EA872 Laboratório de Programação de Software Básico
Atividade 7

Vinícius Esperança Mantovani **RA 247395**

Entrega (limite): 27/09/2023, 13:00

Atividade 8.1:

1) O processo terminal só libera os recursos que ocupava quando seu processo-pai é notificado por meio da resposta do sistema à chamada “wait”, responsável por tal notificação ao pai.
2) O motivo pelo qual existem trechos chamados desse modo é que esses trechos manipulam recursos que são compartilhados entre dois ou mais programas e, tais recursos devem ser usados por apenas um desses programas por vez, sem interrupções. Então, esse nome, “região crítica”, denota a possibilidade de inconsistências no caso de tal recurso ser utilizado por diferentes programas simultaneamente.
3) Sim, isso ocorre, pois no sistema UNIX, a cada recurso compartilhado é associado um identificador, que são analisados por um algoritmo de exclusão mútua de recursos compartilhados, o que permite que, caso os identificadores sejam diferentes, esses recursos possam ser usados, cada um, por um programa distinto. Assim, é possível que dois recursos compartilhados distintos sejam utilizados simultaneamente, desde que cada um deles seja usado por um programa diferente.
4) Um processo deve, antes de usar um código de uma região crítica, executar uma chamada de sistema, componente do algoritmo de exclusão mútua, para o sistema, pedindo para executar essa região. Desse modo, sendo autorizada a execução, o processo utiliza a região e, em seguida, ao final do uso, o processo deve fazer uma chamada de outra função, também parte do algoritmo de exclusão mútua, que tem como resposta do sistema a liberação do recurso compartilhado para ser usado por outro processo.
5) Semáforos são variáveis dadas por um número. São utilizadas pelo sistema para tirar do usuário a incumbência de realizar operações de sincronização de processos. Basicamente, essas variáveis são tratadas por duas operações, UP e DOWN, responsáveis por alterar seu valor. Ela pode assumir o valor zero, indicando que não há recursos disponíveis, como forma de proteger um recurso de uma região crítica contra um possível uso simultâneo por dois processos distintos ou, caso seja

diferente de 0 seu valor (número de operações wakeup), então temos a disponibilidade de n recursos para uso, onde n é o número de operações “wakeup”. Isso se dá como um semáforo mesmo, enquanto o valor da variável é diferente de 0, os processos sabem que o recurso pode ser utilizado, mas, caso seja igual a 0, então o recurso está sendo usado, pois está sendo usado. Sua função é esta, de indicar a disponibilidade de um recurso de uma região crítica.

6) Uma operação DOWN em um semáforo indica que um recurso passou a ser usado, ou seja, um recurso deixou de estar disponível para uso, pois já está sendo utilizado por algum processo.
7) O escalonador de processos de um sistema operacional tem como objetivo analisar a prioridade de cada processo e gerenciar o tempo de uso da CPU por cada um dos processos. Esse seu objetivo é cumprido de modo que, ele analisa, por meio de critérios como eficiência, progresso e outros que definem o comportamento do escalonador. Isso porque, tendo em vista tais critérios, o escalonador define as prioridades que dará a cada processo e, consequentemente, os mantém em execução por um tempo proporcional às suas respectivas prioridades. Desse modo, ele funciona de maneira a intercalar processos que usam a CPU, dando o devido tempo de execução para cada um deles, visando a otimização desse uso.
8) O comando “nice” permite que um processo altere sua prioridade dentro de um limite permitido, sendo essa alteração limitada de forma que ele somente consegue diminuir sua prioridade para permitir que outros processos tenham prioridade maior que a sua e, portanto, tenham acesso por mais tempo à CPU. Isso se dá por meio do valor do “nível” usado no cálculo da prioridade de um processo, que é inversamente proporcional à prioridade, ou seja, quanto maior o “nível”, menor a prioridade.
9) Um exemplo de situação em que um sinal deve ser enviado para um processo é o seguinte: Suponhamos que temos um processo aguardando uma entrada, do teclado ou algum dispositivo de input. Então, o sistema o bloqueia para que a CPU não fique ociosa durante essa espera. Por fim, ocorreria o envio de um sinal para o processo, pois, deste modo, o processo sairia do estado de bloqueio e voltaria a competir pelo uso da CPU.
10) Para que um sinal enviado a um programa em execução não seja ignorado, é necessário que o ele faça uma chamada de “signal”, passando como parâmetro o nome da função de tratamento ou o indicador de função default (SIG_DFL).

