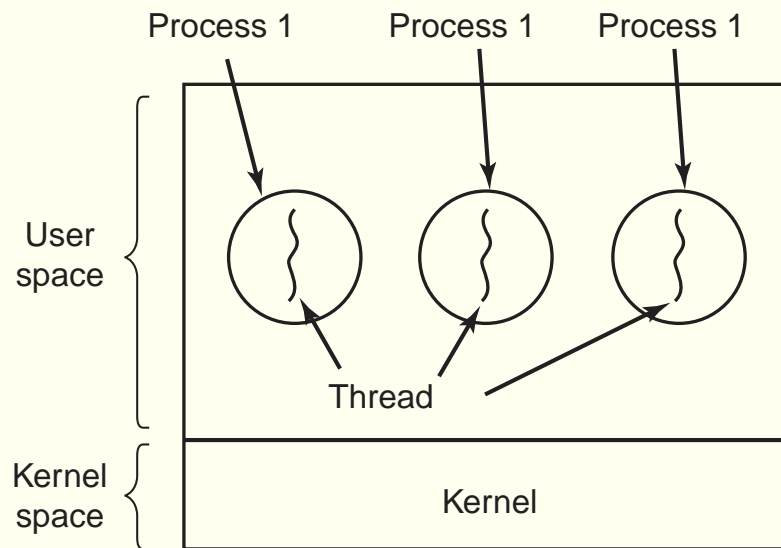


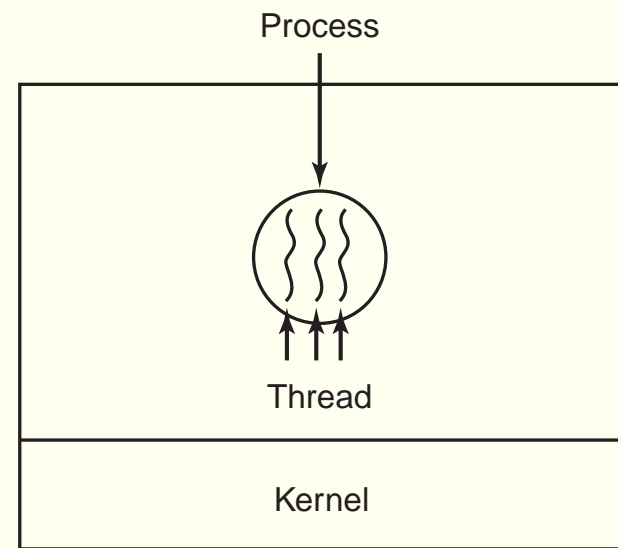
MC514–Sistemas Operacionais: Teoria e Prática
1s2008

Processos e sinais

Processos e threads



(a)



(b)

fork()

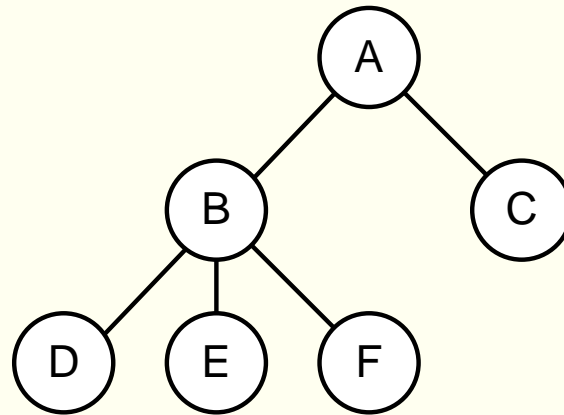
- Cria um novo processo, que executará o mesmo código
- Retorna
PID do processo criado para o pai e
0 para o processo filho

Espaços de endereçamento distintos

```
if (fork() == 0)
    s = 0;
    printf("Filho: &s=%p s=%d\n", &s, s);
} else {
    s = 1;
    printf("Pai: &s=%p s=%d\n", &s, s);
}
```

- Veja o código: fork0.c

Hierarquia de processos



Como implementar uma arquitetura como esta utilizando a chamada fork?

- Veja os códigos: fork1.c fork2.c fork3.c

wait()

```
pid_t wait(int *status);
```

- Aguarda pela morte de um filho.
- Bloqueia o processamento
- Retorna o pid do filho morto
- status indica causa da morte
- Veja os códigos: wait1.c, wait2.c e getppid.c

waitpid()

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

- Aguarda pela morte de um filho.
 - Específico pid = PID
 - Qualquer pid = -1
- Versão não bloqueante (options = WNOHANG)
- Veja o código waitpid1.c

Argumentos para os processos

- Exemplo

```
$ cp
```

```
cp: missing file arguments
```

```
Try 'cp --help' for more information.
```

```
$ cp arq-origem arq-destino
```

