Execução de Processos

Sistemas Operativos

Em nota terminal, verifica-se que unicamente resta ao sistema ser capaz de reproduzir os seguintes 3 pontos:

1. Listar tarefas em execução.
2. Terminar tarefa em execução.
3. Listar Historico de tarefas terminadas.

Sobre os quais será desenvolvida documentação adicional, estima-se que os restantes procedimentos não demorem mais do que um dia a ser feitos, assumindo que é seleccionado um algoritmo adequado para o funcionamento nestas temáticas. Temos portanto as seguintes especializações.

1 Listar tarefas em execução.

É interessante pois todos estes pontos se relacionam de um forma ou outra, assim que resolvermos este primeiro ponto penso que apenas restará adaptar a solução para as restantes temáticas.

Para listar todas as tarefas eme execução, precisamos, primeiramente, de um recipiente dessas tarefas. Dessa forma, penso que seria adequado utilizar uma tabela hash com closed adressing, pois no permitira testes de pertence em tempo constante, sem nos termos de preocupar com a temática de *linear probing*, subjacentes a tabelas de hash com open adressing. Para além disso, a temática do trabalho também não é isso. O ano passado tiraram 20 pessoas que usaram *arrays* para fazer procura em tempo linear. O importante é ter o resto a funcionar.

Por isso, assuma-se que já possuimos uma tabela de hash funcional, chamada `HashTable`, que tira partido da seguinte API:

```
HashTable hash_table_new(DestroyFunc free_func);
int hash_table_insert(HashTable table, unsigned long key, void* value);
int hash_table_contains(HashTable table, unsigned long key);
void* hash_table_find(HashTable table, unsigned long key);
unsigned long hash_table_size(HashTable table);
```

```
int hash_table_remove(HashTable table, unsigned long key);
void destroy_hash_table(HashTable table);
```

Então, já possuimos um *bucket* para toda a informação dos processos em execução. Quando começamos um novo processo inserimos este na tabela de hash. Então como fazemos para poder listar todos que estão em execução? se assumirmos essa tabela de hash como sendo o repósitorio de todas as tarefas a executar, ainda temos a capacidade de simplesmente fazer uma travessia e obter de imediato quais tarefas estão a ser executas.

É um bocado mais que linear, porém acho que um *overenginering* disto vai dar mais dores que cabeça do que qualquer coisa. Assume-se portanto que temos uma hash tabela, que associa a cada *key* o nome da tarefa que está a executar, sendo que o resultado tem de ser apresentado da seguinte forma:

```
#1: ./a.out | teste
#3: date
#5: cut -f7 -d: /etc/passwd | uniq | wc -l
```

Para tal, pode ser útil definir uma função que permita percorrer todos os pares (chave, valor) da tabela de hash, o que poderia ser alguma coisa no ramo de:

```
void hash_table_foreach(HashTable table,
                  void (*feach) (unsigned long, void *));
```

Sendo que para imprimir todas as listas de tarefa em execução bastaria fazer uma função *for each* adequada, o que poderia ser alguma coisa na ordem de:

```
void foreach_task(unsigned long key, void * value) {
  char *buff, *str = (char*)value;
  ssize_t n = asprintf("#%ld: %s\n", key, str);

  if(n && write(pipe_writer, buff, n) == -1)
    throw_error(2, "Erro na escrita de for each.");
}
```

Penso que pode funcionar, falta agora perceber de que forma é que vamos inserir e retirar elementos desta *hash table*, começando pelo fim, podemos verificar o seguinte excerto de código.

Não podemos simplesmente adicionar elementos na hash table atraves de sinais, pois não dá para passar informação pelos processos pais, uma coisa que podemos fazer é ter um pipe sem nome no servidor principal, constantemente a correr, e que simplesmente vai recolhendo os diversos elementos que, com sucesso foram processados. Então, depois de temos a garantia de que conseguimos processar esse pedido, basta-nos escrever no pipe o id da tarefa e o respectivo comando que foi executando. O processo que está no pai vai receber este, processar e adicionar na tabela de hash.

Mas então, como remover da tabela de hash? Isto é certamente um problema, e muito chato. Novamente, acho que precisamos de criar um outro pipe, mas

desta vez o pipe continuamente lê as entradas e remove da tabela de hash.

