Pseudo-SO

Daniella Angelos, Guilherme Branco, Thales Ramos

[1]Departamento de Ciência da Computação - Universidade de Brasília

1. Objetivo

Implementação de um pseudo-SO multiprogramado, composto por um Gerenciador de Processos, um Gerenciador de Memória e por um Gerenciador de Recursos. O gerenciador de processos deve ser capaz de agrupar os processos em quatro níveis de prioridades e executá-los conforme estas prioridades. O gerenciador de memória deve garantir que um processo não acesse as regiões de memória de um outro processo, além de verificar se há espaço suficiente para o processo ser executado. E o gerenciador de recursos deve ser responsável por administrar a alocação e a liberação de todos os recursos disponíveis, garantindo uso exclusivo dos mesmos.

2. Implementação

Foi utilizado a linguagem C++ no padrão *C++11*, assim como a biblioteca *pthreads*, esta para o uso dos semáforos. Foi utilizado uma solução serial por ser mais rápido de se desenvolver, porém a partir de vários métodos existentes é póssivel criar uma solução concorrente sem muita complicação, como criar um novo método de execução onde cada thread o roda e possuem informações de um processo cada, adicionando também váriaveis de condição para bloquear as threads nos momentos necessários(esperando execução de recurso ou esperando sua vez de executar).

2.1. Processo

```
1  class processo_t
2  {
3    public:
4      processo_t(){}
5      bool in_mem();
6      void liberar_recursos();
7      void check();
8    private:
9      int pid;
10     int time_init;
11     int prioridade;
12     int time_proc;
13     int mem_offset;
14     int qtd_blocos;
15     bool uso_impressora;
16     bool uso_scanner;
17     bool uso_modem;
18     bool uso_disco;
19     int recurso_bloqueado;
20     int has_recurso();
21     int use_recurso();
22 };
```

No nosso programa, processos são instâncias de um objeto e abaixo estão as váriaveis que relacionam atributos internos do processo como especificados e algumas outras que se fizeram necessárias ao longo do desenvolvimento:

- pid : identificação do processo
- time_init : tempo que o processo chegou
- prioridade : prioridade do processo
- time_proc : tempo que será necessário para terminar o processo
- mem_offset : *offset* do processo na memória
- qtd_blocos : quantidade de memória gasta pelo processo
- uso_impressora : se utiliza impressora
- uso_scanner : se utilza scanner
- uso_modem : se utiliza modem
- uso_disco : se utiliza disco
- recurso_boqueado : indica qual recurso esta sendo utilizado pelo processo

Assim como alguns métodos para interagir com estes atribuitos (os *gets* e *sets* não serão mencionados por serem auto-explicativos(este também foram omitidos da imagem):

- **bool im_mem()**: Checa se o processo esta em memória.
- **void liberar_recursos()**: Libera qualquer recurso que o processo detenha.
- **int recurso_bloqueado()**: Retorna qual recurso esta bloqueado, zero se não houver.
- **int has_recurso()**: Checa se utiliza algum tipo de recurso.
- **int use_recurso()**: Se utiliza algum recurso retorna qual recurso será utilizado.
- **void check()**: Libera os recursos e tenta atribuir um recurso para o processo.

2.2. Gerenciador de Processos

```
1  class escalonador
2  {
3    private:
4      vector<processo_t> processos;
5      queue<processo_t> f_usuario_p1;
6      queue<processo_t> f_usuario_p2;
7      queue<processo_t> f_usuario_p3;
8      queue<processo_t> f_temporeal;
9      memoria m;
10     void vai_ffila(processo_t _p);
11     void popula();
12     void executa_processo(processo_t& _p);
13     bool prox_processo(processo_t *p);
14     bool ainda_existe_processo();
15     void order_process();
16     void start_time();
17     void show_allp();
18     void despachante (processo_t& p);
19     void print_exec(processo_t p);
20   public:
21     int seconds_passed;
22     void utils_tomem(string nome_arq);
23     void simulacao();
24     int get_time_passed();
25 };
```

As primeiras cinco variáveis privadas são as filas, a primeira um vetor com todos os processos e as seguintes filas FIFO que irão guardar os processos em execução por prioridade. A sexta variável, ”memoria m”, é uma instância do gerenciador de memória que será explicado logo abaixo. Em seguida, estão os métodos:

- **void vai_ffila(processo_t _p)**: Insere o processo recebido no fim da fila correta baseando-se na prioridade do processo.
- **void popula()**: Chama vai_ffila a medida que o tempo passa e os processos chegam para ser executados.
- **void executa_processo(processo_t &_p)**: Coloca um processo em execução.
- **void show_allp()**: Método usado em debugs para mostrar todos os processos em cada fila.
- **void despachante(processo_t &p)**: Chamada para mostrar informações do processo e executa-lo, assim como mostrar informações pós a execução.
- **void print_exec(processo_t p)**: Imprime alguns atributos de um processo em execução.
- **void utils_tomem(string nome_arq)**: Coloca as informações dos processos presente no arquivo de entrada.
- **void order_process()**: Organiza os processos na fila de todos os processos por tempo de chegada.
- **void start_time()**: Inicializa o contador de ”tempo”interno do escalonador.
- **void simulacao()**: Loop principal que controla a execucao dos processos.
- **int get_time_passed()**: Retorna o contador de ”tempo”.
- **bool prox_processo(processo_t *p)**: Retorna *true* se existe um próximo processo a ser executado e o coloca no ponteiro, *false* caso contrário.
- **bool ainda_existe_processo()**: Retorna *true* se ainda existe algum processo em alguma fila para ser executado, *false* caso contrário.

