

**UNIVERSIDADE FEDERAL DE UBERLÂNDIA  
FACULDADE DE ENGENHARIA ELÉTRICA  
ENGENHARIA DE CONTROLE E AUTOMAÇÃO**

**ATIVIDADE 3 – LINUX COMO AMBIENTE DE PROGRAMAÇÃO**

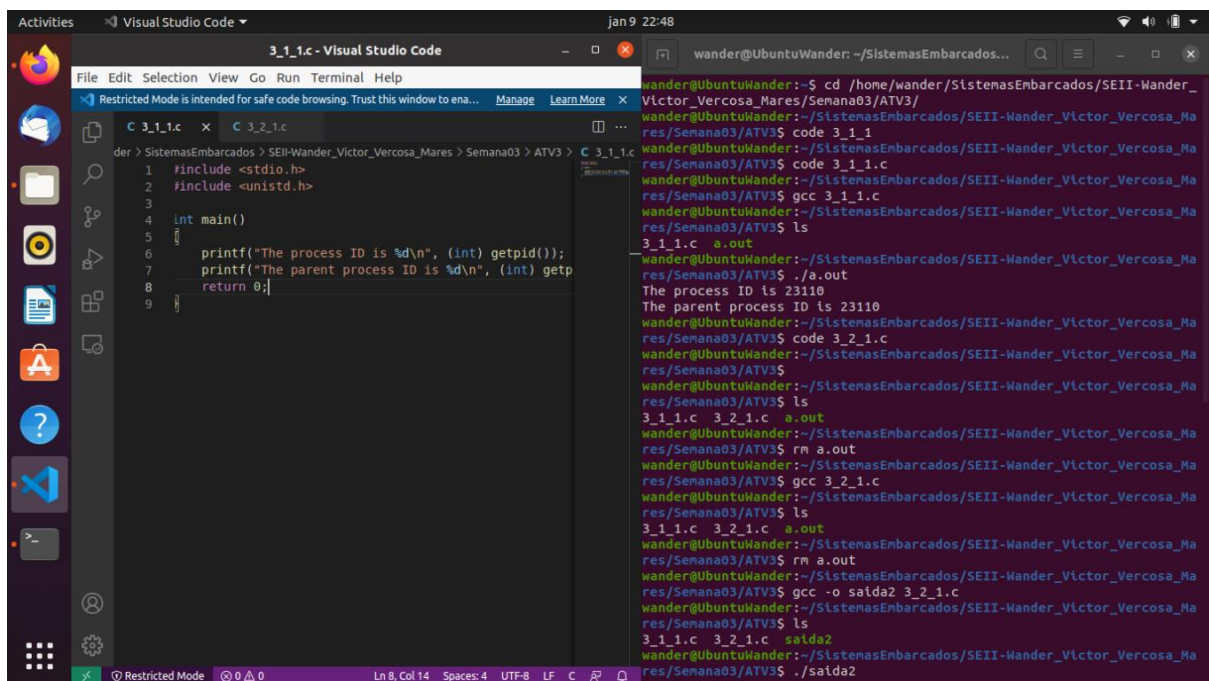
Aluno: Wander Victor Verçosa Mares - 11811EAU010  
Professor: Éder Alves de Moura

Uberlândia  
10/01/2022

### 3. Faça um resumo de todas as seções do Capítulo 3, do livro *Advanced Linux Programming*, e implemente os exemplos disponibilizados.

**3.1.1** - Cada processo no sistema Linux é identificado por um processo único, pode ser referido também como *pid*. Estes processos são identificados em números de 16 bits.

Todo processo também tem um processo parente (exceto o *init* ou “zombie process”). Portanto, podemos pensar os processos no Linux como uma árvore, onde sempre haverá uma ramificação de um processo raiz.



The image shows a Visual Studio Code editor window with a C file named `3_1_1.c`. The code defines a `main` function that prints the process ID and parent process ID using `getpid()` and `getppid()`. To the right, a terminal window shows the execution of this code. The user runs `code 3_1_1`, then `gcc 3_1_1.c`, and finally `./a.out`. The output shows the process ID as 23110 and the parent process ID as 23110. The user then runs `code 3_2_1.c`, `gcc 3_2_1.c`, and `./a.out`, which produces no output. Finally, the user runs `gcc -o salda2 3_2_1.c` and `./salda2`, which produces no output.