Mas se considerarmos isso, existe o potencial de removermos um elemento que não está na tabela, mas que pelas diversas razões ainda está a ser processado. O pedido ia ser removido, não há nenhum pedido com aquele id naquela altura, por isso não faria diferença, de seguida o novo pedido ia ser adicionado. Então, supostamente um pedido que já devia ter sido removido terá sido inserido na tabela, apesar de invalidamente.

Para dar a volta isso, podemos fazer o seguinte:

1. Inserimos todos os pedidos na tabela, independentemente de terem acabado ou não.
2. Quando o processamento tarefa terminar, seja porque razão for, então é emitido um id para um outro pipe anónimo, que vai levar à remoção da entrada da tabela de hash.

Assim, temos a garantir de que todos os elementos só são removidos da tabela de hash depois de terem sido devidamente processados.

Então até agora o que é que temos feito?

- Inserir processos em execução.
- Remover processos em execução.
- Listar tarefas em execução.

Falta apenas lidar com as duas seguintes temáticas: Terminar tarefa em execução, listar registo histórico de tarefas. Começando pela última, temos que.

2 Listar registo histórico de tarefas terminadas.

Para este efeito, é necessário, primeiro de tudo, saber de que forma é que um dado processo terminou, sendo que pode haver várias formas, as quais descrevemos como:

```
enum Command {
  COMMAND_ERROR,
  COMMAND_SUCESS,
  COMMAND_TERMINATED,
  COMMAND_PIPE_TIMEOUT,
  COMMAND_EXEC_TIMEOUT,
};
```

Com as seguintes descrições respectivas:

1. `COMMAND_ERROR`: Uma tarefa, ou comando, pode simplesmente ter terminado devido a um error de qualquer tipo, ou na escrita, ou porque

o programa é inválido, as razões são inumeras, por isso é preciso um identificador só para este efeito.

2. `COMMAND_SUCESS`: Se tudo correr bem, este identificador é enviado, define-se "tudo correr bem" como sendo receber um código sem erro do respectivo wait.
3. `COMMAND_TERMINATED`: Usando o comando `terminate` é possivel matar uma das tarefas em execução, sendo que está deverá ficar descrita como morta.
4. `COMMAND_PIPE_TIMEOUT`: Um comando pode ter terminado porque um dos pipes demorou demasiado tempo a executar, sendo que isto representa inatividade do ponto de vista dos pipes.
5. `COMMAND_EXEC_TIMEOUT`: Semelhante ao exemplo anterior, ocorre se um pedido demorar demasiado tempo a processar o pedido.

Existe três coisas que devem ser feitas quando uma tarefa termina, seja porque razão for:

1. Remover da tabela de hash,
2. Qual foi o tipo de terminação?
3. Escrever, via `LogManager`, o resultado, da seguinte forma:

   `#[ID PEDIDO], [TIPO TERMINACAO]: [COMANDO EXECUTADO]`

Por isso, intuitivamente, dá para perceber que o comando `historico`, na verdade, corresponde só a pedir ao `LogManager` que faça dump do ficheiro de historicos no *file descriptor* indicado, que neste caso será o `pipe_writer`.

Esta função poderá ser alguma coisa do genero de:

```
ssize_t dump_task_history(int fd) {
  char buffer[MAX_BUFFER_SIZE];
  // Coloca fd no inicio do ficheiro de historico
  // Isto porque, temos que imaginar, que tambem
  // existe um processo que escreve neste.
  lseek(histd, 0, SEEK_SET);

  while( (nread = read(histd, buffer, MAX_BUFFER_SIZE) ) > 0 ) {
    if( write(fd, buffer, nread) == -1) {
      // Em caso de erro, voltar a por o fd no fim
      // do ficheiro.
      lseek(histd, 0, SEEK_END);
      return -1;
    }
  }
```

```
  // Aqui o ficheiro ja esta no EOF.
}
```

A cada vez que é preciso fornecer o output, então simplesmente abre o ficheiro de historico, lê e faz dump simultaneamente.

O cerne da questão reside então, necessariamente, em estabelecer de que forma é que uma tarefa é terminada e o seu respectivo código de terminação acomodado.

