

SVCs para Controle de Processos no Unix (cont.)

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.

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.

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.)

- 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()

- 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.)

- 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 ID*
 - < -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 \neq `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, 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`.

LPRM/DI/UFES

Sistemas Operacionais

9

Valores de status

- O POSIX especifica 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
  
```

- Linux:
 - **WIFCONTINUED(status)** (Linux 2.6.10)

LPRM/DI/UFES

11

Sistemas Operacionais

Valores de status

(cont.)

No caso de um `exit(stat)`,

	status	0	var
--	--------	---	------------

`wait(var)`

		status	stat
--	--	--------	-------------

No caso de um Sinal,

	0	signal #	var
--	---	----------	------------

`wait(var)`

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 */
        /* Código de saída = WEXITSTATUS(status) */
    } else {
        /* Processo q terminou anormalmente! */
    }
}
  
```

LPRM/DI/UFES

12

Sistemas Operacionais

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;
}
```

LPRM/DI/UFES

13

Sistemas Operacionais

```
// O programa é lançado em background. Após o segundo filho ser bloqueado no laço infinito, um
// sinal é lançado para interromper a sua execução, por meio de comando shell
//"kill -número-do-sinal <pid-filho2>"
#include <errno.h>
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main() {
```

Exemplo 1: Process fan wait (esta_wait_1.c)

```
    int pid ;
    printf("\nBom dia, eu me apresento. Sou o processo %d.\n", getpid()) ;
    printf("Estou sentindo uma coisa crescendo dentro de minha barriga...");
    printf("sera um filho?!?\n");

    if (fork() == 0) {
        printf("\tOii, eu sou %d, o filho de %d.\n",getpid(),getppid());
        sleep(30) ;
        printf("\tEu sou tao jovem, e ja me sinto tao fraco!\n");
        printf("\tAh nao... Chegou minha hora!\n");
        exit(7) ;
    }

    int ret1, status1 ;
    printf("You esperar que este mal-estar desapareca.\n");
    ret1 = wait(&status1) ;
    if ((status1&255) == 0) {
        printf("valor de retorno do wait(): %d\n",ret1) ;
        printf("Parametro de exit(): %d\n", (status1>>8)) ;
        printf("Meu filho morreu por causa de um simples exit.\n");
    }
    else
        printf("Meu filho nao foi morto por um exit.\n");
    printf("\nSou eu ainda, o processo %d.", getpid());
    printf("\nOh nao, recomeceu! Minha barriga esta crescendo de novo!\n");
}
```

15

LPRM/DI/UFES

16

Sistemas Operacionais

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;
```

LPRM/DI/UFES

14

Sistemas Operacionais

```
    if ((pid=fork()) == 0) {
        printf("\tAlo, eu sou o processo %d, o segundo filho de %d\n",
            getpid(),getppid());
        sleep(3) ;
        printf("\tEu nao quero seguir o exemplo de meu irmao!\n");
        printf("\tNao vou morrer jovem e vou ficar num loop infinito!\n");
        for(;;) ;
    }
    else {
        int ret2, status2, s ;
        printf("Este aqui tambem vai ter que morrer.\n");
        ret2 = wait(&status2) ;
        if ((status2&255) == 0) {
            printf("O filho foi morto por um sinal\n");
        }
        else {
            printf("Valor de retorno do wait(): %d\n",ret2) ;
            s = status2&255 ;
            printf("O sinal assassino que matou meu filho foi:
                %d\n",s) ;
        }
    }
    exit(0);
}
```

Sistemas Operacionais

Exemplo 2: wait e init (testa_wait_2.c)

```
/* O programa é lançado em background.
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. */
#include <errno.h>
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
int main() {
    int pid;
    printf("\n01, eu sou o pai PID = %d. Vou criar um filho.\n", getpid());
    if ((pid=fork()) == 0) {
        printf("\t01, eu sou o Filho1, PID = %d, PPID = %d.\n", getpid(), getppid());
        for(;;);
    }
    else {
        printf("\t01, sou eu, o pai, de novo. Vou criar mais um filho e depois vou entrar em
        wait(NULL);
        if ((pid=fork()) > 0)
            wait(NULL);
        else {
            printf("\t01, eu sou Filho2, PID = %d, PPID = %d.\n", getpid(), getppid());
            sleep(60);
            printf("\t01, sou o Filho2. Acordei mas estou terminando agora. Use ps -l
            novamente.\n");
        }
    }
}
```

15

LPRM/DI/UFES

18

Sistemas Operacionais



Exemplo 4: r_wait

```
/* Para rodar o programa em background: $test_wait_4 <número de processos> &
/* Pai espera todos os filhos terminarem, mesmo se um deles for morto durante o sleep().
/* Usa a função r_wait() para esperar por todos os filhos.
/* Observar a diferença entre EINTR e ECHILD (o processo pai fica esperando eternamente).
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <errno.h>
pid_t r_wait(int *status) {
    int retval;
    while (((retval = wait(status)) == -1) && (errno == EINTR));
    // while (((retval = wait(status)) == -1) && (errno == ECHILD));
    return retval;
}
```