Atividade 8.2: a:

O programa da presente atividade inicia imprimindo os valores do id do processo sendo executado neste momento (processo associado ao próprio programa, que chamaremos de p1) e de seu processo-pai (p0). Em seguida, chama-se “fork()”, que cria um novo processo (p2), filho do processo associado ao programa e, salva-se o retorno da função no inteiro pid. Desse modo, temos dois processos executando o código a partir da linha imediatamente abaixo de “pid = fork()”, pois o novo processo (p2) é igual ao processo-pai (p1). Assim, o processo-pai (p1), que tem o valor de pid diferente de 0 por conta do retorno da função ser o id do novo processo (p2), entra no “else” do bloco condicional, emitindo a mensagem:

Processo-pai:
pid(processo-filho)=4242

Nesta mensagem, vemos o id do processo criado (p2) sendo identificado como processo-filho de p1. Em seguida, o processo p1 faz uma chamada “sleep(1)” para parar de ser executado por 1 segundo. Nesse meio tempo, o processo p2, que tem a variável pid=0, entra no “if (pid == 0)” do bloco condicional do código e imprime a seguinte mensagem:

Processo-filho:
pid(processo-filho)=4242, pid(processo-pai)=4241

Nesta mensagem, vemos que o processo que a imprimiu se auto-denomina processo-filho (p2) e, temos os valores do id de p2 e de p1, respectivamente impressos. Depois disso, o processo p2 faz, também, uma chamada “sleep(6)” para parar sua execução por 6 segundos. Deste modo, ao fim do 1 segundo de espera do processo p1, ele imprime o seguinte no terminal:

...apos aproximadamente 1 segundo…
Processo 4241 terminou!

Esta mensagem é impressa para indicar que p1 foi finalizado. Finalmente, como o fim dos 6 segundos de espera do processo-filho (p2), é impresso o seguinte no terminal:

...apos aproximadamente 6 segundos…
pid(processo-filho)=4242, pid(processo-pai)=1748
Processo 4242 terminou!

Assim, são impressos o id do processo p2 e, em seguida o id do processo que o “adotou” após o fim do processo p1, que é o processo “systemd”, processo que substitui “init” em alguns sistemas linux, como no caso deste (debian 12). Esse processo, como o “init” se responsabiliza por “adotar processos cujo pai foi finalizado”.

Para mostrar que este processo é realmente systemd, executou-se o comando “ps -p 1748” no terminal e, obteve-se de resposta:

PID	TTY	TIME	CMD
1748	?	00:00:00	systemd

Sintetizando, o programa “a_ex.c” imprime mensagens por meio do processo associado, inicialmente, a ele e, cria um outro processo que também executa o mesmo código para que se possa analisar os id’s de cada um desses processos e entender o processo de criação e espera de processos, do modo como foi explicado acima.

Atividade 8.2: b:

O programa tem uma composição muito semelhante à composição do programa do item “a” desta atividade, porém o conteúdo do bloco condicional é diferente, sendo que o processo originário e o processo-filho imprimirão três mensagens cada. O primeiro imprime uma mensagem contendo o id de seu filho:

Processo-pai:
pid(processo-filho)=4793

Enquanto que o segundo, processo-filho, imprime seu próprio id e o id de seu pai:

Processo-filho:
pid(processo-filho)=4793, pid(processo-pai)=4792

anunciando logo em seguida que está sendo finalizado:

Processo 4793 terminou!

Por fim, o processo-pai é, também, encerrado:

Processo 4792 terminou!