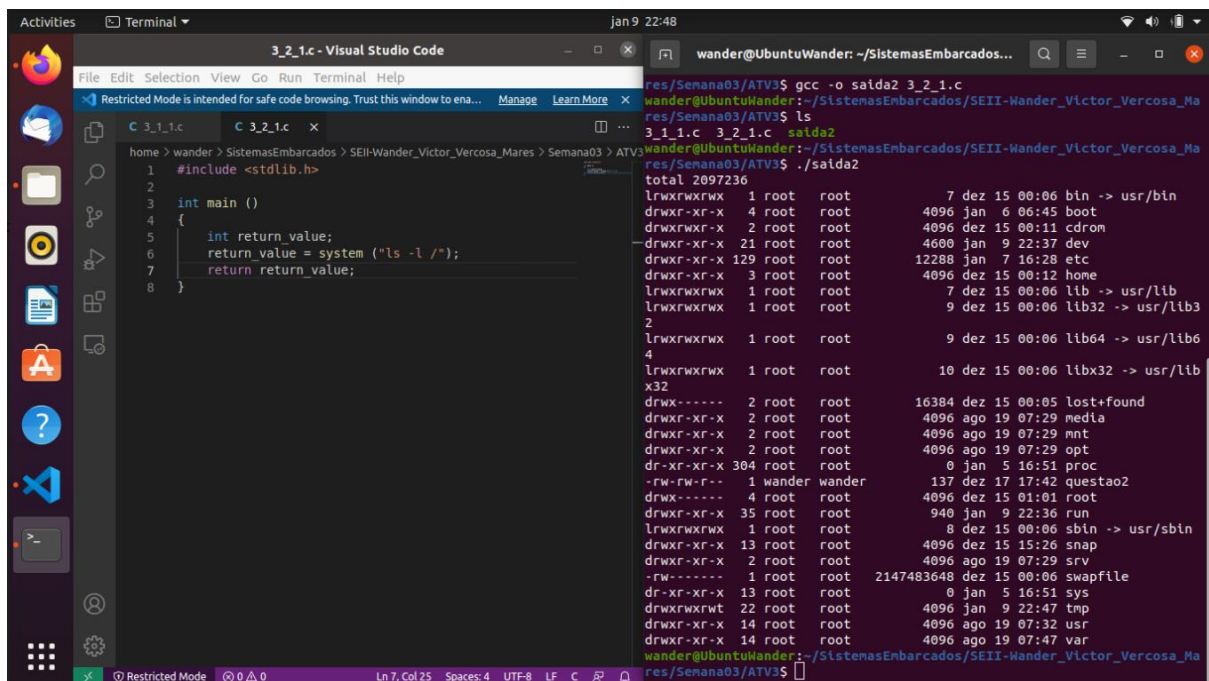
**3.1.2** – O comando *ps* exibe os processos que estão “rodando” no seu sistema.

Ao exibir o painel de processos, o primeiro será o *bash*, seguido do *ps program* e de outros processos subsequentes.

**3.1.3** – Você pode “matar” um processo com uso do comando *kill*. Basta digitar o mesmo junto ao ID do processo em questão, feito isso, o processo recebe um comando para término de tarefa.

**3.2** - Há duas técnicas comuns utilizadas para criar novos processos. A primeira é relativamente simples, mas, possui riscos de seguranças e deveria ser utilizada de modo moderado. A segunda é mais complexa, no entanto, garante mais flexibilidade, velocidade e segurança.

**3.2.1** - Usando função do sistema em biblioteca C, esse caminho garante a execução de um comando proveniente de um programa, que pode ser digitado dentro do *shell*.