LPRM/DI/UFES

19

Sistemas Operacionais

Exemplo 3: wait all children

```
/* Para rodar o programa: $testa_wait_3 <número de processos>
/* Pai espera por todos os filhos - Exercise 3.20
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
int main (int argc, char *argv[]) {
    pid_t childpid = 0;
    int i, n;
    if (argc != 2) { // check number of command-line arguments
        fprintf(stderr, "Usage: %s n\n", argv[0]);
        return 1;
    }
    n = atoi(argv[1]);
    for (i = 1; i < n; i++)
        if ((childpid = fork()) <= 0) //only the child (or error) enters
            break;
    for ( ; ; ) {
        childpid = wait(NULL);
        if ((childpid == -1) && (errno != EINTR))
            break;
    }
    fprintf(stderr, "I am process %ld, my parent is %ld\n", (long)getpid(),
    (long)getppid());
    return 0;
}
```

LPRM/DI/UFES

18

Sistemas Operacionais



Exemplo 4: r_wait

```
int main (int argc, char *argv[]) {
    pid_t childpid = 0;
    int i, n;
    if (argc != 2) { // check for valid number of command-line arguments
        fprintf(stderr, "Usage: %s n\n", argv[0]);
        return 1;
    }
    n = atoi(argv[1]);
    for (i = 1; i < n; i++)
        if ((childpid = fork()) <= 0) { //only the child (or error) enters
            sleep(10); break;
        }
    while (r_wait(NULL) > 0) ; // wait for all of your children
    fprintf(stderr, "i:%d process ID:%ld parent ID:%ld child ID:%ld\n",
    i, (long)getpid(), (long)getppid(), (long)childpid);
    return 0;
}
```

LPRM/DI/UFES

20

Sistemas Operacionais

Exemplo 5: r_wait (testa_wait_5.c - exemplo 3.15)

```

/* Determina o status de exit de um processo filho - TEM ERRO - FALTA ACERTAR!!! */
#include <errno.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
pid_t r_wait(int *status) {
    int retval;
    while (((retval = wait(status)) == -1) && (errno == EINTR)) ;
    return retval;
}
int main(void) {
    pid_t pid;
    int status;
    if ((pid = fork()) < 0)
        fprintf(stderr, "fork error\n");
    else if (pid == 0) exit(7);
    /* child1 finishes normally */
    if (wait(&status) != pid)
        fprintf(stderr, "wait error\n");
    pr_exit(status);
    /* parent code */
    if (wait(status) != pid)
        fprintf(stderr, "wait error\n");
    /* wait for child and print its status */
    if ((pid = fork()) < 0)
        fprintf(stderr, "fork error\n");
    else if (pid == 0) abort();
    /* child2 generates SIGABRT */
    if (wait(&status) != pid)
        fprintf(stderr, "fork error\n");
    pr_exit(status);
}

```

LPRM/DI/UFES

21

Sistemas Operacionais

Exemplo 6: Process chain wait (testa_wait_6.c - exemplo 3.21)

```

// Para rodar o programa: $testa_wait_6 <número de processos>
// Cada filho criado espera por seu próprio filho completar antes de imprimir a msg.
// AS mensagens aparecem na ordem reversa da criação.
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
int main (int argc, char *argv[]) {
    pid_t childpid;
    int i, n;
    pid_t waitreturn;
    if (argc != 2){
        fprintf(stderr, "Usage: %s processes\n", argv[0]);
        return 1;
    }
    n = atoi(argv[1]);
    for (i = 1; i < n; i++)
        if (childpid = fork()) break;
    while (childpid != (waitreturn = wait(NULL)))
        if ((waitreturn == -1) && (errno != EINTR))
            break;
    fprintf(stderr, "I am process %ld, my parent is %ld\n", (long) getpid(), (long) getppid());
    return 0;
}

```

Exemplo 5: r_wait (testa_wait_5.c - exemplo 3.15)

```

if ((pid = fork()) < 0)
    printf(stderr, "fork error\n");
else if (pid == 0) status /= 0;
/* child3 - divide by 0 generates SIGFPE */

if (wait(&status) != pid)
    fprintf(stderr, "wait error\n");
pr_exit(status);
/* wait for child and print its status */

if ((pid = fork()) < 0)
    printf(stderr, "fork error\n");
else if (pid == 0)
    sleep(30);
/* child4 - waiting SIGSTOP */

if (wait(&status) != pid)
    fprintf(stderr, "wait error\n");
pr_exit(status);
/* wait for child and print its status */
exit();
}
void show_return_status(void) {
    pid_t childpid;
    int status;
    childpid = r_wait(&status);
    if (childpid == -1)
        perror("Failed to wait for child");
    else if (WIFEXITED(status))
        printf("normal termination, exit status = %d\n", WEXITSTATUS(status));
    else if (WIFSIGNALED(status))
        printf("abnormal termination, signal number = %d\n", WTERMSIG(status),
            else if (WIFSTOPPED(status))
                printf("child stopped, signal number = %d\n", WSTOPSIG(status));
}
}

```

LFS

15

<http://www.inf.ufes.br/~rgomes/so.htm>

Referências

- Kay A. Robbins, Steven Robbins, *UNIX Systems Programming: Communication, Concurrency and Threads, 2nd Edition*
 - Capítulo 3