memPrincipal, p->PID, pag); //
        inserePagina(p->tabPag, presente, pag, quadro); //in
        push(p->filaPags, pag); //insere paginas na fila

        *memLivre -= tamPag; //diminui quantidade de memoria
    }
    else{ //coloca demais paginas na memoria virtual
        quadro = insereQuadro(memVirtual, p->PID, pag); //in
        inserePagina(p->tabPag, ausente, pag, quadro+1); //i
    }
    pag++;
    tamProcesso -= tamPag;
}
p->pagsUsadas += pag;
printf("Processo %d criado!\n", pid);
printMemoria(memPrincipal, memVirtual, tamPag);
printProcesso(p, alg);
return p;
```

Memória - Print Memória

```c
/** Printa a memoria na tela */
void printMemoria(Memoria *memPrincipal, Memoria *memVirtual, int tamPag){
    int i;
    printf("-> Memoria Principal:\n");
    printf("PID Pagina Quadro Elemento\n");
    for(i = 0; i < memPrincipal->tam/tamPag; i++){
        if(memPrincipal->quadros[i].PID != 0) //se for diferente de 0, espaco da memoria esta ocupado
            printf(" %d    %d      %d       %d\n", memPrincipal->quadros[i].PID, memPrincipal->quadros[i].numPag,
                                                    i, memPrincipal->quadros[i].elemento);
    }
```

A função printMemoria, tem como objetivo exibir de forma “amigável” o que está acontecendo da simulação em relação as memórias, para isso é necessário passar como parâmetro as duas memórias e o tamanho da página.

Printamos inicialmente a *memória principal*, indicando seus valores (pid, número da página, número do quadro e seu elemento).

Memória - Print Memória

```
    printf("-> Memoria Virtual:\n");
    printf("PID Pagina Quadro Elemento\n");
    for(i = 0; i < memVirtual->tam/tamPag; i++){
        if(memVirtual->quadros[i].PID != 0) //se for diferente de 0, espaco da memoria esta ocupado
            printf(" %d    %d      %d       %d\n", memVirtual->quadros[i].PID, memVirtual->quadros[i].numPag,
                                                    i, memVirtual->quadros[i].elemento);
    }
}
```

E depois a *memória virtual*, indicando também os valores de seus elementos.

Processo - Exibição de Informações

- Função responsável por exibir informações do processo.
  - Exibe o identificador de processo.
  - Exibe a tabela de páginas, com a função printTabela.
  - Exibe os elementos presentes na fila, com a função printFila.

```c
/** Printa processo na tela */
void printProcesso(Processo *p, char alg){
    printf("-> Processo %d:\n", p->PID);
    printf("--> Tabela de paginas: ");
    printTabela(p->tabPag);
    printFila(p->filaPags);
}
```

Página - Print Tabela

A função printTabela tem como objetivo, auxiliar a exibição amigável na tela, para isso é necessário passar como parâmetro a tabela de páginas e caso a página esteja sendo utilizada é exibido então o valor de seu quadro.

```c
/** Printa tabela de paginas */
void printTabela(tabelaPagina *tabPag){
    int i = 0;
    while(i < tabPag->tam){
        if(tabPag->paginas[i].bpa == presente)
            printf("%d ",tabPag->paginas[i].quadro);
        i++;
    }
    printf("\n");
}
```

Fila - Print da Fila

- Função responsável por exibir as informações da Fila.
  - O elemento auxiliar receberá o início da fila.
  - A condição realiza o print elemento por elemento até chegar ao final da fila.
  - Além de printar a seta “-> “ entre os elementos.

```c
/** Printa todos os elementos da fila */
void printFila(Fila *fila){
    Elemento *elem = fila->inicio;
    printf("Fila: ");
    while (elem != NULL){
        printf("%d ", elem->pag);
        elem = elem->prox;
        if(elem != NULL)
            printf("-> ");
    }
    printf("\n");
}
```

Main - Leitura

- Executa o comando responsável por ler um endereço de memória e retornar o elemento presente.
  - Caso o endereço fornecido seja maior do que o permitido, será exibida uma mensagem de erro.
    - Com endereços de n bits, temos, no máximo, até 2^n-1 como endereços permitidos.