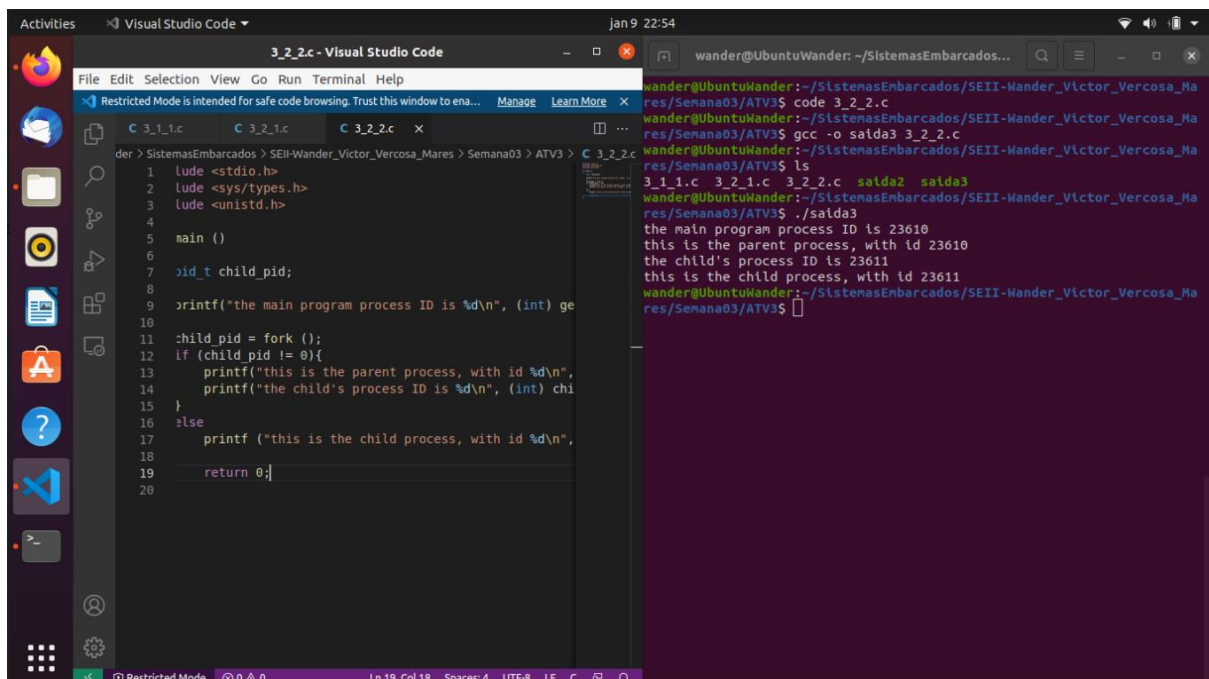


```
res/Semana03/ATV3$ gcc -o saida2 3_2_1.c
wander@UbuntuWander: ~/SistemasEmbarcados/SEII-Wander_Victor_Vercosa_Mares/Semana03/ATV3$ ls
3_1_1.c 3_2_1.c saida2
wander@UbuntuWander: ~/SistemasEmbarcados/SEII-Wander_Victor_Vercosa_Mares/Semana03/ATV3$ ./saida2
total 2097236
lrwxrwxrwx 1 root root 7 dez 15 00:06 bin -> usr/bin
drwxr-xr-x 4 root root 4096 jan 6 06:45 boot
drwxrwxr-x 2 root root 4096 dez 15 00:11 cdrom
drwxr-xr-x 21 root root 4600 jan 9 22:37 dev
drwxr-xr-x 129 root root 12288 jan 7 16:28 etc
drwxr-xr-x 3 root root 4096 dez 15 00:12 home
lrwxrwxrwx 1 root root 7 dez 15 00:06 lib -> usr/lib
lrwxrwxrwx 1 root root 9 dez 15 00:06 lib32 -> usr/lib32
lrwxrwxrwx 1 root root 9 dez 15 00:06 lib64 -> usr/lib64
drwxr-xr-x 1 root root 10 dez 15 00:06 libx32 -> usr/libx32
drwx----- 2 root root 16384 dez 15 00:05 lost+found
drwxr-xr-x 2 root root 4096 ago 19 07:29 media
drwxr-xr-x 2 root root 4096 ago 19 07:29 mnt
drwxr-xr-x 2 root root 4096 ago 19 07:29 opt
dr-xr-xr-x 304 root root 0 jan 5 16:51 proc
-rw-rw-r-- 1 wander wander 137 dez 17 17:42 questao2
drwx----- 4 root root 4096 dez 15 01:01 root
-rwxr-xr-x 35 root root 940 jan 9 22:36 run
lrwxrwxrwx 1 root root 8 dez 15 00:06 sbin -> usr/sbin
drwxr-xr-x 13 root root 4096 dez 15 15:26 snap
drwxr-xr-x 2 root root 4096 ago 19 07:29 srv
-rw----- 1 root root 2147483648 dez 15 00:06 swapfile
dr-xr-xr-x 13 root root 0 jan 5 16:51 sys
drwxrwxrwt 22 root root 4096 jan 9 22:47 tmp
drwxr-xr-x 14 root root 4096 ago 19 07:32 usr
drwxr-xr-x 14 root root 4096 ago 19 07:47 var
wander@UbuntuWander: ~/SistemasEmbarcados/SEII-Wander_Victor_Vercosa_Mares/Semana03/ATV3$
```

O sistema retorna o status de saída do comando shell, se o shell não conseguir se executar o sistema retorna 127, se outro erro ocorrer, o sistema retorna -1.

