SISTEMAS OPERACIONAIS

SVCs para Controle de Processos (cont)

Término de Processos no Unix

- Um processo pode terminar normalmente ou anormalmente nas seguintes condições:
- Normal:
  - Executa `return` na função `main()`, o que é equivalente à chamar `exit();`
  - Invoca diretamente a função `exit()` da biblioteca C;
  - Invoca diretamente o serviço do sistema `_exit()`.
- Anormal:
  - Invoca o função `abort();`
  - Recebe sinais de terminação gerados pelo próprio processo, ou por outro processo,
  - ou ainda pelo Sistema Operacional.
- A função abort()
  - Destina-se a terminar o processo em condições de erro e pertence à biblioteca padrão do C.
  - Em Unix, a função `abort()` envia ao próprio processo o sinal `SIGABRT`, que tem como consequência terminar o processo. Esta terminação deve tentar fechar todos os arquivos abertos.

A chamada `exit()`

- `void exit (code)`
  - O argumento code é um número de 0 a 255, escolhido pela aplicação e que será passado para o processo pai na variável `status`.
- A chamada `exit()` termina o processo; portanto, `exit()`
- nunca retorna
  - Chama todos os *exit handlers* que foram registrados na função `atexit()`.
  - A memória alocada ao segmento físico de dados é liberada.
  - Todos os arquivos abertos são fechados.
  - É enviado um sinal para o pai do processo. Se este estiver bloqueado esperando o filho, ele é acordado.
  - Se o processo que invocou o `exit()` tiver filhos, esses serão "adotados" pelo processo *init*.
  - Faz o escalonador ser invocado.

As chamadas `wait()` e `waitpid()`

- São usadas para esperar por mudanças de estado nos filhos do processo chamador e obter informações sobre aqueles filhos cujos estados tenham sido alterados.
  - Ex: quando um processo termina (executando → terminado) o kernel notifica o seu pai enviando-lhe o sinal SIGCHLD.
- Considera-se uma alteração de estado:
  - o término de execução de um filho (exit);
  - o filho foi parado devido a um sinal (CTRL-z);
  - o filho retornou à execução devido a um sinal (SIGCONT).
- Se o filho já teve o seu estado alterado no momento da chamada, elas retornam imediatamente; caso contrário, o processo chamador é bloqueado até que ocorra uma mudança de estado do filho ou então um "*signal handler*" interrompa a chamada.

As chamadas `wait()` e `waitpid()` (cont.)

