Tarefa 1: Modo usuário e chamadas de sistema

Escreva um código mínimo em linguagem de montagem que contenha apenas uma instrução para atribuir um valor a um registrador qualquer da CPU. Compile-o com as flags apropriadas para evitar que o compilador adicione bibliotecas ou instruções extras. Ao executar o programa, você deverá observar uma falha de segmentação (segmentation fault).

1. Explique por que essa falha ocorre.
2. Corrija o código para que a falha não aconteça mais.
3. Execute o programa com o comando `strace` antes e depois da correção e observe a diferença na sequência de chamadas de sistema.
4. Explique por que foi necessário incluir uma chamada de sistema para encerrar corretamente o programa.
5. Discuta brevemente quais seriam as implicações se os programas de usuário pudessem ser executados livremente, sem recorrer ao sistema operacional para acessar recursos básicos como o encerramento.

Enviar relatório em PDF com no máximo 4 páginas ou 2000 palavras.

Tarefa 2: Modo usuário e chamadas de sistema

Utilizando a linguagem C, escreva um programa simples que utilize a função `write()` da biblioteca padrão para imprimir uma mensagem na tela. Compile e execute o programa normalmente e, em seguida, utilize o comando `strace` para observar as chamadas de sistema realizadas durante a execução. Identifique no `strace` a chamada que realiza a escrita no terminal e observe qual mecanismo é utilizado para a transição entre o modo usuário e o modo kernel.

A seguir, reescreva o mesmo programa utilizando linguagem de montagem, implementando a chamada de sistema correspondente de forma manual (por exemplo, usando a instrução `syscall`). Compile e execute o código e, novamente, utilize `strace` para observar a diferença no fluxo de execução.

Responda:

1. Qual é o papel da instrução `syscall` no programa em Assembly?

2. O que muda na forma de execução entre o programa em C e o em Assembly?

3. Por que é necessário utilizar chamadas de sistema para realizar tarefas simples como imprimir na tela?

4. O que garante a transição segura entre os modos de execução? O que aconteceria se o modo usuário tivesse acesso direto aos dispositivos?

Enviar relatório em PDF com no máximo 4 páginas ou 2000 palavras.

Tarefa 3: Criação e identificação de processos

Escreva um programa em linguagem C que imprima na tela o identificador do processo atual (PID) e o identificador do processo pai (PPID), utilizando as funções `getpid()` e `getppid()`. Compile e execute o programa, e observe os valores exibidos. Em seguida, execute o programa múltiplas vezes e registre se os valores de PID e PPID permanecem os mesmos ou variam entre execuções. O que isso nos diz sobre a natureza dos processos?

A seguir, modifique o programa para que ele crie um novo processo utilizando a função `fork()`. O processo pai e o processo filho devem imprimir suas respectivas identificações, informando claramente quem é quem. Compile e execute o programa várias vezes, e analise os resultados.

Agora, utilize os comandos `ps`, `top` ou `htop` durante a execução do programa para observar os processos ativos. Tente identificar o processo pai e o filho em tempo real.

Responda:

1. Qual é a diferença entre `getpid()` e `getppid()`? O que eles representam?
2. O que acontece com o PID do processo filho após o `fork()`?
3. Como o sistema operacional identifica e organiza os processos em execução?
4. É possível prever quantas vezes o programa imprimirá mensagens após o `fork()`? Justifique.
5. Por que o mesmo programa pode ter múltiplos processos com identidades distintas?

Enviar relatório em PDF com no máximo 4 páginas ou 2000 palavras.

Tarefa 4: Redirecionamento usando pipe

Escreva um programa em linguagem C que executa um processo filho (por exemplo, “ls -l”) e redireciona a sua saída padrão para um pipe cujo outro lado é lido pelo pai até ser fechado.

Responda:

1. Como funciona o mecanismo de redirecionamento?
2. Por que o processo pai precisa fechar o lado da escrita da pipe?
3. O que mudaria se o processo pai quisesse redirecionar a saída de erro padrão?
4. E se quisesse redirecionar a entrada padrão?

Enviar relatório em PDF com no máximo 4 páginas ou 2000 palavras.

Tarefa 5: Concorrência com threads

Dado o trecho abaixo em linguagem C:

```c
uint64_t valor = 0;

void *thread(void *arg)
{
	size_t i = 1000000;
	while (i--) {
		valor++;
	}
}
```

Adicione ao programa acima uma função main que crie duas threads que executam a função acima concorrentemente e espere que elas terminem; ao final, imprima na tela o valor da variável `valor`.

Responda:

1. Por que, quando compilado com gcc nas suas configurações padrões, o resultado não é 2 milhões como é de se esperar?
2. Por que o resultado é 2 milhões quando habilitamos as otimizações do gcc?
3. Com as otimizações habilitadas, seria possível o valor final ser menos de 2 milhões? Por quê?
4. Descreva uma forma de implementar a função acima que resolve o problema da contagem errada. Por que ela resolve o problema?

Enviar relatório em PDF com no máximo 4 páginas ou 2000 palavras.