**3.2.2 - O Linux fornece uma função, fork, que cria um processo filho exatamente igual ao processo pai.**

Quando um programa chama *fork*, um processo é duplicado, chamado assim então de processo filho.



```
wander@UbuntuWander: ~/SistemasEmbarcados/SEII-Wander_Victor_Vercosa_Mares/Semana03/ATV3$ gcc -o saida3 3_2_2.c
wander@UbuntuWander: ~/SistemasEmbarcados/SEII-Wander_Victor_Vercosa_Mares/Semana03/ATV3$ ./saida3
the main program process ID is 23610
this is the parent process, with id 23610
the child's process ID is 23611
this is the child process, with id 23611
wander@UbuntuWander: ~/SistemasEmbarcados/SEII-Wander_Victor_Vercosa_Mares/Semana03/ATV3$
```

A função *exec* substitui o programa que está rodando em um processo por um outro programa.

The screenshot shows a Visual Studio Code editor with a C file named `3_2_2_1.c`. The code implements a `spawn` function that forks a child process and runs a command from an argument list. The `main` function calls `spawn("ls", arg_list)` with `arg_list` containing `"ls"`, `"-l"`, `"/*"`, and `NULL`. The terminal on the right shows the execution of the program, displaying the output of `ls` and process IDs. It also shows compilation commands: `gcc -o saida3 3_2_2.c` and `gcc -o saida4 3_2_2_1.c`. A compiler warning is visible: `error: 'child_pild' undeclared (first use in this function); did you mean 'child_pid'?`.

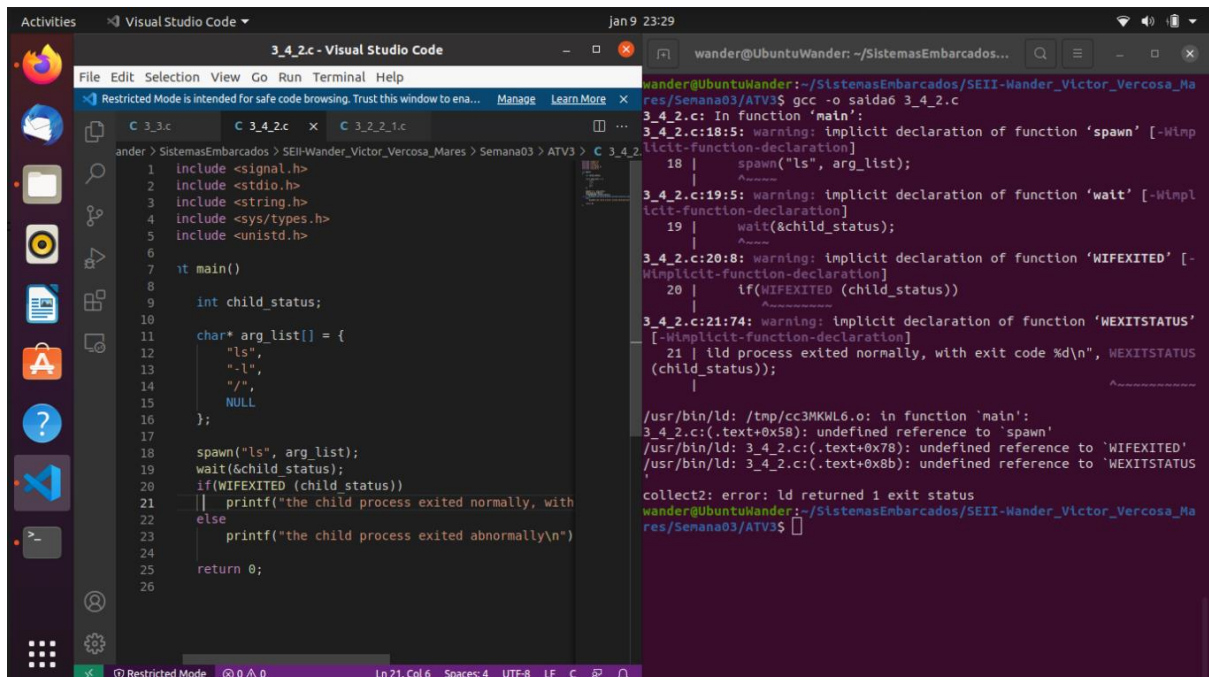
**3.2.3 –** O Linux programa os processos parentes e filhos independentemente. Não há garantias de qual irá ocorrer primeiro, ou por quanto tempo. No entanto, você pode definir grau de importância, fornecendo grau de menor prioridade.

**3.3 –** *Signals* são mecanismos para comunicação com manipulação de processos no Linux. Um *signal* é uma mensagem especial enviada a um processo, estes são assíncronos. Quando um processo recebe um *signal* ele processa o mesmo imediatamente, sem necessariamente terminar a tarefa anteriormente atribuída.