- Implementação

```
int main(int argc, char** argv)
```


Variáveis de ambiente

- Exemplo

`PWD=/home/islene/mc514`

`HOME=/home/islene`

`LOGNAME=islene`

- Implementação

```
int main(int argc, char** argv, char** envp)
```

- Veja o código: envp.c

Execução de outros códigos

Família exec

```
int execve(const char *filename,  
           char *const argv [],  
           char *const envp[]);
```

- Executa o programa filename, passando argv[] e envp[] como argumentos
- Outras opções: execl(), execlp(), execl(), execv() e execvp()
- Veja o código: execve1.c

Terminação de processos

- saída normal (voluntária)
- saída por erro (voluntária)
- erro fatal (involuntária)
- morto por outro processo (involuntária)
- Veja o código: `execve2.c`

Shell

```
#define TRUE 1
```

```
while (TRUE) {                                /* repeat forever */
    type_prompt( );                           /* display prompt on the screen */
    read_command(command, parameters);        /* read input from terminal */

    if (fork( ) != 0) {                        /* fork off child process */
        /* Parent code. */
        waitpid(-1, &status, 0);              /* wait for child to exit */
    } else {
        /* Child code. */
        execve(command, parameters, 0);       /* execute command */
    }
}
```

Shell

- Como implementar processos em *background*?

```
$ cp arquivo_grande copia_grande &
```

Como criar threads?

- Veja a documentação da função `clone()`
- Veja o código `clone.c`

Como tratar erros de execução?

```
FILE *file = fopen ("arq.txt","r");
```

- Valor de retorno indica se a execução foi bem sucedida:
Upon successful completion fopen returns a FILE pointer. Otherwise, NULL is returned and the global variable errno is set to indicate the error.
- Veja o manual: fopen, errno e perror
- Veja o código: fopen.c

Como tratar erros deste tipo?

```
int *px = (int*) 0x01010101;  
*px = 0;
```

- Programa recebe um sinal SIGSEGV
- O comportamento padrão é terminar o programa
- Veja o código: segfault1.c (use o gdb!)

E erros deste tipo?

```
int i = 3/0;
```

- Programa recebe um sinal SIGFPE
- O comportamento padrão é terminar o programa
- Veja o código: div0.c (use o gdb!)

Sinais

- Indicam a ocorrência de condições excepcionais
- Tipos de sinais
 - Divisão por zero
 - Acesso inválido à memória
 - Interrupção do programa
 - Término de um processo filho
 - Alarme
- Existem sinais síncronos e assíncronos

Alarme

Exemplo de sinal assíncrono

```
unsigned int alarm(unsigned int seconds);
```

- Envia um sinal do SIGALRM para o processo após alguns segundos.
- Veja o código: alarm1.c

Como ignorar um sinal?

- É possível ignorar SIGALRM?

```
signal(SIGALRM, SIG_IGN);
```

Veja o código: alarm2.c

- É possível ignorar SIGSEGV?

```
signal(SIGALRM, SIG_IGN);
```

Veja o código: segfault2.c

Como tratar um sinal?

- Rotina `signal` permite alterar o comportamento do programa em relação ao recebimento de um sinal específico.

```
typedef void (*sighandler_t)(int);  
sighandler_t signal(int signum,  
                    sighandler_t handler);
```

Como tratar SIGSEGV?

- Devemos escrever um tratador

```
void trata_SIGSEGV(int signum) {  
    /* ... */  
}
```

- e instalá-lo

```
signal(SIGSEGV, trata_SIGSEGV);
```

- Veja o código: segfault3.c (use o gdb!)

Como recuperar o tratador padrão?

```
signal(SIGALRM, SIG_DFL);
```

- É possível fazer isso a partir do programa principal
Veja o código: alarm3.c
- ou a partir do próprio tratador.
Veja os códigos: alarm4.c e segfault4.c

Um comentário sobre portabilidade

The original Unix `signal()` would reset the handler to `SIG_DFL`, and System V (and the Linux kernel and `libc4,5`) does the same. On the other hand, BSD does not reset the handler, but blocks new instances of this signal from occurring during a call of the handler. The `glibc2` library follows the BSD behaviour.

Problemas de consistência

- Um tratador de sinais pode encontrar dados “inconsistentes” .
- Veja o código: `consistencia.c`
- Quais funções podem ser invocadas a partir de um tratador de sinais?

Controle de execução

- SIGKILL: encerra a execução.
- SIGTERM: encerra a execução, mas um tratador pode ser invocado.
- SIGSTOP: interrompe a execução.
- SIGTSTP: interrompe a execução, mas um tratador pode ser invocado.
- SIGCONT: continua a execução
- Veja os códigos: sigterm.c sigint.c e sigcont.c

Como depurar um processo filho?