Analisando agora o estado do processo-filho enquanto o processo-pai está rodando em background, temos o seguinte, para a execução do comando “ps ux” em diferentes momentos:

Vemos pela resposta do programa, que o id do processo-filho é 4836 e, buscando por ele na lista retornada pelo comando “ps ux”, temos o seguinte:

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
pininsu	4836	0.0	0.0	0	0	pts/0	Z	20:16	0:00	[ex_b] <defun

A saída foi a mesma para todos os momentos em que foi executado “ps ux” para o id do processo-filho. Assim, olhando o valor “STAT” dado como “Z”, temos que o estado do processo-filho é o estado “zumbi”, um estado transitório que ocorre quando um processo-filho termina sua execução, mas tem de manter-se ativo até que o processo-pai possa ler o status de saída do processo-filho. Isso ocorre, pois o processo-pai fica em modo de espera por 10 segundos, devido à chamada de “sleep(10)” no código, fazendo com que só volte a ser executado depois desse tempo e, consequentemente, que só possa ler o status de saída do processo-filho após isso.

Atividade 8.2: c:

O programa “ex_c.c” funciona da seguinte forma: inicialmente, são declaradas algumas variáveis e, logo em seguida, são impressos os valores de id do processo dado pelo programa (chamaremos de p1) e do processo-pai (p0). Em sequência, é feita uma chamada “fork” para criar um novo processo (p2_1), filho de p1, e, seu id é armazenado em pid1. Então, temos um bloco de condicional que será executado tanto por p1 como por p2_1, sendo que p1 entra no “else” e p2_1 entra no “if”. Deste modo, p2_1 escreve as seguintes mensagens no terminal:

Primeiro processo-filho:
pid(1o. processo-filho)=5416, pid(processo-pai)=5415

e entra em estado de espera por 10 segundos, por conta da chamada “sleep(10)”. Depois de sua espera, o processo p2_1 é finalizado, por causa da chamada “exit” que encerra a execução do programa no que tange a este processo.

Ainda durante o estado de espera de p2_1, o processo p1 executa o bloco “else” e dá origem a outro processo (p2_2), por meio de outra chamada “fork”, a qual tem seu retorno salvo em pid2 (id do processo criado). Então, ainda dentro do “else”, temos outro bloco condicional e, por ter pid2 = 0, o programa no processo-filho p2_2 entra no bloco “if (pid2 == 0)” e imprime a seguinte mensagem na tela:

```
Segundo processo-filho:
pid(2o. proc-filho)=5417, pid(processo-pai)=5415
Processo 5417 terminou!
```

amostrando seu id e o id de seu processo-pai (p1) e, indicando que ele próprio terminou sua execução. Por fim, por causa das duas primeiras linhas executadas pelo processo-pai (p1) dentro do bloco “else” interno ao bloco “else” citado anteriormente, p1 espera que ambos seus processos-filho (p2_1 e p2_2) sejam encerrados para que ele imprima o seguinte no terminal:

```
Processo-pai:
pid(1o. processo-filho)=5416
pid(2o. processo-filho)=5417
O processo 5417 terminou com estado=30
O processo 5416 terminou com estado=20
Processo 5415 terminou!
```

indicando os id’s de seus processos-filhos e, em seguida, o estado nos quais foram terminados os processos p2_2 e p2_1, respectivamente. Ademais, ainda na mensagem, o processo-pai indica que ele próprio foi terminado imediatamente antes de ser, de fato, terminado com uma chamada “exit”.

Vale destacar o motivo pelo qual o filho 1 não imprimiu a mensagem “Processo xxxx terminou!” como o filho 2 e seu pai. Isso se dá, devido ao fato de que o filho 1 é terminado com uma chama “exit” anterior à linha em que se impõe a impressão dessa mensagem. Desse modo, o processo filho 1 é finalizado antes da execução do código chegar à linha que imprimiria tal mensagem, ficando esta, portanto, sem ser impressa pelo processo-filho 1.

Expliquemos, agora, o porquê de as chamadas “wait” eximirem os processos-filhos de permanecerem em estado terminal. Isso ocorre porque, com as chamadas wait, o processo-pai aguarda até ambos os processos-filhos tenham uma saída por meio de “exit” para que, então, eles sejam postos em estado terminal e, em seguida, sejam extintos juntos de seu pai que também será extinto em breve.