The screenshot shows a Visual Studio Code editor with a C file named `3_3.c`. The code includes `<signal.h>`, `<stdio.h>`, `<string.h>`, `<sys/types.h>`, and `<unistd.h>`. It defines a `sig_atomic_t` variable `sigusr1_count` and a `handler` function that increments this count. The `main` function sets up a `sigaction` structure to catch `SIGUSR1` signals and prints the count when raised. The terminal on the right shows the compilation command `gcc -o saida5 3_3.c` and the execution of the program, which outputs `SIGUSR1 was raised 0 times`.



**3.4** – Normalmente, um processo termina em umas de duas maneiras. Ambas executam um chamado para fim da função. Um processo também pode ser terminado via *signal*. Por exemplo: SIGBUS, SIGSEV e SEGVPE são sinais que finalizam um processo.



The screenshot shows the Visual Studio Code editor with a C program in the left pane and its compilation output in the right pane. The C program, named `3_4_2.c`, includes `<signal.h>`, `<stdio.h>`, `<string.h>`, `<sys/types.h>`, and `<unistd.h>`. It defines a `main` function that creates an argument list `arg_list` containing `"ls"`, `"-l"`, and `NULL`. It then calls `spawn("ls", arg_list)`, waits for the child process with `wait(&child_status)`, and prints the exit status using `WIFEXITED` and `WEXITSTATUS` macros. The compilation output in the terminal shows several warnings about implicit declarations of `spawn`, `wait`, `WIFEXITED`, and `WEXITSTATUS`. The linker command `ld` returns an error: `collect2: error: ld returned 1 exit status`, indicating a linking failure due to undefined references to the `spawn` and `wait` functions.

Segue um exemplo em que o comando `ls`, em primeiro caso, é executado corretamente e retorna uma saída de código zero. No segundo caso, é apresentado um erro que retorna 1.

**3.4.1** – Executando os comandos *fork* e *exec*, você pode não ter notado que a saída proveniente do programa `ls`, depois do “main program” já foi concluída. Isso se deve pelo fato de o processo filho ser agendado independente do processo pai.

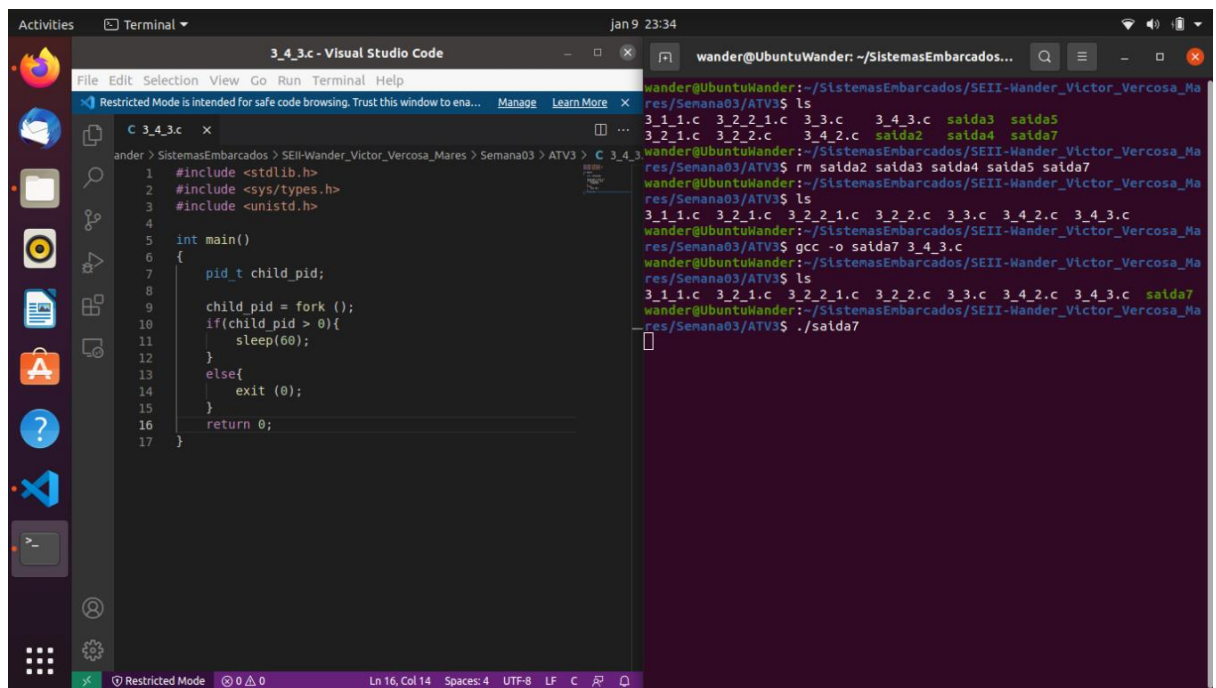
**3.4.2** – A função *wait* retorna um código de status por meio de um argumento em ponteiro inteiro, no qual podem ser extraídas informações sobre o processo filho encerrado.