- Um processo pode esperar que seu filho termine e, então, aceitar o seu código de terminação, executando uma das seguintes funções:
  - **wait(int *status)**: suspende a execução do processo até a morte de seu filho. Se o filho já estiver morto no instante da chamada da primitiva (caso de um processo zumbi), a função retorna imediatamente.
  - **waitpid(pid_t pid, int *status, int options)**: suspende a execução do processo até que o filho especificado pelo argumento pid tenha morrido. Se ele já estiver morto no momento da chamada, o comportamento é idêntico ao descrito anteriormente.

```
#include <sys/wait.h>

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

As chamadas `wait()` e `waitpid()` (cont.)

- Em resumo, um processo que invoque `wait()` ou `waitpid()` pode:
  - bloquear - se nenhum dos seus filhos ainda não tiver terminado;
  - retornar imediatamente com o código de terminação de um filho - se um filho tiver terminado e estiver à espera de retornar o seu código de terminação (filho zombie).
  - retornar imediatamente com um erro - se não tiver filhos.

- Se `wait()` ou `waitpid()` retornam devido ao status de um filho ter sido reportado, então elas retornam o PID daquele filho.

- Se um erro ocorre (ex: se o processo não existe, se o processo especificado não for filho do processo que o invocou, se o grupo de processos não existe), as funções retornam -1 e setam a variável global errno.

- Os erros mandatórios para `wait()` e `waitpid()` são:
  - `ECHILD`: não existem filhos para terminar (wait), ou pid não existe (waitpid)
  - `EINTR`: função foi interrompida por um sinal
  - `EINVAL`: o parâmetro options do waitpid estava inválido

As chamadas `wait()` e `waitpid()` (cont.)

- Diferenças entre `wait()` e `waitpid()`:
  - `wait()` bloqueia o processo que o invoca até que um filho qualquer termine (o primeiro filho a terminar desbloqueia o processo pai);
  - `waitpid()` não espera que o 1o filho termine, tem um argumento para indicar o processo pelo qual se quer esperar.
  - `waitpid()` tem uma opção que impede o bloqueio do processo chamador (útil quando se quer apenas obter o código de terminação do filho);

As chamadas `wait()` e `waitpid()` (cont.)

- O argumento pid de `waitpid()` pode ser:
  - >0: espera pelo filho com o pid indicado;
  - -1: espera por um filho qualquer (= `wait()`);
  - 0: espera por um filho qualquer do mesmo process group
  - <-1 : espera por um filho qualquer cujo process group ID seja igual a |*pid*|.

- `waitpid()` retorna um erro (valor de retorno = -1) se:
  - o processo especificado não existir;
  - o processo especificado não for filho do processo que o invocou;
  - o grupo de processos não existir.

Valores de status

- O argumento status de `waitpid()` pode ser `NULL` ou apontar para um inteiro. No caso de status ser ≠ `NULL`, o código de terminação do processo que finalizou é guardado na posição indicada por status. No caso de ser = `NULL`, este código de terminação é ignorado.

- A morte do processo pode ser devido a:
  - uma chamada `exit()` e, neste caso, o byte à direita de status vale 0 e o byte à esquerda é o parâmetro passado a `exit()` pelo filho;
  - uma recepção de um sinal fatal e, e neste caso, o byte à direita de status é não nulo e os sete primeiros bits deste byte contém o número do sinal que matou o filho.

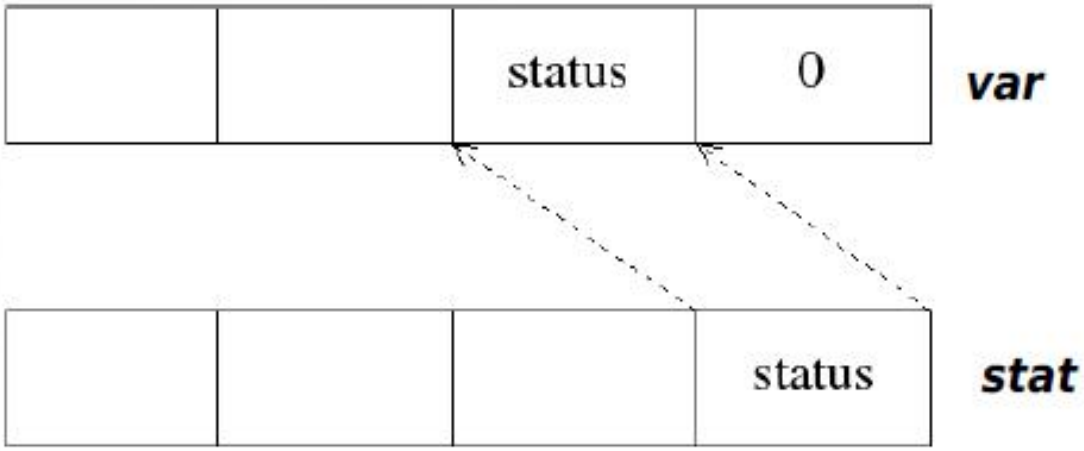
- O estado do processo filho retornado por status tem certos bits que indicam se a terminação foi normal, o número de um sinal, se a terminação foi anormal, ou ainda se foi gerado um core file.

- O estado de terminação pode ser examinado (os bits podem ser testados) usando macros, definidas em `<sys/wait.h>`. Os nomes destas macros começam por WIF e podem ser são listadas com o comando shell *man 2 wait*.

Valores de status (cont)

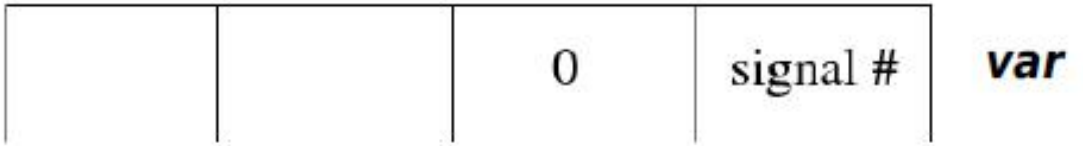
No caso de um ••••••••••,

••••••••••



No caso de um Sinal,

••••••••••



Valores de status

- O POSIX expecifica seis macros, projetadas para operarem em pares:
  - **WIFEXITED(status)** – permite determinar se o processo filho terminou normalmente. Se WIFEXITED avalia um valor não zero, o filho terminou normalmente. Neste caso, WEXITSTATUS avalia os 8-bits de menor ordem retornados pelo filho através de _exit(), exit() ou return de main.
  - **WEXITSTATUS(status)** – retorna o código de saída do processo filho.
  - **WIFSIGNALED(status)** – permite determinar se o processo filho terminou devido a um sinal
  - **WTERMSIG(status)** – permite obter o número do sinal que provocou a finalização do processo filho
  - **WIFSTOPPED(status)** – permite determinar se o processo filho que provocou o retorno se encontra congelado (stopped)
  - **WSTOPSIG(status)** – permite obter o número do sinal que provocou o congelamento do processo filho

Valores de status (cont)

- Estrutura Geral