```
case 'R':
    // Lê o endereço de memoria especificado logo após
    if(maxEnd > getDec(mode, op))
        lerEndereco(pro[pid-1], memPrincipal, memVirtual, getDec(mode, op), page_size, &memLivre, subs_alg);
    else
        printf("O endereco logico tem mais bits que o permitido.\n");
    break;
```

Processo - Ler em um endereço

- Função lerEndereco procura um endereço especificado e retorna o retorna o elemento presente na página representada por esse endereço.
  - Essa página é representada pela divisão entre o endereço especificado e o tamanho das páginas.
  - Caso não tenha sido criado um processo, será exibida uma mensagem de erro e a função será finalizada.

```
/** Lê o endereco logico e o elemento que esta no quadro associado a ele */
void lerEndereco(Processo *p, Memoria *memPrincipal, Memoria *memVirtual, int endereco,
                                                    int tamPag, int *memLivre, char alg){
    int pag = endereco / tamPag;

    if(p == NULL){
        printf("Processo nao existe.\n");
        return;
    }
```

Processo - Ler em um endereço

- Caso o endereço especificado seja maior que o próprio processo, será exibida outra mensagem de erro, além de exibir a memória e o processo com as funções printMemoria e printProcesso, respectivamente.
- Caso a página desejada esteja presente na memória principal, será exibida uma mensagem com suas informações e elemento.
    - Caso o algoritmo escolhido seja o LRU, deve-se mover a página recém-utilizada para o final da fila.

```
if(endereco > p->tam){
    printf("Processo tentou ler em pagina que estava fora da memoria.\n");
    printMemoria(memPrincipal, memVirtual, tamPag);
    printProcesso(p, alg);
    return;
}

if(p->tabPag->paginas[pag].bpa == presente){
    printf("Processo %d acessou pagina %d no quadro %d e leu %d.\n", p->PID, pag,
        p->tabPag->paginas[pag].quadro, memPrincipal->quadros[p->tabPag->paginas[pag].quadro].elemento);
    if(alg == 'L')
        moveFim(p->filaPags, pag); //move pagina para o final da fila
}
```

Fila - Página recentemente acessada

- Função moveFim é responsável por mover a página acessada para o fim da fila:
  - O elemento auxiliar “elem” receberá o início da fila. Enquanto um outro elemento “ant” receberá o elemento lido anteriormente.
  - Iniciamos um condicional que irá correr até o elemento a ser movido for encontrado.
  - E comparar se o elemento atual analisado é o que deve ser movido.
  - Enquanto não achar, o anterior recebe o atual e o atual recebe o próximo.
  - Após encontrar a página verificamos se a página está no fim da fila ou não está nela, em ambos os casos encerramos a função.

```
/** Move elemento para o final da fila */
void moveFim(Fila *fila, int pag){
    Elemento *elem = fila->inicio;
    Elemento *ant;

    //loop para procurar o elemento a ser
    while (elem != NULL){
        if(elem->pag == pag)
            break; //indica que achou o e
        ant = elem;
        elem = elem->prox;
    }
    if(elem == NULL)
        return;
    if(elem == fila->fim) //indica que já
        return;
```

Fila - Página recentemente acessada

- Função moveFim é responsável por mover a página acessada para o fim da fila:
  - Comparamos se o elemento atual corresponde ao início da fila, deslocamos o início para o sucessor, removemos o sucessor do elemento atual, colocamos ele como próximo do atual elemento no fim da fila e atualizamos o elemento do final da fila.
  - Caso a página esteja em qualquer posição no meio da fila, colocamos o próximo do anterior como o próximo do elemento atual, removemos o próximo do elemento atual e colocamos ele ao final da fila.

