

Atividade 3: Backlog e Processos

Zumbis

Aluno: Guilherme Luis Domingues

RA: 155619

Instituto de Computação
Universidade Estadual de Campinas

Campinas, 23 de Novembro de 2020.

Sumário

1	Backlog e número de conexões	2
2	Backlog de um socket TCP no kernel linux	2
3	Backlog de um socket TCP no kernel linux	2
4	Retirando processos zombies	5

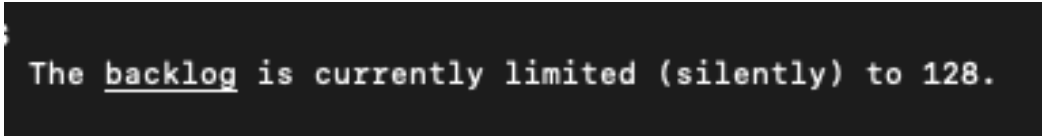
1 Backlog e número de conexões

Ao abrir uma conexão TCP e começar a escutar por requisições, um dos parâmetros configurados é o backlog. O backlog é o tamanho máximo das filas de conexões, isto é, quantas requisições poderão ficar aguardando, ao mesmo tempo, para serem executadas. Já o número de conexões é a quantidade de conexões abertas e estabelecidas ao servidor naquele momento.

Uma vez que o número do backlog do servidor foi atingido, ele começa a ignorar o SYN, enviado pelo cliente, não permitindo que novas conexões sejam estabelecidas.

2 Backlog de um socket TCP no kernel linux

Ao executar o comando 'man listen', obtivemos que o tamanho do backlog padrão para um socket TCP é de 128 conexões. A Figura 1 ilustra o resultado obtido.



```
The backlog is currently limited (silently) to 128.
```

Figura 1: Retorno do comando man listen, exibindo o valor padrão de 128 conexões no backlog

3 Backlog de um socket TCP no kernel linux

Para realizar o experimento, o script `execute_clients` foi desenvolvido.

Para que o resultado esteja de acordo, devemos rodar o servidor em um outro terminal, executando ele na porta 8000, utilizando o código abaixo

```
1 // Executando o servidor
2 gcc -Wall -o servidor servidor.c && ./servidor 8000
```

Uma vez o servidor executando, podemos então executar este script utilizando o comando `./execute_clients.sh`

```
1 #!/bin/bash
2 gcc -Wall -o cliente cliente.c
3 ./cliente 127.0.0.1 8000 &
4 ./cliente 127.0.0.1 8000 &
5 ./cliente 127.0.0.1 8000 &
6 ./cliente 127.0.0.1 8000 &
7 ./cliente 127.0.0.1 8000 &
8 ./cliente 127.0.0.1 8000 &
9 ./cliente 127.0.0.1 8000 &
10 ./cliente 127.0.0.1 8000 &
11 ./cliente 127.0.0.1 8000 &
12 ./cliente 127.0.0.1 8000 &
13 ./cliente 127.0.0.1 8000
```

Começando com o backlog igual à 0, que apenas uma conexão das 10 consegue ser feita executada, as restantes não chegam nem ser estabelecidas. A Figura 2 ilustra o resultado obtido.



The figure consists of two side-by-side terminal screenshots. The left terminal shows the server's output: it starts with a prompt, then runs the command to compile and execute the server. It then shows two 'Client' messages: 'Client 127.0.0.1:51553 disconnected' and 'Client 127.0.0.1:51554 disconnected', followed by a cursor. The right terminal shows the output of the script that runs 10 client processes. It shows a 'connect' message, followed by 'getpeername: Invalid argument', 'Server address: 0.0.0.0', 'Client socket: 127.0.0.1:51551', 'read error: Connection reset by peer', 'Server address: 127.0.0.1:8000', 'Client socket: 127.0.0.1:51553', and then 'dwp', 'sl', and 'sp'.

Figura 2: Verificando o backlog 0 com o comando netstat

Paralelamente, podemos utilizar o comando netstat para verificar as conexões estabelecidas e em espera.

```

Every 1.0s: sudo netstat -taulnp | grep 1024
tcp        2      0 127.0.0.1:1024          0.0.0.0:*               LISTEN      29970/./servidor
tcp        0      1 127.0.0.1:42024         127.0.0.1:1024          SYN_SENT    30027/./cliente
tcp        0      0 127.0.0.1:42022         127.0.0.1:1024          ESTABLISHED 30021/./cliente
tcp        0      0 127.0.0.1:1024         127.0.0.1:42020         ESTABLISHED -
tcp        0      0 127.0.0.1:1024         127.0.0.1:42022         ESTABLISHED -
tcp        0      0 127.0.0.1:42020        127.0.0.1:1024          ESTABLISHED 30020/./cliente