Tarefa 6: Escalonamento de processos

Descubra o número de núcleos de CPU disponíveis no seu sistema utilizando o comando `nproc`. Com base nesse valor N, crie N processos idênticos que executem laços infinitos para consumir CPU continuamente. Utilize ferramentas como `top`, `htop` ou `ps` para observar como o sistema operacional distribui o uso da CPU entre esses processos. Registre o tempo de CPU consumido, o estado dos processos e verifique se todos recebem tratamento justo.

Em seguida, repita o experimento com N+1 processos. Observe como o sistema lida com a sobrecarga. Todos os processos continuam recebendo tempo de CPU de forma igualitária? Justifique com base nas observações.

Na etapa seguinte, altere a prioridade de um dos processos com o comando `renice`, atribuindo-lhe uma prioridade significativamente mais alta (menor valor de nice). Observe se o uso da CPU por esse processo muda e como isso afeta os demais.

Agora, reinicie o experimento com N processos de carga e adicione um processo adicional que apenas aguarde por entrada do teclado (bloqueado em `read`). Observe o comportamento desse processo em repouso e após uma entrada ser fornecida. Modifique sua prioridade e verifique se há impacto no uso da CPU.

Por fim, reflita sobre o comportamento observado e responda:

1. O sistema distribuiu o tempo de CPU de forma justa nos diferentes cenários? Como isso está relacionado ao funcionamento do CFS?
2. O que aconteceu quando a prioridade de um processo foi alterada? Por que isso ocorreu?
3. Por que o processo bloqueado por entrada não utilizou CPU até receber dados?
4. A mudança de prioridade afeta processos bloqueados? Por quê?
5. Compare esse comportamento com o que seria esperado em algoritmos como FIFO, Round Robin e Shortest Job First (SJF).
6. Qual é a vantagem do modelo do Linux em relação aos algoritmos tradicionais?

Enviar relatório em PDF com no máximo 4 páginas ou 2000 palavras.

Tarefa 7: Algoritmo de Peterson

Dado o algoritmo de Peterson implementado na linguagem C abaixo (extraído de livro de Tanenbaum):

```
#define FALSE 0
#define TRUE  1
#define N     2                       /* number of processes */

int turn;                             /* whose turn is it? */
int interested[N];                    /* all values initially 0
(FALSE) */

void enter_region(int process)        /* process is 0 or 1 */
{
     int other;                   /* number of the other process */
     other = 1 - process;         /* the opposite of process */
     interested[process] = TRUE; /* show that you are interested */
     turn = process;              /* set flag */
     while (turn == process && interested[other] == TRUE) /* null
statement */ ;
}

void leave_region(int process)        /* process: who is leaving */
{
     interested[process] = FALSE;    /* indicate departure from
critical region */
}
```

Responda:

1. Por que usar `enter_region` e `leave_region` para estabelecer uma região crítica não funciona com compiladores modernos?
2. Como os problemas descritos na pergunta anterior podem ser resolvidos?
3. Com os problemas dos compiladores modernos resolvidos, qual problema ainda impede que o algoritmo de Peterson funcione em processadores modernos?
4. Como os problemas descritos na pergunta anterior podem ser resolvidos?

Enviar relatório em PDF com no máximo 4 páginas ou 2000 palavras.

Tarefa 8: Mutexes

Dada a implementação em linguagem C de um Mutex abaixo:

```c
atomic_bool trava = false;

void enter_region(void)
{
	bool v;
	do {
		v = false;
	} while (!atomic_compare_exchange_strong(&trava, &v, true));
}

void leave_region(void)
{
	atomic_store(&trava, false);
}
```

Responda:

1. Por que a implementação acima não é eficiente quando há contenção?
2. Como a implementação poderia ser *minimamente* modificada para usar a chamada de sistema `futex` do Linux? Como essa versão seria melhor que a original?
3. Por que a versão anterior ainda não é ideal do ponto de vista de desempenho?
4. Implemente uma versão que resolve os problemas da questão anterior. Como ela resolve os problemas?

Enviar relatório em PDF com no máximo 4 páginas ou 2000 palavras.

Tarefa 9: Produtor e consumidor

Dadas as função de produtor e consumidor em linguagem C abaixo:

```c
#define TAMANHO 10
volatile int dados[TAMANHO];
volatile size_t inserir = 0;
volatile size_t remover = 0;

void *produtor(void *arg)
{
	int v;
	for (v = 1;; v++) {
		while (((inserir + 1) % TAMANHO) == remover);
		printf("Produzindo %d\n", v);
		dados[inserir] = v;
		inserir = (inserir + 1) % TAMANHO;
		usleep(500000);
	}

	return NULL;
}

void *consumidor(void *arg)
{
	for (;;) {
		while (inserir == remover);
		printf("%zu: Consumindo %d\n", (size_t)arg,
dados[remover]);
		remover = (remover + 1) % TAMANHO;
	}

	return NULL;
}
```

Responda:

1. Qual é o problema fundamental da implementação acima?
2. Como a implementação pode ser modificada para resolver o problema usando um ou mais Mutexes?
3. A solução apenas com Mutexes tem problemas de desempenho. Quais são esses problemas?
4. Como esses problemas podem ser resolvidos com semáforos?
5. Como esses problemas podem ser resolvidos com variáveis de condição?

Enviar relatório em PDF com no máximo 4 páginas ou 2000 palavras.