```
    if(elem == fila->inicio){
        fila->inicio = elem->prox;
        elem->prox = NULL; //o ultimo nao
        fila->fim->prox = elem; //indica
        fila->fim = elem; //atualiza o fi
    }
    else{
        ant->prox = elem->prox; //o anter
        elem->prox = NULL; //o ultimo nao
        fila->fim->prox = elem; //indica
        fila->fim = elem; //atualiza o fi
    }
}
```

Processo - Ler em um endereço

- Caso a página desejada não esteja presente na memória principal e haja memória livre, deve-se:
  - Inserir um quadro na memória principal com insereQuadro.
  - Remover a página da tabela de páginas e inserir uma nova, mas, desta vez, com o bit presente.
  - Inserir página na fila do processo.
  - Caso o algoritmo LRU tenha sido escolhido, utiliza a função moveFim.
  - Após isso, deve-se encontrar o quadro relacionado à página (encontraQuadro) e removê-lo da memória virtual (removeQuadro).

```
else{
    if(*memLivre > 0){
        int quadro = insereQuadro(memPrincipal, p->PID, pag); //insere um novo quadro na memoria fisica
        removePagina(p->tabPag, pag); //remove pagina da tabela
        inserePagina(p->tabPag, presente, pag, quadro); //insere um novo elemento na tabela de paginas e o re
        push(p->filaPags, pag); //insere nova pagina na fila
        printf("Processo %d acessou pagina %d no quadro %d e leu %d.\n", p->PID, pag,
            p->tabPag->paginas[pag].quadro, memPrincipal->quadros[p->tabPag->paginas[pag].quadro].elemento);
        if(alg == 'L')
            moveFim(p->filaPags, pag); //move pagina para o final da fila
        int pagina = encontraQuadro(memVirtual, p->PID, tamPag, pag);
        removeQuadro(memVirtual, pagina); //remove da memoria virtual
        *memLivre -= tamPag;
    }
```

Processo - Ler em um endereço

- Caso não haja memória livre, deve-se utilizar um algoritmo de substituição para realizar a troca de páginas.
  - A função que representa esses algoritmos é a trocaPaginaLRU_FIFO.
  - Após a troca, exibe mensagem com informações do processo e o elemento presente na página.
  - Depois de tudo, exibe a memória e o processo com as funções printMemoria e printProcesso, respectivamente.

```
        else{ //aplica o algoritmo de substituicao
            trocaPaginaLRU_FIFO(p, memPrincipal, memVirtual, pag, memVirtual->quadros[pag].elemento);
            printf("Processo %d acessou pagina %d no quadro %d e leu %d.\n", p->PID, pag,
                p->tabPag->paginas[pag].quadro, memPrincipal->quadros[p->tabPag->paginas[pag].quadro].elemento);
        }
    }
    printMemoria(memPrincipal, memVirtual, tamPag);
    printProcesso(p, alg);
}
```

Processo - Substituição de páginas

- Função responsável pela aplicação dos algoritmos LRU e FIFO.
  - A principal diferença entre ambos os algoritmos é que, no LRU, a página recém-utilizada vai para o final da fila, enquanto no FIFO isso não ocorre.
  - Primeiramente, é preciso encontrar a nova página memória, que será armazenada na variável quadro.
  - Caso a fila esteja vazia, não é possível aplicar nenhum dos algoritmos.

```
/** Troca uma pagina antiga por uma nova com o LRU e o FIFO */
void trocaPaginaLRU_FIFO(Processo *p, Memoria *memPrincipal, Memoria *memVirtual, int pag, int var, char alg){
    int quadro, pagRemovida, novoQuadro, elemento;
    quadro = p->tabPag->paginas[pag].quadro; //encontra a nova pagina na memoria

    if(estaVazia(p->filaPags)){
        if(alg == 'L')
            printf("Nao foi possivel aplicar o LRU, pois a fila esta vazia.\n");
        else
            printf("Nao foi possivel aplicar o FIFO, pois a fila esta vazia.\n");
        return;
    }
```

Processo - Substituição de páginas

- Função responsável pela aplicação dos algoritmos LRU e FIFO.
  - O primeiro elemento da fila é retirado, e seu valor é retornado e armazenado.
  - Com esse valor, é determinado o novo quadro da memória principal.
  - E, com esse novo valor de quadro, seu elemento é armazenado.
  - Após isso, deve-se atualizar os registros da nova página na memória principal, além de inseri-la na tabela de páginas com o bit presente.