```

Figura 3: Executando o servidor com 0 conexões permitidas no backlog

Já com o backlog igual à 1 temos que dois cliente conseguem se conectar e serem executados. Os demais, no mesmo momento, recebem o erro de conexão recusada. A Figura 4 ilustra o resultado obtido.

```

code git:(main) * gcc -Wall -o servidor servidor.c && ./servidor 8000
Client 127.0.0.1:52544 disconnected
Client 127.0.0.1:52541 disconnected
Client 127.0.0.1:52543 disconnected
Client 127.0.0.1:52547 disconnected
Client 127.0.0.1:52542 disconnected
]

code git:(main) * ./execute_clientes.sh
connect: Connection reset by peer
connect: Connection reset by peer
Server address: 127.0.0.1:8000
Client socket: 127.0.0.1:52541
Server address: 127.0.0.1:8000
Client socket: 127.0.0.1:52544
dwp
dwp
sl
sl
sp
sp
getpeername: Invalid argument
Server address: 127.0.0.1:8000
Server address: 0.0.0.0:0
Client socket: 127.0.0.1:52543
Client socket: 127.0.0.1:52546
read error: Connection reset by peer
getpeername: Invalid argument
getpeername: Invalid argument
Server address: 127.0.0.1:8000
Client socket: 127.0.0.1:52547
Server address: 0.0.0.0:0
Server address: 0.0.0.0:0
Client socket: 127.0.0.1:52548
Client socket: 127.0.0.1:52549
read error: Connection reset by peer
read error: Connection reset by peer
dwp
dwp
sl
sl
sp
sp
getpeername: Invalid argument
Server address: 0.0.0.0:0
Server address: 127.0.0.1:8000
Client socket: 127.0.0.1:52545
Client socket: 127.0.0.1:52542
read error: Connection reset by peer

```

Figura 4: Resposta do servidor com 1 conexão no backlog

Com o backlog igual à 5 temos que seis dos cliente conseguem se conectar e serem executados ao mesmo tempo. O restante acaba sendo executado quando os demais encerram. A Figura 5 ilustra o resultado obtido.

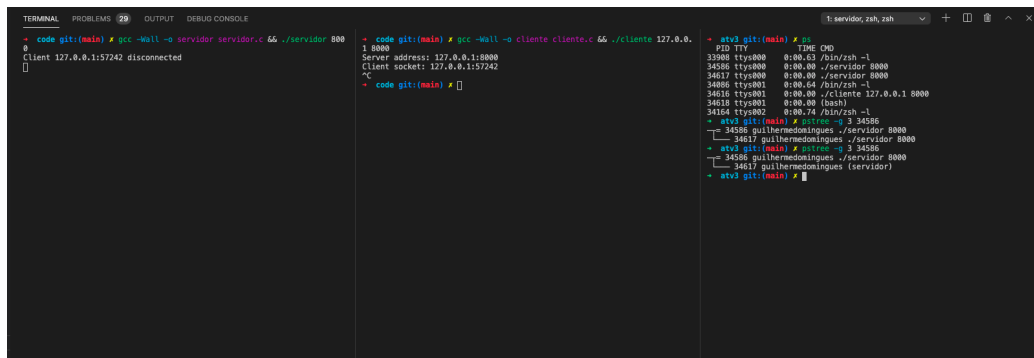
```
➤ code git:(main) x gcc -Wall -o servidor servidor.c && ./servidor 8000
Client 127.0.0.1:52702 disconnected
Client 127.0.0.1:52704 disconnected
Client 127.0.0.1:52705 disconnected
Client 127.0.0.1:52706 disconnected
Client 127.0.0.1:52708 disconnected
Client 127.0.0.1:52703 disconnected
Client 127.0.0.1:52704 disconnected
Client 127.0.0.1:52712 disconnected
Client 127.0.0.1:52709 disconnected
Client 127.0.0.1:52707 disconnected
Client 127.0.0.1:52710 disconnected
Client 127.0.0.1:52711 disconnected
[]

➤ code git:(main) x ./execute_clients.sh
Server address: 127.0.0.1:8000
Client socket: 127.0.0.1:52702
Server address: 127.0.0.1:8000
Client socket: 127.0.0.1:52703
Server address: 127.0.0.1:8000
Client socket: 127.0.0.1:52704
Server address: 127.0.0.1:8000
Client socket: 127.0.0.1:52705
Server address: 127.0.0.1:8000
Client socket: 127.0.0.1:52706
Server address: 127.0.0.1:8000
Client socket: 127.0.0.1:52708
dwp
dwp
dwp
dwp
dwp
sl
sl
sl
sl
sl
sl
sp
sp
sp
sp
sp
sp
sp
sp
sp
sp
➤ code git:(main) x Server address: 127.0.0.1:8000
Client socket: 127.0.0.1:52710
Server address: 127.0.0.1:8000
Server address: 127.0.0.1:8000
Client socket: 127.0.0.1:52709
Client socket: 127.0.0.1:52707
Server address: 127.0.0.1:8000
Server address: 127.0.0.1:8000
Client socket: 127.0.0.1:52711
Client socket: 127.0.0.1:52712
```