Atividade 8.2: d:

O código “ex_d.c” tem, como os anteriores, uma chamada da função “fork”, que cria um novo processo-filho e armazena o id desse filho em “pid”. Assim, os

dois processos, pai e filho, executam o código a partir desse ponto, sendo que o processo pai executa os comandos de dentro do bloco “else”, enquanto que o processo filho executa os comandos do bloco “if”. O que ocorre é que o processo-pai entra em “else”, salva o tempo atual em “tv1” e, chega na linha “wait(&estado);”, então, ele aguarda até que seu processo-filho faça uma chamada de “exit” terminando sua execução. Neste meio tempo, o processo-filho imprime a mensagem:

pid(processo-filho)=8022

indicando seu id e, subsequentemente, faz uma chamada de “execlp” que imprime um calendário no seguinte formato:

```
   September 2023
Su Mo Tu We Th Fr Sa
                1  2
 3  4  5  6  7  8  9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
```

por meio da chamada de sistema “execlp”, que desvia o processo para executar o código de “cal” com os argumentos passados no comando de execução do programa “ex_d.c” (mês e ano).

Em seguida, com a chamada de “exit” do processo filho, o processo-pai continua e salva o novo tempo atual em “tv2”. Assim, logo em seguida, o processo-pai imprime seu id e, segue para o cálculo do tempo que ele esperou até que o processo-filho chamasse “exit”. Por fim, o processo pai imprime a mensagem:

O tempo de execucao do processo 8022 e' 0.000702, terminando com estado=30
Processo 8021 terminou!

indicando o tempo calculado de execução do processo-filho e, informando que o processo-pai está por terminar.

Por fim, executando o programa diversas vezes, nota-se que o tempo de execução do processo-filho varia entre valores próximos de 0,0017, conforme se nota abaixo:

Exec 1°: 0.001795
Exec 2°: 0.001713
Exec 3°: 0.001713
Exec 4°: 0.001784

No entanto, nota-se que, ao retirarmos a linha 16, os tempos registrados em diferentes execuções varia entre valores próximos de 0,000557, conforme o que se segue:

Exec 1°: 0.000557
Exec 2°: 0.000452

Exec 3°: 0.000519
Exec 4°: 0.000483

Assim, podemos concluir, conforme o esperado, que a execução do comando “cal” por meio da chamada “execlp” aumenta em aproximadamente três vezes o tempo de execução do processo-filho.

Atividade 8.2: e:

Abaixo, estão os testes feitos e as justificativas para os comportamentos de cada um deles:

Exec 1°:

Primeiro trecho de tratamento de ctrl-c
^C^C^C^C
Segundo trecho de tratamento de ctrl-c
^CO sinal SIGINT foi captado. Continue a execucao!

Terceiro trecho de tratamento de ctrl-c
^C

Nesta execução, logo que é impresso a primeira mensagem no terminal, informando que é iniciado o primeiro trecho de tratamento de ctrl-c, pressionou-se quatro vezes ctrl-c e, notou-se que, em nenhuma delas houve qualquer resposta que fosse do terminal. Isso se dá, por causa de termos, no código, uma linha impondo que o sinal de ctrl-c recebido seja ignorado. Tal linha é “signal(SIGINT, SIG_IGN);” e, a imposição de se ignorar o sinal é dada pelo argumento “SIG_IGN”, enquanto que o argumento “SIGINT” indica que se refere ao tratamento do sinal de ctrl-c.

Ademais, no segundo trecho, pressionou-se, novamente ctrl-c e, o resultado foi a impressão da mensagem “O sinal SIGINT foi captado. Continue a execucao!”. Isso ocorre porque, logo após a impressão da mensagem indicando que entramos no segundo trecho de tratamento, temos uma chamada de “signal”, agora, com o segundo argumento sendo “ger_nova”, que é a função que se deseja usar para tratar o sinal. Desse modo, como a função imprime a mensagem em questão, quando pressionamos ctrl-c nesse trecho de tratamento, temos a impressão dessa mensagem e o fim do estado de espera.

Por fim, no terceiro trecho, temos a imposição, também por uma chamada de “signal”, para que o sinal de ctrl-c seja tratado da maneira padrão com que o sistema o trata. Sendo assim, vemos, conforme esperado, que, ao pressionarmos ctrl-c no terceiro trecho de tratamento, o programa é encerrado pelo sistema. Isso porque, o tratamento padrão da tecla ctrl-c no terminal é de encerrar o programa em execução neste terminal.