```
pagRemovida = pop(p->filaPags); //pega a pagina utilizada menos recentemente
novoQuadro = p->tabPag->paginas[pagRemovida].quadro; //encontra essa pagina na memoria principal
elemento = memPrincipal->quadros[novoQuadro].elemento;

//insere o registro da nova pagina na memoria principal
memPrincipal->quadros[novoQuadro].PID = p->PID;
memPrincipal->quadros[novoQuadro].numPag = pag;
memPrincipal->quadros[novoQuadro].elemento = var;
inserePagina(p->tabPag, presente, pag, novoQuadro);
```

Processo - Substituição de páginas

- Função responsável pela aplicação dos algoritmos LRU e FIFO.
  - Página antiga é inserida na memória virtual, além de inseri-la na tabela de páginas com o bit ausente.
  - Após isso, a página é inserida na fila do processo e uma mensagem de sucesso é exibida, para ambos os algoritmos.

```
    //insere pagina antiga na memoria virtual
    memVirtual->quadros[quadro-1].PID = p->PID;
    memVirtual->quadros[quadro-1].numPag = pagRemovida;
    memVirtual->quadros[quadro-1].elemento = elemento;
    inserePagina(p->tabPag, ausente, pagRemovida, quadro); //altera o regist
    push(p->filaPags, pag); //insere nova pagina no fim da fila

    if(alg == 'L')
        printf("LRU: Pagina %d foi trocada pela %d.\n", pagRemovida, pag);
    else
        printf("FIFO: Pagina %d foi trocada pela %d.\n", pagRemovida, pag);
}
```

Main - Escrita

- Executa o comando responsável por escrever em um endereço de memória.
  - Caso o endereço fornecido seja maior do que o permitido, será exibida uma mensagem de erro.
    - Com endereços de n bits, temos, no máximo, até 2^n-1 como endereços permitidos.

```
case 'W':
    // Escrita no endereço especificado logo após
    if(maxEnd > getDec(mode, op))
        escreverEndereco(pro[pid-1], memPrincipal, memVirtual, getDec(mode, op), page_size, &memLivre, subs_alg);
    else
        printf("O endereco logico tem mais bits que o permitido.\n");
    break;
```

Processo - Escrever em um endereço

- Função escreverEndereco procura um endereço especificado e escreve um valor aleatório na página representada por esse endereço.
  - Essa página é representada pela divisão entre o endereço especificado e o tamanho das páginas.
  - Caso não tenha sido criado um processo, será exibida uma mensagem de erro e a função será finalizada.

```
/** Escreve em um quadro que está associado a um endereco logico */
void escreverEndereco(Processo *p, Memoria *memPrincipal, Memoria *memVirtual, int endereco,
                                                    int tamPag, int *memLivre, char alg){
    int quadro, var;
    int pag = endereco / tamPag;

    if(p == NULL){
        printf("Processo nao existe.\n");
        return;
    }
```

Processo - Escrever em um endereço

- Caso o endereço especificado seja maior que o próprio processo, será exibida outra mensagem de erro, além de exibir a memória e o processo com as funções printMemoria e printProcesso, respectivamente.
- É gerado um valor pseudo-randômico entre 1 e 9, o qual representa o elemento a ser inserido na página indicada pelo endereço.

```
if(endereco > p->tam){
    printf("Processo tentou escrever em pagina que estava fora da memoria.\n");
    printMemoria(memPrincipal, memVirtual, tamPag);
    printProcesso(p, alg);
    return;
}

srand(time(0));
var = (rand() % 9) + 1; //numero aleatorio entre 1 e 9
```

Processo - Escrever em um endereço

- Caso a página desejada esteja presente na memória principal, será exibida uma mensagem com suas informações e elemento.
  - Caso o algoritmo escolhido seja o LRU, deve-se mover a página recém-utilizada para o final da fila.