Notoriamente, não será preciso armazenar nenhum código de terminação, isto porque não interessa ter está informação em memória, por isso podemos imediatamente a colocar em disco, em principio da seguinte forma:

```
ssize_t append_task_info(unsigned long key, char *task,
                    enum Command term) {
  // Pelo codigo acima, o histd esta sempre no final do
  // ficheiro, logo, e so escrever esta informação, consoante o erro.
  // Primeiro precisamos de encontrar o tipo.
  // Assume que existe um array que armazena as mensagens associadas
  // a cada um dos erros.
  char *buff;
  ssize_t n = asprintf(&buff, "#%ld, %s: %s\n", key, com_msg[term], task);

  if( write(histd, buff, n) == -1) {
      throw_error(2, ... );
      return -1;
  }

  return n;
}
```

Então, parte-se do pressuposto que qualquer tipo de terminação que seja efetuada vai fazer com que o sistema tenha de fazer uma chamada à função `append_task_info`. Então, vamos analisar o caso em que o cliente solicita a terminação. Para conseguirmos ganhar *insight* sobre o funcionamento dos restantes casos.

3 Terminação de Execução

O que é que deve acontecer quando um cliente pede a terminação de uma tarefa? automaticamente, deve ser feito alguma coisa que possibilite a terminação imediata, ou assim que possivel, do processo que está a processar aquela exata thread. Penso que isto poderá ser feito recorrendo a um SIGKILL sobre aquele processo. Esta definição parte do pressuposto que, algures, possuimos o identificador do processo que está responsável por tratar da tarefa necessária, então, por esta razão, poderemos vir a ter a necessidade de, em paralelo com a string do comando,

armazenarmos o pid que está a processar o pedido.

Isto não é de todo dificil, basta para tal criar uma struct que armazene esta informação devidamente, ou seja, o pid_t e a string associada, sendo que a tabela de hash já ira possuir como chave o respectivo id da tarefa. Então iremos necessitar da seguinte struct:

```
typedef struct taskinfo {
  pid_t care_taker;
  char *command;
}*Taskinfo;
```

Isto implica ligeiras alterações na função *for each* definida acima, ora, mantendo o invariante de que só estão na tabela de hash aqueles processos que ainda não estão terminados, então, para forçar a terminação de um processo especifico, seria suficiente aceder diretamente à entrada, caso exista, daquele pedido na tabela de hash, e se este existir, então é enviado um SIGKILL para àquela thread em especifico. Futuramente, isto poderá ser testado se definirmos pedirmos ao sistema para executar o comando `sleep 20 ; cat /etc/passwd`, caso no qual se deverá verificar, que se o pedido for morto antes dos 20 segundos, então não deverá haver *output* resultante.

Na prática, usufruindo de pseudo-código, isto poderia ser feito da seguinte forma:

```
void process_kill_task(char** argv) {
  char * str = "";
    unsigned long id = strtoul(argv[0],&str,0);

  TaskInfo info = hash_table_find(table, id);

  if(info == NULL) {
    // Aquele id nao esta em execucao,
    // envia msg a notificar
    n = asprintf(&str,
          "A tarefa %ld nao esta em execucao.\n", id);

    if( write(pipe_writer, str, n) == -1 )
      throw_error(2, "Erro inesperado na escrita.");
  }
  else {
    // Aquele pedido está contido na hash
    // table, deve por isso ser morto e removido.
    kill(info->care_taker, SIGKILL);
    hash_table_remove(table, id);

    // Para alem disso, e tambem preciso
    // inserir nos logs a causa da sua morte.
    append_task_info(id, info->command,
```

```
                        COMMAND_TERMINATED);
  }
}
```

Com isto, e em teoria, temos agora a capacidade de matar qualquer tarefa que esteja a ser executada. Falta-nos portanto tocar nos seguintes: `COMMAND_SUCESS`, `COMMAND_EXEC_TIMEOUT`, `COMMAND_PIPE_TIMEOUT`.

4 Sucesso ou Erro

Vamos assumir que temos uma função processadora do comando chamada `proc_line`, e que esta mesma função retorna 0 caso tenha terminado com sucesso, qlqr outro numero caso contrário. Então, para adicionar este informação basta-nos fazer o seguinte.

```
...
enum Command c = COMMAND_SUCESS;
if(!fork()) {
  redir_log_file(1);

  pid = fork();

  if(!pid) {
    status = process_line(argv[0]);
    _exit(status);
  }
  else {
    // e preciso verificar erros notificar
    // o cliente
    // notifica_cliente(coisas);

    wait(&status);

    if(status != 0) {
      // Aconteceu erros algures
      c = COMMAND_ERROR;
    }

    append_task_info(id, argv[0], c);

    kill(getppid(), SIGUSR1);
  }

  _exit(0);
}
```

...