Você pode usar a macro `WIFEXITED` para determinar a partir do status de saída de um processo filho se esse processo saiu normalmente.

**3.4.3** – Se um processo filho termina enquanto o processo pai está chamando uma função de espera, o processo filho desaparece e seu status de término é passado para o processo pai por meio de uma *wait* call.

No entanto, no caso de um processo filho terminar e o processo pai não estiver chamado um *wait* o processo filho é denominado *zombie*.

Um processo *zombie* é um processo que foi encerrado, mas ainda não foi limpo. É responsabilidade do processo pai limpar o *zombie child*.

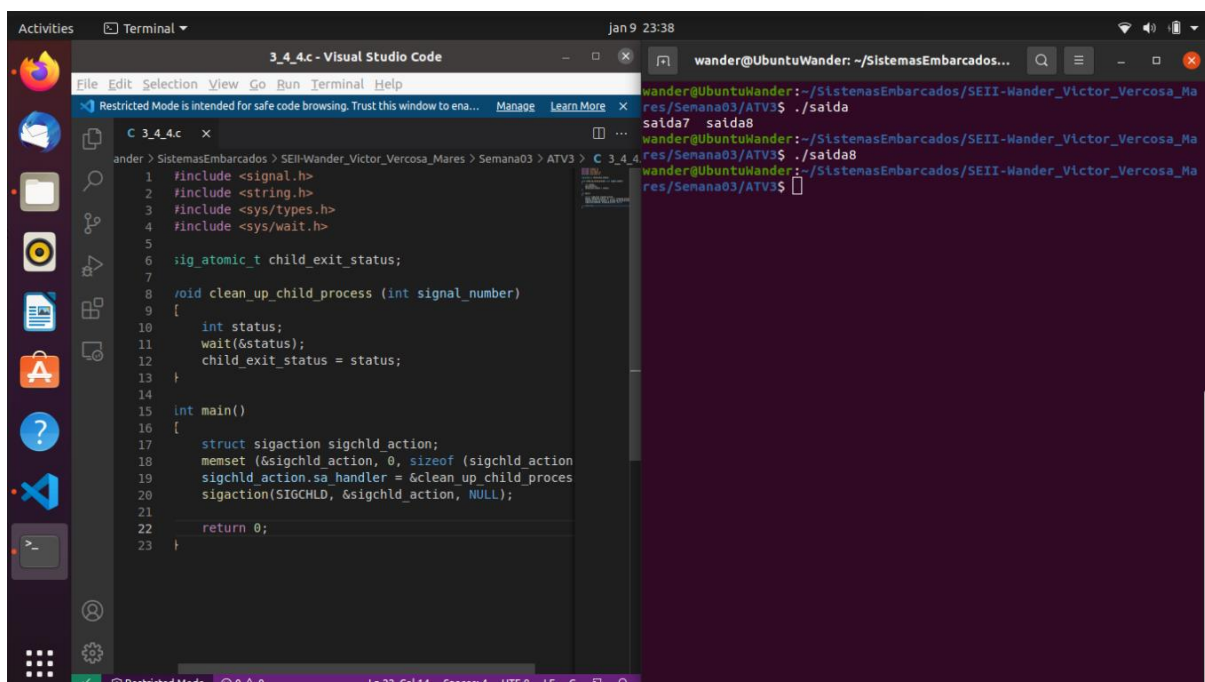


```
1 #include <stdlib.h>
2 #include <sys/types.h>
3 #include <unistd.h>
4
5 int main()
6 {
7     pid_t child_pid;
8
9     child_pid = fork ();
10    if(child_pid > 0){
11        sleep(60);
12    }
13    else{
14        exit (0);
15    }
16    return 0;
17 }
```

```
wander@UbuntuWander: ~/SistemasEmbarcados/SEII-Wander_Victor_Vercosa_Ma
res/Semana03/ATV3$ ls
3_1_1.c 3_2_2_1.c 3_3.c 3_4_3.c saida3 saida5
3_2_1.c 3_2_2.c 3_4_2.c saida2 saida4 saida7
wander@UbuntuWander: ~/SistemasEmbarcados/SEII-Wander_Victor_Vercosa_Ma
res/Semana03/ATV3$ rm saida2 saida3 saida4 saida5 saida7
wander@UbuntuWander: ~/SistemasEmbarcados/SEII-Wander_Victor_Vercosa_Ma
res/Semana03/ATV3$ ls
3_1_1.c 3_2_2_1.c 3_2_2.c 3_3.c 3_4_2.c 3_4_3.c
wander@UbuntuWander: ~/SistemasEmbarcados/SEII-Wander_Victor_Vercosa_Ma
res/Semana03/ATV3$ gcc -o saida7 3_4_3.c
wander@UbuntuWander: ~/SistemasEmbarcados/SEII-Wander_Victor_Vercosa_Ma
res/Semana03/ATV3$ ./saida7
```

