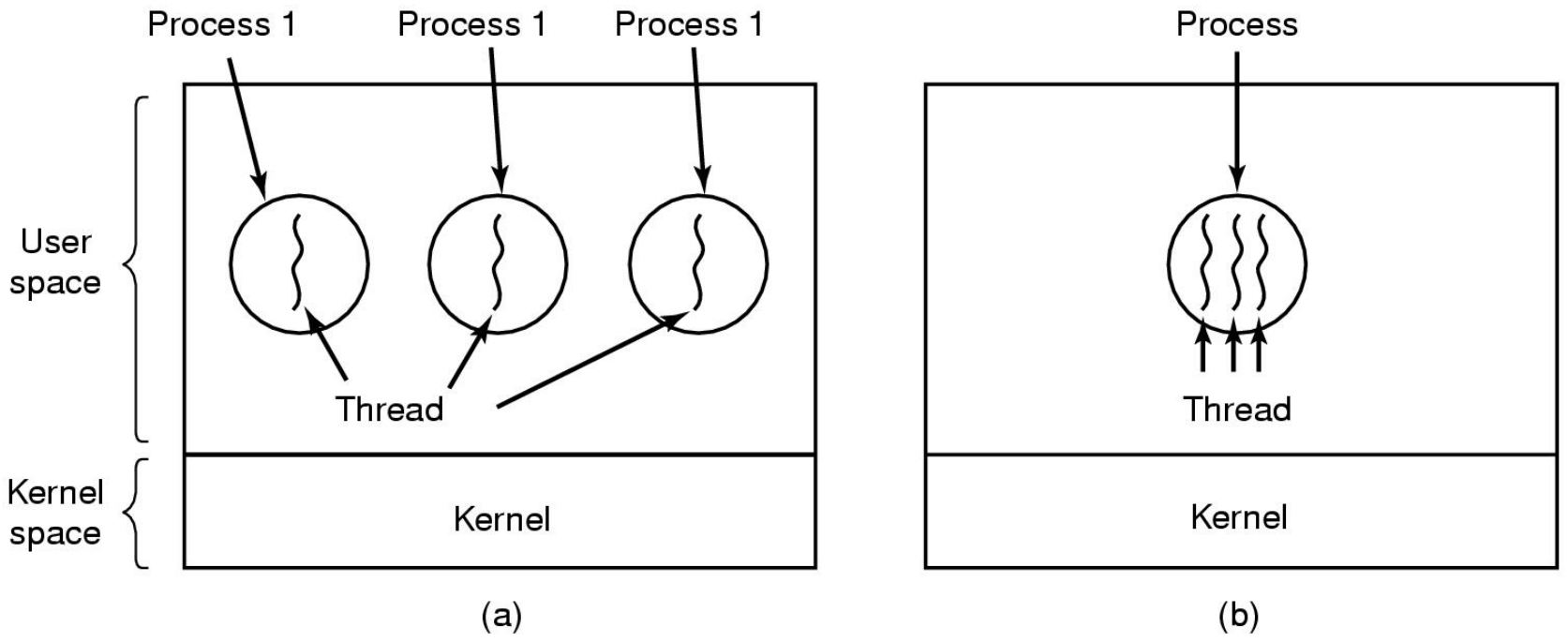
MC504 - Sistemas Operacionais

Processos e Sinais

Islene Calciolari Garcia
Instituto de Computação - Unicamp
Primeiro Semestre de 2017

Sumário

Processos e threads



Tanenbaum: Figura 2.6

fork()

- Cria um novo processo, que executará o mesmo código
- Retorna

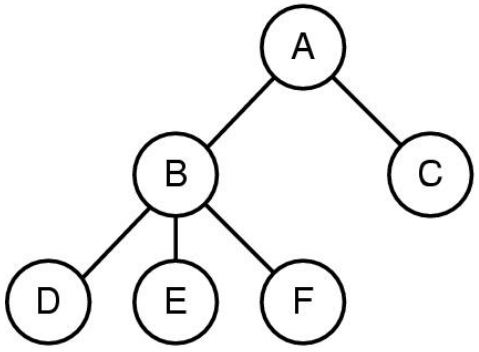
  PID do processo criado para o pai e
  0 para o processo filho

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



Tanenbaum: Figura 1.12

Como implementar uma arquitetura como esta utilizando a chamada `fork`?

- Veja os códigos: `fork1.c`, `fork2.c` e `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` e `wait2.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=/l/home/islene/mc504
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()`, `execle()`, `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 */
      }
}
```

Tanenbaum: Figura 1.19

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? *Async-Signal-Safe Functions*

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

Pthreads e Sinais

- Sinais são uma propriedade dos processos
- Máscaras são propriedades de threads

- Veja o código: `thr-sinais.c`