# Sistemas Operacionais 2

## *POSIX Threads: pthreads*

Hélio Crestana Guardia
*helio@dc.ufscar.br*

Universidade Federal de São Carlos
Departamento de Computação

# Processos

- **Criação: *fork( )***
  - Cópia de memória de pai para filho
  - Duplicação dos descritores
  - *Copy-on-write* pode atrasar cópia até que dados sejam necessários
  - ***vfork***( ) não copia dados, aguardando ***exec***( )
  - Mecanismos de comunicação (IPC) são usados para passar informações entre pai e filho(s)
- **Comunicação e sincronização:**
  - **Arquivos**: arquivos comuns e *mmap'ed*
  - **BSD IPC**: *sockets* (UNIX e INET), *pipes*
  - **System V IPC**: *shared memory,* semáforos, filas de mensagem

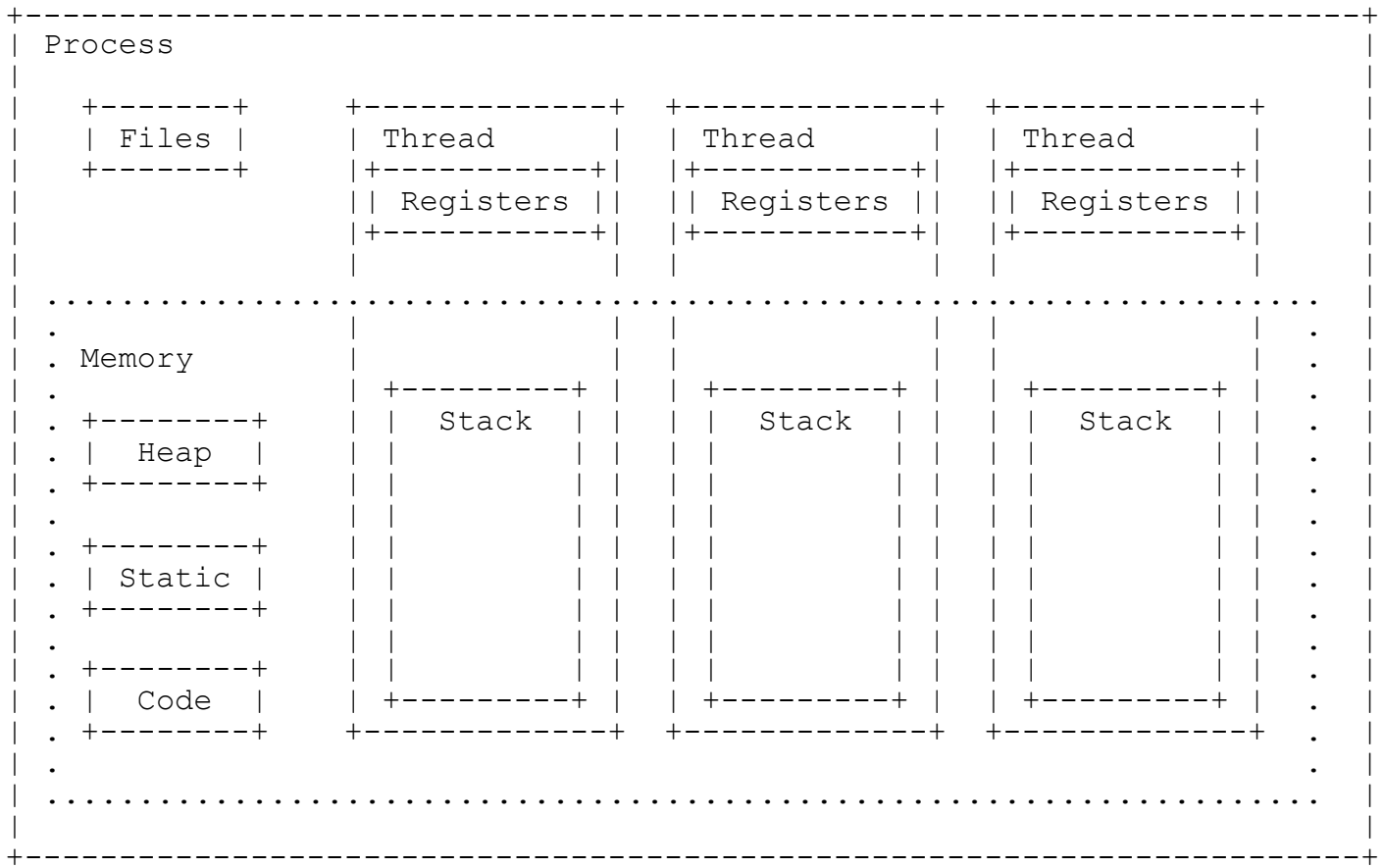
# *Threads*

- Em ambientes multiprocessados com **memória compartilhada,** como SMPs, ***threads*** podem ser usadas para **execução paralela**
- Uma ***thread*** (tarefa?) pode ser definida como um **fluxo de instruções** (**função**) independente, **associado a um processo**
- Processos contêm ao menos uma *thread*, iniciada com a função ***main()***
- Em um programa ***multi-threaded***, diversas funções podem ser selecionadas para execução simultânea
- Execução simultânea é possibilitada pela replicação de alguns recursos para cada *thread*
- *Threads* compartilham a **mesma memória global** (dados e *heap*), mas cada uma possui sua própria **pilha** (*automatic variables*: alocação em chamadas de funções)