**3.4.4 –** Se você estiver usando um processo filho simplesmente para executar outro programa, não há problema em chamar *wait* imediatamente no processo pai, que será bloqueado até que o processo filho seja concluído.

Portanto, uma maneira fácil de limpar os processos filhos é manipulando SIGCHLD. Obviamente, ao limpar o processo filho, é importante armazenar seus status de encerramento se essa informação for necessária, porque uma vez que o processo é limpo usando esperar, essa informação não estará mais disponível.



```
1 #include <signal.h>
2 #include <string.h>
3 #include <sys/types.h>
4 #include <sys/wait.h>
5
6 sig_atomic_t child_exit_status;
7
8 void clean_up_child_process (int signal_number)
9 {
10     int status;
11     wait(&status);
12     child_exit_status = status;
13 }
14
15 int main()
16 {
17     struct sigaction sigchld_action;
18     memset(&sigchld_action, 0, sizeof(sigchld_action));
19     sigchld_action.sa_handler = &clean_up_child_process;
20     sigaction(SIGCHLD, &sigchld_action, NULL);
21
22     return 0;
23 }
```

```
wander@UbuntuWander: ~/SistemasEmbarcados/SEII-Wander_Victor_Vercosa_Ma
res/Semana03/ATV3$ ./saida
saida7 saida8
wander@UbuntuWander: ~/SistemasEmbarcados/SEII-Wander_Victor_Vercosa_Ma
res/Semana03/ATV3$ ./saida8
wander@UbuntuWander: ~/SistemasEmbarcados/SEII-Wander_Victor_Vercosa_Ma
res/Semana03/ATV3$
```

#### **4. Faça um resumo dos tópicos apresentados nas seguintes seções... (vídeo no youtube).**

##### **1. About Linus and Linux Creation**

Linus Torvald foi um estudante da Universidade de Helsinki e desenvolveu a primeira versão do Linux em 1991. Inicialmente o sistema era um emulador de terminal, ele escreveu um programa especificamente para o hardware que estava usando e independente de um sistema operacional, porque queria usar as funções de seu novo computador com um processador 80386.

##### **2. How Linux Kernel was same and different...**

O kernel do Linux é de código aberto baseado no Unix, portanto, todos tem acesso ao código fonte. Isso significa que qualquer pessoa pode trabalhar no desenvolvimento, e é livre para usar como quiser.

O kernel do Linux possui arquitetura monolítica, além de manter suas configurações na forma de arquivos e permite um ambiente multiusuário.

##### **5. How coding inside the kernel...**

O espaço do kernel é onde o núcleo do sistema operacional funciona e fornece seus serviços. É algo que o usuário não pode interferir. Já o espaço do usuário é a parte da memória do sistema em que os processos são gerenciados pelo kernel.

As principais diferenças são: no kernel não há bibliotecas ou cabeçalhos padrão; no kernel o código é feito somente em linguagem C; no kernel o espaço de memória é limitado, portanto alguns processos rodados nesse local podem apresentar “kill the process” devido à falta de memória.

##### **6. How process are tracked...**

O processo atual deve ser processado antes que outro processo possa ser selecionado para execução. Se a política de agendamento dos processos atuais for “redonda”, então ela será colocada na parte de trás da fila de execução. Se a tarefa for interrompida e receber um *signal* a última vez que foi agendada, seu estado se torna RUNNING.

Se o processo atual tiver sido cronometrado, então seu estado se tornará RUNNING. Se o processo atual estiver em execução, então ele permanecerá nesse estado.

Processos que não estavam funcionando nem interrupções são removidos da fila de execução. Isso significa que eles não serão considerados para correr quando o agendador procurar o processo mais merecido para executar.

##### **7. Threads in Linux**

O Linux tem uma única implementação de threads. O kernel do Linux não fornece nenhuma semântica de agendamento especial ou estruturas de dados para representar threads. Em vez disso, a ideia de thread é apenas um processo que compartilha certos recursos com outros processos.

Cada segmento tem uma `task_struct` única e aparece no kernel como um processo normal.