```
if(p->tabPag->paginas[pag].bpa == presente){ //verifica se a pagina esta na memoria principal
    atualizaQuadro(memPrincipal, pag, var);
    printf("Processo %d acessou pagina %d no quadro %d e escreveu %d.\n", p->PID, pag,
                    p->tabPag->paginas[pag].quadro, memPrincipal->quadros[pag].elemento);
    if(alg == 'L')
        moveFim(p->filaPags, pag); //move pagina para o final da fila
}
```

Processo - Escrever em um endereço

- Caso a página desejada não esteja presente na memória principal e haja memória livre, deve-se:
  - Inserir um quadro na memória principal com insereQuadro.
  - Remover a página da tabela de páginas e inserir uma nova, mas, desta vez, com o bit presente.
  - Inserir página na fila do processo.
  - Para atualizar as informações do quadro recém-inserido na memória principal, deve-se utilizar a função atualizaQuadro.
  - Após isso, deve-se encontrar o quadro relacionado à página (encontraQuadro).

```
else{
    if(*memLivre > 0){ //se possui memoria livre, nao precisa usar um algoritmo de substituicao
        //int pagina = p->tabPag->paginas[pag].quadro;
        int quadro = insereQuadro(memPrincipal, p->PID, pag); //insere um novo quadro na memoria f
        removePagina(p->tabPag, pag); //remove pagina da tabela
        inserePagina(p->tabPag, presente, pag, quadro); //insere um novo elemento na tabela de pag
        push(p->filaPags, pag); //insere nova pagina na fila
        atualizaQuadro(memPrincipal, quadro, var);
        int pagina = encontraQuadro(memVirtual, p->PID, tamPag, pag);
```

Memória - Encontra Quadro

```c
/** Encontra um quadro da memoria */
int encontraQuadro(Memoria *mem, int pid, int tamPag, int pag){
    int quadro;
    for(quadro = 0; quadro < mem->tam/tamPag; quadro++){
        if(mem->quadros[quadro].PID == pid && mem->quadros[quadro].numPag == pag)
            return quadro; //encontrou
    }
    return -1; //nao encontrou
}
```

A função encontrarQuadro tem como objetivo retornar o valor do quadro caso encontre, e caso não encontre retorna -1. Para isso é necessário passar como parâmetro a memória, o identificador de processos, o tamanho da página e a página, e então percorremos a memória comparando os valores do identificador de processos e o número da página.

Memória - Atualiza Quadro

A função atualizaQuadro tem como objetivo atualizar o valor do elemento do quadro, é necessário passar como parâmetro a memória, a página e o elemento.

```
/** Função para atualizar o elemento do quadro */
void atualizaQuadro(Memoria *mem, int pag, int elem){
    mem->quadros[pag].elemento = elem;
}
```

Processo - Escrever em um endereço

- Deve-se remover a página (que foi inserida na memória principal) da memória virtual com a função removeQuadro.
- Depois, é exibida uma mensagem com informações do processo e o novo elemento presente na página.
- Caso o algoritmo LRU tenha sido escolhido, deve-se mover a página para o final da fila.

```
    if(pagina == -1)
        return;
    removeQuadro(memVirtual, pagina); //remove da memoria virtual
    printf("Processo %d acessou pagina %d no quadro %d e escreveu %d.\n", p->PID, pag,
                    p->tabPag->paginas[pag].quadro, memPrincipal->quadros[quadro].elemento);
    if(alg == 'L')
        moveFim(p->filaPags, pag); //move pagina para o final da fila
    *memLivre -= tamPag;
}
```

Processo - Escrever em um endereço

- Caso não haja memória livre, deve-se utilizar um algoritmo de substituição para realizar a troca de páginas.
  - A função que representa esses algoritmos é a trocaPaginaLRU_FIFO, onde também já será feita a escrita do novo elemento.
  - Após a troca, exibe mensagem com informações do processo e o elemento presente na página.
  - Depois de tudo, exibe a memória e o processo com as funções printMemoria e printProcesso, respectivamente.