2.3. Gerenciador de Memória

```
1  class memoria
2  {
3    private:
4      bitset<MAX_MEM> mem;
5    public:
6      memoria();
7      void show();
8      unsigned int aloca(unsigned int qtd,int tipo_p);
9      void desaloca(unsigned int offset, unsigned int qtd);
10     unsigned int verifica(unsigned int qtd,unsigned int start, unsigned
           int end);
11 };
```

A única variável desta classe é a memória, foi usado um bitset pois foi feito somente um mapa de bits para o gerenciamento de memória. Após isto estão os métodos que são:

- **unsigned int aloca(unsigned int qtd, int tipo_p)**: Retorna o *offset* quando possível a alocação de uma quantidade(qtd) para um processo usuário ou real(tipo_p).
- **void show()**: Mostra o estado do mapa de bits.

- **void desaloca(unsigned int offset, unsigned int qtd)**: Desaloca a memória a partir de um `offset` e uma quantidade (qtd).
- **unsigned int verifica(unsigned int qtd, unsigned int start, unsigned int end)**: Verifica se existe uma quantidade(qtd) contígua de memória livre a partir de um começo (`start`) até um certo limite (`end`). O começo e o fim servem para delimitar a memória de processos tipo Real e processos tipo Usuário.

2.4. Gerenciador de Recursos

```
1  enum Recursos{ SEM_RECURSO = 0, IMPRESSORA, SCANNER, DISCO, MODEM,
       TEM_RECURSO = 1};
2
3  extern sem_t scanner;
4  extern sem_t impressora;
5  extern sem_t modem;
6  extern sem_t disco;
7
8  void inicializarSemaforos();
9  bool bloquear_recurso(int recurso);
10 void liberar_recurso(int recurso);
```

Usamos um enumerador para marcar os tipos de recursos e ter ou não recursos. Após isto vem quatro semáforos que são utilizados para atribuir um recurso a um processo e então os métodos:

- **void inicializarSemaforos()**: Inicializa os semáforos, scanner e modem com um, e impressora e disco com dois.
- **bool bloquear_recurso(int recurso)**: Tenta bloquear um recurso mediante `try_wait(&sem_t)` caso positivo retorna `true`, caso negativo retorna `false`.
- **void liberar_recurso(int recurso)**: Libera um recurso a partir do valor que o representa na enumeração mediante `sem_post(&sem_t)`.

3. Análise teórica

Para execução dos **processos** levando em consideração suas prioridades, usamos, como especificado, uma abordagem não-preemptiva aos processos de tempo real, enquanto que para os processos de usuários, o algoritmo usado foi o Escalonamento de Múltiplas Filas preemptivo com o *quantum* de 1 segundo. Para evitar *starvation*, a cada vez que um processo é executado, sua prioridade diminui, ou seja, o mesmo vai para o final da fila da prioridade inferior (a não ser que já tenha a prioridade mais baixa).

A **memória** foi organizada, como já foi dito, como um mapa de bits, isto é, verificar se um processo cabe em memória se resume em varrer o mapa de bits e procurar espaços livres com o tamanho do processo. Neste caso, a alocação é contínua, sem se fazer uso de páginas, ou seja, se o processo ocupar `n` bits, e tiver `n` bits livres, porém, espalhados em memória, este processo não poderá ser alocado. O algoritmo de busca de espaços livres utilizado foi o *First Fit*, então os primeiros `n` bits livres encontrados, serão os utilizados para alocar o processo.

Os processos que utilizarão algum **recurso**, tentarão bloquear o mesmo em um tempo aleatório, simulando a falta de previsibilidade que temos dos processos.

4. Dificuldades e soluções

Uma das primeiras dificuldades que enfrentamos, foi popular a fila de processos por prioridade obedecendo a ordem de chegada. Para resolver este problema, criamos uma função `popula()` na gerência de processos, que verifica se existe algum processo que deve ir à fila, pois o tempo que já passou é igual ao tempo de chegada do processo antes de escolher o próximo a ser escalonado.

Quando não há memória para um processo, debatemos sobre o que fazer em seguida, e decidimos que o mesmo deveria voltar à fila de processos, para que, futuramente, pudesse tentar executar novamente.

Para a gerência de memória, pensamos em usar uma lista de espaços vazios, ao invés do mapa de bits, mas por maior facilidade de implementação do segundo e por sabermos que a memória não é grande para a nossa simulação, ou seja, a busca no vetor não seria tão custosa, consideramos mais vantajoso usar o mapa de bits.

5. Distribuição de tarefas

Abaixo a divisão de tarefas para cada componente do grupo:

- **Guilherme B.**: Estrutura do código, criação das classes, parte da gerência dos processos, escrita do relatório.
- **Daniella A.**: Parte da gerência dos processos, gerência de recursos, saída do programa, escrita do relatório.
- **Thales R.**: Leitura do arquivo de entrada, gerência de memória, escrita do relatório.

6. Considerações finais

O projeto foi relevante para compreendermos melhor a dificuldade de se construir um sistema operacional que leva em consideração todos os aspectos fundamentais de *hardware* e *software* para garantir justiça aos processos e transparência ao usuário.

7. Referências Bibliográficas

[1] Slides da disciplina Sistemas Operacionais, ministradas pela professora Aletéia Araújo, em 1º/2015 na Universidade de Brasília

[2] https://computing.llnl.gov/tutorials/pthreads/ [1]

[3] http://www.cplusplus.com/doc/tutorial/ [1]

[1] Consultados ao longo do desenvolvimento do projeto para melhor entender o funcionamento das estruturas utilizadas