## **8. Process Scheduling and...**

As políticas de agendamento são regras que o agendador segue para determinar o que deve ser executado e quando. Essa política considera dois processos: os vinculados à I/O e os processos vinculados à CPU.

Processos vinculados a I/O passam a maior parte do tempo esperando que as operações I/O, como solicitação de rede ou operação de teclado, sejam concluídas.

Já processos vinculados à CPU passam a maior parte do tempo executando o código. Estes, são frequentemente antecipados porque bloqueiam muitas vezes as solicitações de I/O.

O kernel usa dois valores prioritários separados. Um bom valor (-20 a +19), e um valor prioritário em tempo real (0 a 99). Para o primeiro caso, quanto maior o valor menos a prioridades, já no segundo caso a ideia é contrária, quanto maior o valor maior é a prioridade.

O *timeslice* representa quanto tempo um processo pode ser executado antes de ser antecipado. A política do agendador deve decidir sobre um timeslice padrão.

## **9. What is a system call, how to...,**

Chamadas do sistema são como um programa entra no kernel para executar alguma tarefa. Os programas usam chamadas do sistema para executar uma variedade de operações como: criação de processos, fazer IO de rede e arquivo, entre outros. Essas chamadas de sistema variam entre as arquiteturas da CPU.

## **10. System Call implementation...**

A chamada de sistema é implementada por uma “interrupção de software” que transfere o controle para o código do kernel. A chamada específica do sistema que está sendo invocada é armazenada no registro, seus argumentos são mantidos nos outros registros de processadores.

Após a mudança para o kernel, o processador deve salvar todos os seus registros e despachar a execução para função adequada do kernel, depois verificar se está fora de alcance.

## **12. What is na interrupt and how They are...**



Uma interrupção é um evento que altera o fluxo normal de execução de um programa e pode ser gerado por dispositivos de hardware ou até mesmo pela própria CPU. Quando uma interrupção ocorre o fluxo atual de execução é suspenso e interrompe as corridas do manipulador. Depois que o manipulador de interrupção executa o fluxo de execução anterior é retomado.

### **13. What is an IRQ?**

Um dispositivo que suporta interrupções tem um pino de saída usado para sinalizar um Interrupt ReQuest (IRQ). Os pinos IRQ estão conectados a um dispositivo chamado Programmable Interrupt Controller (PIC).

### **15. About critical regions and race...**

Um *race condition* geralmente envolvem um ou mais processos acessando um recurso compartilhado (arquivo ou variável), onde esse acesso múltiplo não foi devidamente controlado. Os *race conditions* podem ser definidos como:

*Interferência causa por processos não confiáveis.* Algumas taxonomias de segurança chamam esse problema de condição de “sequência” ou “não atômica”. Essas são condições causadas por processos em execução de outros programas diferentes.

*Interferência causa por processos confiáveis.* Algumas taxonomias chamam esses impasses, *livelock* ou condições de falha de bloqueio. Estas condições são causadas por processos que executam o mesmo programa.

### **17. Understanding Kernel Notion of Time**

O tempo de processo é definido como a quantidade de tempo de CPU usado por um processo. Isso às vezes é dividido em usuário e componentes de sistema. O tempo de CPU do sistema é o tempo gasto pelo kernel executando em modo de sistema em nome do processo.

A maioria dos computadores tem um relógio de hardware (alimentado por bateria) que o kernel lê no momento de inicialização para inicializar o software.

### **19. Kernel Memory Management Theory**

O sistema operacional faz com que o sistema pareça como se tivesse uma quantidade maior de memória do que realmente tem. A memória virtual pode ser muitas vezes maior do que a memória física no sistema.

Cada processo no sistema tem seu próprio espaço de endereço virtual. O mapeamento de memória é usado para mapear arquivos de imagem e dados em um espaço de endereço de processos. No mapeamento de memória, o conteúdo de um arquivo está ligado diretamente ao espaço de endereço virtual de um processo.

## **24. Filesystem Abstraction Layer**

Tal interface genérica para qualquer tipo de sistema de arquivos só é viável porque o próprio kernel implementa uma camada de abstração em torno de sua interface de sistema de arquivos de baixo nível. Esta camada de abstração permite que o Linux suporte diferentes sistemas de arquivos, mesmo que eles diferem muito em recursos ou comportamentos suportados.

O resultado é uma camada geral de abstração que permite que o kernel suporte muitos tipos de sistemas de arquivos de forma fácil e limpa. Os sistemas de arquivos são programados para fornecer as interfaces abstratas e estruturas de dados que o VFS espera; por sua vez, o kernel funciona facilmente com qualquer sistema de arquivos e a interface de espaço de usuário exportada funciona perfeitamente em qualquer sistema de arquivos.