```
        else{ //aplica algoritmo de substituicao
            trocaPaginaLRU_FIFO(p, memPrincipal, memVirtual, pag, var);
            //atualizaQuadro(memPrincipal, pag, var);
            printf("Processo %d acessou pagina %d no quadro %d e escreveu %d.\n", p->PID, pag,
                                                    p->tabPag->paginas[pag].quadro, var);
        }
    }
    printMemoria(memPrincipal, memVirtual, tamPag);
    printProcesso(p, alg);
}
```

Main - Instrução a ser executada pela CPU

```
case 'P':
    // Indicando instrução a ser executada pela CPU
    operacaoCPU(pro[pid-1], memPrincipal, memVirtual, getDec(mode, op), page_size, subs_alg);
    break;
```

- O comando P levará uma instrução a ser realizada pela CPU.
  - esse comando chamará a função operacaoCPU dos Processos.

Processo - Instrução a ser executada pela CPU

```c
void operacaoCPU(Processo *p, Memoria *memPrincipal, Memoria *memVirtual, int instrucao, int tamPag, char alg){
    if(p == NULL){
        printf("Processo nao existe.\n");
        return;
    }
    printf("Processo %d executou a instrucao %d, que foi executada pela CPU.\n", p->PID, instrucao);
    printMemoria(memPrincipal, memVirtual, tamPag);
    printProcesso(p, alg);
}
```

- A operação recebe os parâmetros do processo, dentre eles o Processo *p, se esse for igual a NULL não existe processo a ser executado, porém se existir, printamos sua execução feita pela CPU e seus efeitos na memória principal e virtual.

Main - Instrução de I/O a ser executada

```
case 'I':
    // Indicando instrução de I/O
    operacaoIO(pro[pid-1], memPrincipal, memVirtual, getDec(mode, op), page_size, subs_alg);
    break;
```

- O comando I levará uma instrução de Entrada e Saida.
  - esse comando chamará a função operacaoIO dos Processos.

Processo - Instrução de I/O a ser executada

```
void operacaoIO(Processo *p, Memoria *memPrincipal, Memoria *memVirtual, int instrucao, int tamPag, char alg){
    if(p == NULL){
        printf("Processo nao existe.\n");
        return;
    }
    printf("Processo %d executou a instrucao %d, que e de I/O\n", p->PID, instrucao);
    printMemoria(memPrincipal, memVirtual, tamPag);
    printProcesso(p, alg);
}
```

- A operação recebe os parâmetros do processo, dentre eles o Processo *p, se esse for igual a NULL não existe processo a ser executado, porém se existir, printamos sua execução e seus efeitos na memória principal e virtual, sendo essa uma instrução de I/O.

Main - Fim de uma Execução

- A cada comando lido, é necessário apertar Enter para ler um novo comando, para isso utilizamos a função pressEnter.
- Após término da leitura do arquivo fechamos o arquivo com fclose
- E liberamos os espaços alocados da memória principal e virtual com a função destroiMemoria.

```
    printf("Aperte enter para continuar...\n");
    pressEnter(stdin, &first); //limpa o buffer do
}
fclose(fp);

destroiMemoria(memVirtual);
destroiMemoria(memPrincipal);

return 0;
```

Memória - Liberar a memória da máquina

A função destroMemoria tem como objetivo apenas liberar os espaços que alocamos durante a simulação, dando os “free” nos ponteiros dos itens relacionados a memória.