Figura 5: Resposta do servidor com 5 conexões no backlog

4 Retirando processos zombies

Sim, o código do jeito que foi implementado no último laboratório gera processos zombies, isto é, o processo criado a partir do fork não consegue ser mais acessado, ficando ativo no sistema. A Figura 6 ilustra o resultado de um novo cliente ter se conectado e morto, gerando um processo zombie. No terminal mais à direita todos os processos são listados e, mesmo após matar o processo do cliente, a árvore de processo ainda o mostra como ativo.



```
code git:(main) # gcc -Wall -o servidor servidor.c 66 ./servidor 800
1 8000
Server address: 127.0.0.1:8000
Client socket: 127.0.0.1:157242
Client 127.0.0.1:157242 disconnected
code git:(main) #

code git:(main) # gcc -Wall -o cliente cliente.c 66 ./cliente 127.0.0.1
code git:(main) #

atv3 git:(main) # ps
PID TTY TIME CMD
33908 tty$000 0:00.63 /bin/zsh -l
34586 tty$000 0:00.00 ./servidor 8000
34617 tty$000 0:00.00 ./servidor 8000
34886 tty$001 0:00.64 /bin/zsh -l
34616 tty$001 0:00.00 ./cliente 127.0.0.1 8000
34618 tty$001 0:00.00 (bash)
34664 tty$002 0:00.74 /bin/zsh -l
atv3 git:(main) # ps -o p 3 34586
34586 guilhermedomingues ./servidor 8000
34617 guilhermedomingues ./servidor 8000
atv3 git:(main) # ps -o p 3 34586
34586 guilhermedomingues ./servidor 8000
34617 guilhermedomingues (servidor)
atv3 git:(main) #
```

Figura 6: Evidenciando existência de processos zumbies

Para eliminar este problema, precisamos adicionar a função "Signal", que será chamada após o servidor começar a escutar o endereço. Além desta função, devemos implementar a rotina que receberá o sinal do filho, chamada sig_chld. Desta maneira, todos os processos filhos, gerados pelo processo do servidor, serão escutados e, caso encerrados, chamarão a função de sig_chld, para que o processo pai consiga definitivamente terminar com ele.

Com o servidor e cliente executando, listamos todos os processos e pegamos a árvore do processo pai (servidor). Com esse PID, listamos todos os processos que derivaram deste e, podemos ver que o cliente é o único. Após matar o cliente, repetimos novamente o comando para exibir a árvore de processo filho. Desta vez, não existe mais nenhum processo rodando além do servidor. A Figura 7 exemplifica isso.

```
TERMINAL PROBLEMS 29 OUTPUT DEBUG CONSOLE
+ code git:(main) # gcc -Wall -o servidor servidor.c 66 ./servidor 800
1 8000
Client 127.0.0.1:57196 disconnected
child 34425 terminated
[]

+ code git:(main) # gcc -Wall -o cliente cliente.c 66 ./cliente 127.0.0.1
Server address: 127.0.0.1:8000
Client socket: 127.0.0.1:57196
^C
+ code git:(main) # []

+ atv3 git:(main) # ps
PID TTY TIME CMD
33988 tty000 0:00.57 /bin/zsh -l
34269 tty000 0:00.00 ./servidor 8000
34425 tty000 0:00.00 ./servidor 8000
34886 tty001 0:00.53 /bin/zsh -l
34424 tty001 0:00.00 ./cliente 127.0.0.1 8000
34426 tty001 0:00.00 (bash)
34264 tty002 0:00.20 /bin/zsh -l
+ atv3 git:(main) # ps tree -g 3 34425
+ atv3 git:(main) # ps tree -g 3 34369
+ atv3 git:(main) # ps tree -g 3 34369
+ atv3 git:(main) # ps tree -g 3 34369
+ atv3 git:(main) #
```

Figura 7: Listando todos os processos após remoção dos processos zombies