Primeira abordagem

- Após o `fork()` o processo filho pode interromper seu processamento via `raise(SIGSTOP)`;
- O `gdb` pode depurar um processo que já está rodando via comando `attach`
- Colocamos um breakpoint adequado no processo filho
- Enviamos um sinal `SIGCONT` para o processo filho
- Veja o código: `attach.c`

Comando pause()

```
alarm(nseg);  
pause(); /* Bloqueia execução  
         até a chegada de um sinal */
```

- Veja o código sleep0.c

Como bloquear sinais

Trabalha-se com um conjunto de sinais

```
sigset_t set;
```

sobre o qual as seguintes operações são possíveis:

- `int sigemptyset (sigset_t *SET);`
- `int sigfillset (sigset_t *SET);`
- `int sigaddset (sigset_t *SET, int SIGNUM);`
- `int sigdelset (sigset_t *SET, int SIGNUM);`

Como bloquear sinais

```
int sigprocmask (int HOW,  
                const sigset_t *restrict SET,  
                sigset_t *restrict OLDSET)
```

- SIG_BLOCK: bloqueia os sinais no conjunto set, adicionando-os à máscara atual.
- SIG_UNBLOCK: desbloqueia os sinais no conjunto set, removendo-os da máscara atual
- SIG_SETMASK: substitui a máscara atual.
- Máscara anterior é retornada em OLDSET.

Implementando sleep()

Funciona sempre?

```
int sleep(int nseg) {  
    /* Bloqueia todos os sinais  
       exceto SIGALRM */  
    alarm(nseg);  
    pause(); /* Bloqueia execução  
               até a chegada de um sinal */  
    /* Restaura máscara anterior */  
}
```

- Veja o código: sleep.c

Implementando sleep()

Funciona sempre?

```
int sleep(int nseg) {  
    /* Bloqueia todos os sinais */  
    alarm(nseg);  
    /* Desbloqueia SIGALRM em mask */  
    sigsuspend(&mask); /* Bloqueia execução,  
                           instala mask e  
                           aguarda um sinal */  
    /* Restaura máscara anterior */  
}
```

- Veja o código: sigsuspend.c

Como depurar um processo filho?

Segunda abordagem

- Após o `fork()` o processo filho pode interromper seu processamento via `sigsuspend()`.
- Processo filho aguarda `SIGUSR1`
- Usuário envia `SIGUSR1`
- Veja o código: `attach_SIGUSR1.c`

Como tratar a morte de um filho?

```
if (fork() != 0) /* Processo pai */  
    if (wait(NULL))  
        printf("Meu filho morreu\n");
```

- Processo pai fica bloqueado até que um filho morra.
- Veja o código: wait1.c

Como tratar a morte de um filho?

```
if (fork() != 0) /* Processo pai */  
    while (waitpid(-1, NULL, WNOHANG) == 0) {  
        printf("Meu filho ainda não morreu\n");  
        faz_alguma_coisa();  
    }  
}
```

- Processo pai faz verificações periódicas enquanto o filho não morre.
- Veja o código: `waitpid1.c`

Como tratar a morte de um filho?

```
void trata_SIGCHLD(int signum) {  
    int pid;  
    pid = wait(NULL);  
    printf("Meu filho %d morreu.\n", pid);  
}
```

- Sinal SIGCHLD é enviado quando um filho morre.
- Veja o código: sigchld1.c

Como tratar a morte de um filho?

- Suponha que o processo pai quer gerar todos os filhos antes de saber das mortes.

```
/* Bloqueia SIGCHLD */  
  gera_filhos();  
/* Desbloqueia SIGCHLD */  
/* Aguarda mortes */
```

- Será que usando o mesmo tratador do código sigchld1.c todas as mortes serão percebidas?
- Veja o código: sigchld2.c

Como tratar a morte de um filho?

```
void trata_SIGCHLD(int signum) {  
    int pid;  
    while ((pid = waitpid(-1, NULL, WNOHANG)) > 0) {  
        printf("Meu filho %d morreu.\n", pid);  
        n_filhos++;  
    }  
}
```

- Mais de um filho pode ter morrido enquanto o sinal não foi tratado.
- Veja o código: sigchld3.c

Duelo entre pai e filho

- Pai envia SIGTERM para o filho
- Filho envia SIGTERM para o pai
- Ambos devem morrer
- Veja os códigos: duelo1.c e duelo2.c

Tratadores encadeados

- Um sinal pode ser tratado durante o tratamento de outro sinal
- Veja o código: `encadeados.c`
- Como tentar bloquear isto?
- Veja o código: `encad-bloq1.c`

sigaction()

```
int sigaction(int signum,  
              const struct sigaction *act,  
              struct sigaction *oldact);  
  
struct sigaction {  
    void (*sa_handler)(int);  
    sigset_t sa_mask;  
    /*    */  
};
```

- Estabele uma função e uma máscara para ser usada no momento do tratamento do sinal.
- Veja o código: encad-bloq2.c

Pthreads e Sinais

- Sinais são uma propriedade dos processos
- Máscaras são propriedades de threads
- Veja o código: `thr1-sinais.c`