```
/** Destroi a estrutura de memoria */
void destroiMemoria(Memoria *mem){
    free(mem->quadros);
    free(mem->quadrosLivres);
    free(mem);
}
```

Interação e resultados:

Após a implementação dos programas no diretório onde se encontram os mesmos, segue os seguintes passos para a execução:

- Criar os binários com “gcc -o nome_exec *.c -lm”
- Executá-los com “./nome_exec nome_arq.txt”

Executando o algoritmo temos um exemplo da resposta para cada comando.

Resultados - arquivoSimular.txt

```
gso03@tau01-vm3:~/trab3$ cat arquivoSimular.txt
##### início do arquivo ######
P1 C 500
P1 W  (1)2
P1 R (0)2
P1 P  (1)2
P1 W  (1024)2
P1 R (1024)2
P7 C 1000
P7 W  (800)2
P7 R (4095)2
P7 R  (800)2
P1 W  (231)2
P7 I  (2)2
P7 W  (4096)2
P7 W  (123)2
P7 R (123)2
P7 W  (800)2
P7 W  (231)2
P7 R (435)2
P7 W (718)2
P7 W (225)2
P1 R (312)
P7 W  (512)2
P7 R (801)2
##### final do arquivo ######
```

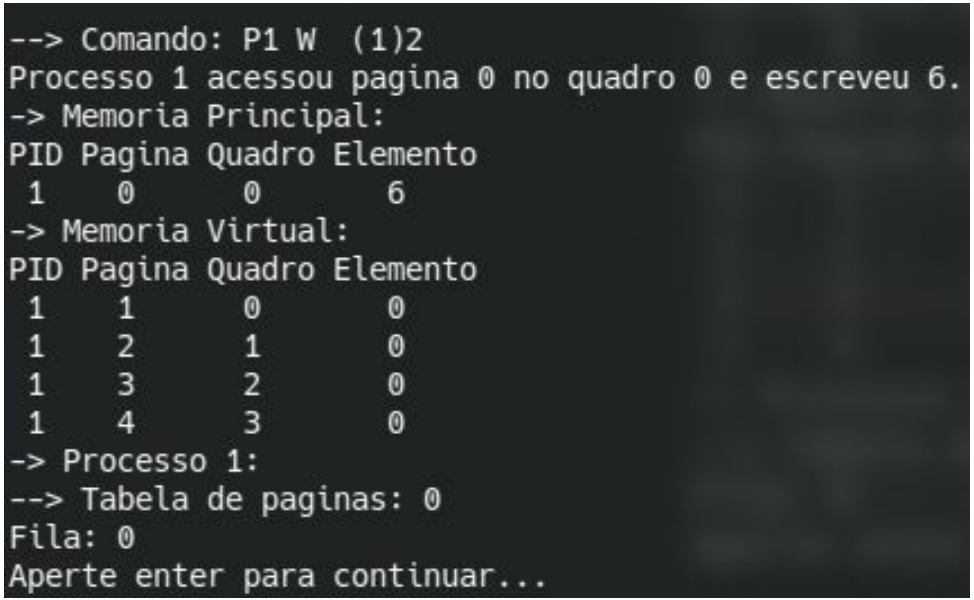
Resultados - Criação de processos

- Como o primeiro comando é o de criar o processo 1, temos que esse é o único conteúdo presente nas memórias no momento.

```
--> Comando: P1 C 500
Processo 1 criado!
-> Memoria Principal:
PID Pagina Quadro Elemento
 1    0      0       0
-> Memoria Virtual:
PID Pagina Quadro Elemento
 1    1      0       0
 1    2      1       0
 1    3      2       0
 1    4      3       0
-> Processo 1:
--> Tabela de paginas: 0
Fila: 0
Aperte enter para continuar...
```

Resultados - Escrita

- Como visto no arquivo arquivoSimular.txt, o segundo comando é de escrita.
- Como é gerado um valor aleatório, vemos que, desta vez, foi escrito o número 6.



Resultados - Leitura

- Como visto no arquivo arquivoSimular.txt, o comando ao lado é o terceiro a ser executado.
- E, comprovando o que a função de escrita realizou, temos que o elemento lido foi o 6.