```
q = wait(&status);

if (q == -1) {
    /* Erro */
} else if (q > 0) {
    /* q -> pid do processo que terminou */

    if (WIFEXITED(status)) {
        /* Processo q terminou normalmente */
        /* Codigo de saida = WEXITSTATUS(status) */
    } else {
        /* Processo q terminou anormalmente! */
    }
}
```

E se o processo pai receber um sinal?

- Solução para que um processo pai continue esperando pelo término de um processo filho, mesmo que o pai seja interrompido por um sinal:

```
#include <errno.h>
#include <sys/wait.h>

pid_t r_wait(int *stat_loc) {
    int retval;

    while (((retval = wait(stat_loc)) == -1) && (errno == EINTR)) ;
    return retval;
}
```

Como usar wait sem bloquear?

- A opção WNOHANG na chamada *waitpid* permite que um processo pai verifique se um filho terminou, sem que o pai bloqueie caso o status do filho ainda não tenha sido reportado (ex: o filho não tenha terminado)
  - Neste caso *waitpid* retorna 0

```
pid_t child pid;

while (childpid = waitpid(-1, NULL, WNOHANG))
   if ((childpid == -1) && (errno != EINTR))
      break;
```

Exemplo 1: Fan wait (testa_wait_1.c)

- O programa e lançado em background e, apos o segundo filho estiver bloqueado num laco infinito, um sina será lançado para interromper sua execução através do comando shell
  - testa_wait_1 &
  - kill -8 29081

- Após a criação dos filhos, o processo pai ficará bloqueado na espera de que estes morram. O primeiro filho morre pela chamada de um `exit()`, sendo que o parâmetro de `wait()` irá conter, no seu byte esquerdo, o parâmetro passado ao `exit()`; neste caso, este parâmetro tem valor 7.

- O segundo filho morre com a recepção de um sinal, o parâmetro da primitiva `wait()` ira conter, nos seus 7 primeiros bits, o numero do sinal (no exemplo anterior ele vale 8).

Exemplo 2: wait e init (testa_wait_2.c)

- O programa é lançado em background.
  - testa_wait_2 &

- Primeiro, rode normalmente o programa. Verifique que o pai sai do wait e é concluído assim que um dos filhos termina.
- Na segunda vez, rode o programa matando o primeiro filho logo depois que o Filho2 for dormir.
  - Verifique que agora o pai sai do `wait()`, terminando antes do Filho2. Verifique que Filho2 foi adotado pelo init.

Exemplo 3: wait all child (testa_wait_3.c)

- Pai espera por todos os filhos
  - testa_wait_3 <número de processos>

Exemplo 4: chain wait (testa_wait_4.c)

- Cada filho criado espera por seu próprio filho completar antes de imprimir a msg.
- As mensagens aparecem na ordem reversa da criação.
  - testa_wait_4 <número de processos>

Exemplo 5: macros (testa_wait_5.c)

- Determina o status de exit de um processo filho

Referências

- Slides adaptados de Roberta Lima Gomes (UFES)

- Bibliografia
  - Kay A. Robbins, Steven Robbins, *UNIX systems programming: communication, concurrency, and threads*. Prentice Hall Professional, 2003 - 893 pages - Capítulo 3