Exec 2°:

Primeiro trecho de tratamento de ctrl-c

Segundo trecho de tratamento de ctrl-c

Terceiro trecho de tratamento de ctrl-c

Quarto trecho de tratamento de ctrl-c
^CO sinal SIGINT foi captado. Continue a execucao!

Tchau!

Nesta execução, para termos testado todas as possibilidades de comportamento de cada trecho de tratamento, não foi pressionado ctrl-c em nenhum dos três primeiros trechos de tratamento, somente no quarto. Desse modo, notamos que, nesses três primeiros trechos, a ausência de pressionamentos de ctrl-c não causa a ocorrência de coisa alguma, uma vez que estamos tratando de sinais de ctrl-c, logo, se não temos sinais, consequentemente, não temos tratamento nem resposta.

Por fim, notamos, no quarto trecho, a impressão de “O sinal SIGINT foi captado. Continue a execucao!” e, na segunda linha depois dessa, “Tchau!”, Isso acontece, pois na linha em que se chama “signal” no terceiro trecho, definimos o ponteiro de void “ger_antiga” como sendo o ponteiro que aponta para a função anteriormente passada como função de tratamento de sinal na função “signal“. Ou seja, ocorre que “ger_antiga” passa a ser a mesma função que “ger_nova”, portanto, ao pressionar ctrl-c no trecho quatro, temos a mesma resposta que temos no trecho 2 e, em seguida, a impressão de “Tchau!”, cuja responsável é a última linha da main (“printf ("\nTchau!\n");”).

Observação: as demais execuções não tem os resultados justificados por obviedade e semelhança com as execuções anteriores.

Exec 3°:

Primeiro trecho de tratamento de ctrl-c
^C^C^C^C^C
Segundo trecho de tratamento de ctrl-c
^CO sinal SIGINT foi captado. Continue a execucao!

Terceiro trecho de tratamento de ctrl-c

Quarto trecho de tratamento de ctrl-c

Tchau!

Exec 4°:

Primeiro trecho de tratamento de ctrl-c
^C
Segundo trecho de tratamento de ctrl-c

Terceiro trecho de tratamento de ctrl-c
^C

Existem, ainda, outras combinações possíveis para analisarmos as saídas, no entanto, as explicações das duas primeiras saídas são o suficiente para se conhecer plenamente o comportamento do programa.

Atividade 8.2: f:

O programa funciona da seguinte forma:

Já no começo da main, é determinada a função de tratamento de sinais de término de processos-filhos (SIGCHLD), “ger_sinal”, por meio de uma chamada “signal” com esses dois argumentos. Assim, a partir desse momento, o término de um ou mais processos-filhos ocasionará a entrada do processo-pai nessa função. Em sequência, são criados dois processos-filhos, que anunciam, cada um deles, sua própria atividade, por meio das mensagens:

- Processo 3945 esta' ativo/em execucao
o Processo 3946 esta' ativo/em execuca

sendo cada uma das linhas impressa por um dos processos-filhos. Vale destacar que o processo-pai não entra em nenhuma das duas condicionais “if”, pois o valor de pid1 e pid2 em sua execução são ambos diferentes de 0, pois têm valores iguais aos id’s de cada processo-filho (cada um de um processo-filho).

Seguindo, como cada um dos processos-filhos ficam presos em uma das duas condicionais, por conta dos loops “while” contidos nos blocos condicionais, ocorre que apenas o processo-pai segue para o resto do código. Desse modo, temos as seguintes impressões a seguir:

12- Processo 3945 esta' ativo/em execucao
o Processo 3946 esta' ativo/em execuca

em que os processos-filhos imprimem 1 e 2 indicando que seu processo de espera da primeira iteração acabou e, o resto da mensagem, indicando que os processos-filhos estão ativos ainda. Fazem isso durante a espera de 5 segundos do processo-pai;

12- Processo 3945 esta' ativo/em execucao
o Processo 3946 esta' ativo/em execuca
12- Processo 3945 esta' ativo/em execucao
o Processo 3946 esta' ativo/em execuca
12- Processo 3945 esta' ativo/em execucao
o Processo 3946 esta' ativo/em execuca

aqui, notamos que o mesmo ocorrido explicado acima se repete mais três vezes de maneira exatamente igual. Ou seja, no tempo de 5 segundos, temos 5 impressões por cada um dos processos-filhos (5 iterações).