```
--> Comando: P1 R (0)2
Processo 1 acessou pagina 0 no quadro 0 e leu 6.
-> Memoria Principal:
PID Pagina Quadro Elemento
 1    0      0       6
-> Memoria Virtual:
PID Pagina Quadro Elemento
 1    1      0       0
 1    2      1       0
 1    3      2       0
 1    4      3       0
-> Processo 1:
--> Tabela de paginas: 0
Fila: 0
Aperte enter para continuar...
```

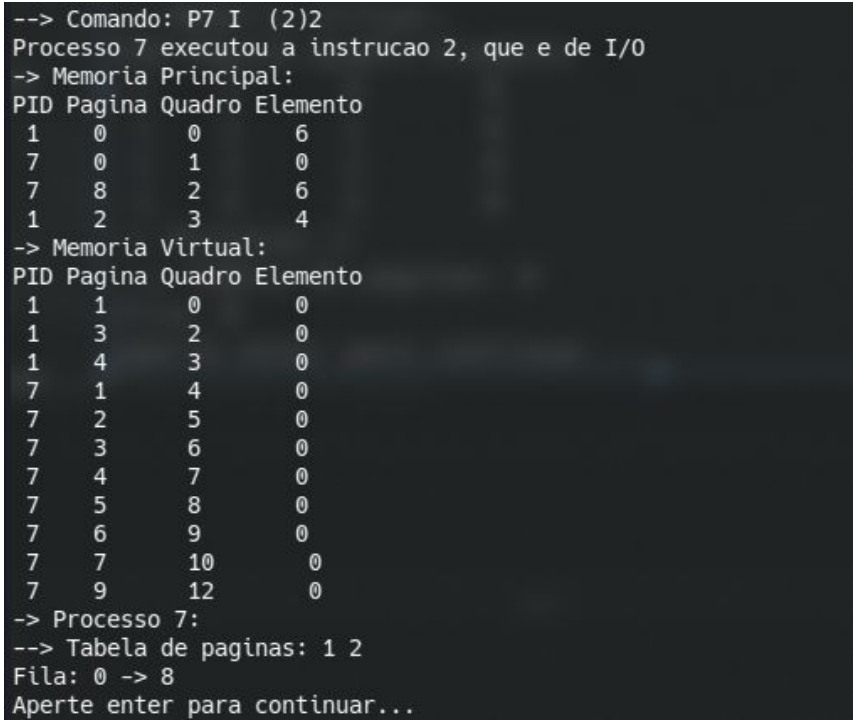
Resultados - Instrução executada pela CPU

- É possível ver que, ao utilizar uma instrução executada pela CPU, não há modificações nas memórias.

```
--> Comando: P1 P  (1)2
Processo 1 executou a instrucao 1, que foi executada pela CPU.
-> Memoria Principal:
PID Pagina Quadro Elemento
 1    0      0       6
-> Memoria Virtual:
PID Pagina Quadro Elemento
 1    1      0       0
 1    2      1       0
 1    3      2       0
 1    4      3       0
-> Processo 1:
--> Tabela de paginas: 0
Fila: 0
Aperte enter para continuar...
```

Resultados - Instrução de I/O

- É possível ver que, ao utilizar uma instrução de I/O, não há modificações nas memórias.



Conclusão

Após realizar esta simulação, concluímos que o gerenciamento de memória é essencial em um sistema operacional e a paginação é uma excelente ferramenta na organização dos processos. Por meio da simulação, apresentamos via terminal todas as informações relevantes sobre o gerenciamento, conseguimos realizar o gerenciamento de memória utilizando os algoritmos de substituição, para um número indefinido de processos adicionados, paralelo a isso concluímos que a execução de processos dependem diretamente da organização da memória que pode ser expandida com o uso da memória virtual.

Bibliografia

- **http://campeche.inf.furb.br/tccs/2004-II/2004-2gustavomoritzvf.pdf**
- https://github.com/joelrlneto/memoriavirtual/blob/master/SMV.c
- Tanenbaum, A. S.; Sistemas Operacionais Modernos. Editora Pearson Brasil, 2a. edição, 2003.