Porém, falta ainda remover a entrada na tabela de hash. Não podemos, como fizemos com o processamento do sinal de terminação, aceder diretamente à tabela de hash, pois apesar de esta ser global, não é partilhada com os processos filhos! Precisamos então de outro metódo de IPC, por isso um meio adequado seria ter um pipe anonimo reservado especificamente para estes problemas. Chamemos a esse pipe `remove_pipe`. Então, na ponta `remove_pipe[1]` temos a ponta de escrita, e em `remove_pipe[0]` a respectiva ponta de leitura. Vamos assumir que este pipe está sempre à escuta, e que a cada vez que recebe uma nova mensagem, sabe de imediato que essa mensagem possui um tamanho de `sizeof(unsigned long)`, e sabe também, tal como é seu proposito, que deve proceder à remoção deste elemento da tabela de hash. Porém, isto também não resolve o problema, porque se temos um pipe constamente a ler, então, vamos presicar que este comportamento seja encapsular num fork do processo do servidor, ora, isto ao seu verificar implica que alterações na hash_table também não se iam notar no processo pai.

Então, o IPC escolhe terá de ser, obrigatoriamente, um signal. então fariamos qualquer coisa na forma de:

```
kill(getppid(), SIGUSR2);
```

Isto implica que iriamos precisa de um `signalhandler` correspondente, porém, isto só é tão útil como a nossa capacidade de transmitir valores entre o filho e o pai, neste caso, pretende-se unicamente comunicar qual o id do processo que acabou de sair da tabela de hash. Obviamente, o `kill` não permite a transmissão deste tipo de conhecimento, uma FIFO iria levar a problemas do mesmo genero aos mencionados no pipes anonimos.

Uma solução viável é a combinação destes dois, ou seja, enviamos o `SIGUSR2`, o `signalhandler` correspondente abre o pipe para leitura, enquanto que o processo filho abre o pipe para escrita. O processo filho envia o id da tarefa que terminou, o `signalhandler` lê este valor e atualiza devidamente a tabela de hash. Acho que esta solução, apesar de inicialmente robusta, é a mais viável para o tipo de comunicação desejada. Então ficariamos com um handler da forma de:

```
void process_termination(int signum) {
  unsigned long id;

  // Fecha ponta de escrita.
  close(terminate_pipe[0]);

  if( read(terminate_pipe[1], &id,
        sizeof(unsigned long)) == -1 )
        throw_error(2, "error ao ler pipe.");

  hash_table_remove(table, id);
```

```
  // Fecha a ponta de leitura
  close(terminate_pipe[1]);
}
```

Enquanto que o procesos que executa esta tarefa estaria, obrigatoriamente na necessidade de efetuar o seguinte:

```
if(!fork()) {
  redir_log_file(1);

  pid = fork();

  if(!pid) {
    status = process_line(argv[0]);
    _exit(status);
  }
  else {
    // e preciso verificar erros notificar
    // o cliente
    // notifica_cliente(coisas);

    wait(&status);

    if(status != 0) {
      // Aconteceu erros algures
      c = COMMAND_ERROR;
    }

    append_task_info(id, argv[0], c);

    kill(getppid(), SIGUSR1);
    kill(getppid(), SIGUSR2);

    // Fecha ponta de leitura
    close(terminate_pipe[1]);

    if( write(terminate_pipe[0], &id,
            sizeof(unsigned long)) == -1 )
      throw_error(2, "Error na escrita.");

    // Fecha ponta de escrita
    close(terminate_pipe[0]);
  }

  _exit(0);
}
```

Este mecanismo visa emular o seguinte comportamento: O filho envia um sinal ao pai assim que sabe que pode remover uma tarefa da tabela, o pai recebe o sinal e fica atento ao pipe anonimo à espera do id da tarefa que tem de remover, o filho envia este id via o pipe e o pai processa-o devidamente.

5 Resumo

Quando começamos, faltavam as seguintes componentes:

1. Listar tarefas em execução.
2. Terminar tarefa em execução.
3. Listar Historico de tarefas terminadas.

Sendo que agora todas as estas se verificam, sendo a elas associado o respectivo código de saida do processo. Falta unicamente estudar de que forma deverá ser feito o processamento dos código `COMMAND_PIPE_TIMEOUT` e `COMMAND_EXEC_TIMEOUT`. Porém, estes só serão pensados quando uma implementação do presente documento for feita. Para evitar acumular demasiados bugs.