Por fim, o processo pai imprime a seguinte sequência de estados:

3944:vou parar o 1o.3944: Um sinal de mudanca do estado do processo-filho 0 foi captado!

3944:vou parar o 2o.3944: Um sinal de mudanca do estado do processo-filho 0 foi captado!

indicando que o processo-pai para a execução de cada um dos processos filhos, o que é tratado por “ger_sinal” quando é retornado um sinal de mudança de estado de filhos, do seguinte modo: ao receber o sinal, o fluxo do programa do processo-pai é desviado para a função “ger_sinal”, que implementa um loop durante o tempo em que o id do processo-filho é maior que 0 e, durante esse loop, é verificado sempre se esse id ainda não é zero por meio de uma chamada “wait3” que verifica se houve mudança de estado do processo-filho. Assim, ao perceber uma mudança, temos o estado salvo na variável “estado”, que é impresso juntamente com o id do processo-filho enquanto ele não foi extinto.

Seguindo, temos as seguintes mensagens:

3944:vou continuar o 1o.3944: Um sinal de mudanca do estado do processo-filho 0 foi captado!

3944:vou matar o 2o.3944: Um sinal de mudanca do estado do processo-filho 0 foi captado!

impressas, também, ambas pelo processo-pai, indicando que ele continuará o processo-filho 1 e 2, embora a segunda mensagem esteja escrita de maneira incorreta (substitua “matar” por continuar). Essas mensagens são produzidas do mesmo modo que aquelas citadas acima.

Quase no fim, temos agora a seguinte sequência de mensagens sendo impressa:

3944:vou matar o 1o.3944: Um sinal de mudanca do estado do processo-filho 3945 foi captado!
O processo-filho 3945 foi extinto! Estado=0
em loop! 3944: Um sinal de mudanca do estado do processo-filho 0 foi captado!

em que o processo-pai anuncia que matará o 1° processo-filho e, a função “ger_sinal” imprime um aviso de que o sinal de mudança de estado do processo-filho foi recebida pelo processo-pai e que o processo filho foi extinto (pois, como pid na função ger_sinal passa a ser maior que 0, ocorre a entrada no bloco condicional “if”).

Temos, então, do mesmo modo que anteriormente, as seguintes mensagens:

3944:vou matar o 2o.
3944: esperando sinal "chegar"...3944: Um sinal de mudanca do estado do processo-filho 3946 foi captado!

O processo-filho 3946 foi extinto! Estado=0
em loop! 3944: Um sinal de mudanca do estado do processo-filho -1 foi captado!

que indicam o mesmo processo, mas com relação, agora, ao processo-filho 2°.

Por fim, o processo-pai entra em um loop “for” que informa que o processo está aguardando o sinal de extinção dos processos-filhos chegarem, até que o processo-pai é posto em modo de espera por um segundo e termina, então, sua execução. Isso se amostra a seguir, nas mensagens impressas:

3944: esperando sinal "chegar"...
3944: esperando sinal "chegar"...
3944: esperando sinal "chegar"...
3944: esperando sinal "chegar"...

Testando no caso de tirarmos a chamada de “wait3”, acontece um comportamento indeterminado no loop de “ger_sinal”, uma vez que o valor de pid fica indefinido, sendo assumido, portanto, o valor que se encontra inicialmente no bloco de memória em que foi alocada a variável “pid”. Sendo assim, podemos tanto ter a finalização desse loop em uma iteração, como podemos, também, ter esse loop executando eternamente.

Por fim, constata-se que, conforme descrição das funções, a chamada “wait3” difere da chamada “wait” por conta de a primeira aguardar uma mudança de estado do processo-filho, enquanto que a segunda aguarda o encerramento de um processo-filho, ou seja, suspende a continuação do fluxo do processo-pai até que ele receba informações do processo-filho que está por ser encerrado.