# *Threads*: implementações

- Padrão IEEE POSIX 1003.1c (POSIX.1) define uma **interface** comum para manipulação de *threads*, chamada ***POSIX threads***, ou ***Pthreads***
- Formas de implementação:
  - Implementação **dentro do *Kernel*** (***kernel-space***):
    - trocas de contexto ocorrem sem interferência do usuário ou da aplicação
    - Podem beneficiar-se de paralelismo em ambientes multiprocessados
  - Implementação no **espaço de usuário** (***user-space***):
    - trocas de contexto controladas pela aplicação, mais rápidas - *swapcontext*(2) ou *setjmp*(3) /*longjmp*(3)
    - Mais portáveis (?)
    - Limitação do paralelismo efetivo: *threads* dividem fatias de tempo do processo
- Bibliotecas de *threads*:
  - ***LinuxThreads***: implementação *Pthreads* original no Linux (já obsoleta)
  - **NPTL** (***Native POSIX Threads Library***): imlementação *Pthreads* moderna no **Linux**, mais aderente à especificação POSIX.1 e com melhor desempenho para criação de um grande número de *threads*. Requer *kernel* 2.6.

# *Threads*: Recursos




(From the DECthreads manual)

# *Threads*: dados compatilhados

http://www.opengroup.org/onlinepubs/000095399/basedefs/xbd_chap03.html#tag_03_393

POSIX.1 especifica que *threads* de um processo devem compartilhar os seguintes atributos:

- *process ID*
- *parent process ID*
- *process group ID and session ID*
- *controlling terminal*
- *user and group IDs*
- *open file descriptors*
- *record locks (see fcntl(2))*
- *signal dispositions*
- *file mode creation mask (umask(2))*
- *current directory (chdir(2)) and root directory (chroot(2))*
- *interval timers (setitimer(2)) and POSIX timers (timer_create())*
- *nice value (setpriority(2))*
- *resource limits (setrlimit(2))*
- *measurements of the consumption of CPU time (times(2)) and resources (getrusage(2))*

# *Threads*: dados replicados

Além da **pilha** (***stack***), POSIX.1 determina que os seguintes atributos devem ser distintos para cada *thread*:

- *thread ID (the pthread_t data type)*
- *signal mask (pthread_sigmask())*
- *the errno variable*
- *alternate signal stack (sigaltstack(2))*
- *real-time scheduling policy and priority (sched_setscheduler(2) and sched_setparam(2))*

No **Linux**, os seguintes atributos também são mantidos **por *thread***:

- *capabilities (capabilities(7))*
- *CPU affinity (sched_setaffinity(2))*

# *Threads*: Aspectos

- *Threads* são associadas a um processo, do qual compartilham recursos:
  - Alterações globais são visíveis por todas as *threads* (manipulação de arquivos, e.g.)
  - **Ponteiros iguais** apontam para a **mesma área** de memória
  - Acessos simultâneos à mesma área de memória são possíveis e requerem **sincronização**
  - *Threads* de um processo devem ser alocadas na mesma memória, normalmente no mesmo processador

# *Threads*: Aspectos

**Reentrância de código**

- **Funções reentrantes** apresentam comportamento correto mesmo se chamadas simultaneamente por diversas *threads*
- Funções que manipulam **informações globais** do processo (memória, arquivos, etc.) devem ser cuidadosamente projetadas para garantir a reentrância de código
- Abordagens para prover reentrância:
  - passagem de parâmetros
  - uso de dados internos da *thread* (*thread-specific*), alocados na pilha

***Thread-safety***

- ***Thread-safety*** está relacionada com **condições de disputa** (***race conditions***) e com o comprometimento de dados globais do processo, produzindo **resultados incorretos** ou **imprevisíveis** em função da **ordem** em que *threads* são executadas
- **Função** é dita ***thread-safe*** quando seu comportamento é correto mesmo quando executada simultaneamente por diversas *threads*
  - Garantia é normalmente obtida **encapsulando** (***wrapping***) a função original em uma nova, que utiliza um mecanismo de controle de acesso (***mutex***)
  - Funções e chamadas de sistema são ***non-thread-safe*** a menos que atestado o contrário

***Asynchronous-safe***

- Função ***asynchronous-safe*** é aquela que pode ser usada dentro do contexto de tratadores de sinal (***signal handlers***)

# Aspectos da Lib C

- *Reentreant Routines: a function whose effect, when called by two or more threads, is guarenteed to be as if the threads each executed the function one after another in an undefined order, even if the actual execution is interleaved.*
- *All routines which use static data (**strtok**, **ctime**, **gethostbyname**) are nonreentrant*
- *Non reentrant routines have a _r reentrant counterpart*
- *Some implementation reimplement functions that return static data to use thread specific data.*
- *Errno*
  - *The variable errno is now per thread*
  - *Declaring errno extern int is no longer valid*
  - *Must include errno.h to get the correct declaration*
- *Stdio*
  - *All stdio calls inherintly lock the FILE*
  - *A recursive locking mechanizim is provided to lock accross calls.*
  - *For speed previous macro routines have a new name*

- *void flockfile(FILE * file);*
- *int ftrylockfile(FILE * file);*
- *void funlockfile(FILE * file);*
- *int getc_unlocked(FILE * file);*
- *int getchar_unlocked(void);*
- *int putc_unlocked(FILE * file);*
- *int putchar_unlocked(void);*

# Porque usar *Threads*

**Velocidade**:

- Comunicação entre *threads* é mais rápida, baseada em memória compartilhada

**Economia de recursos:**

- Compartilhamento de recursos favorece economia de memória e diminui tempo das operações de criação de *threads* em relação aos processos

***Responsiveness* (Interatividade)**:

- Uso de múltiplas *threads* em aplicação interativa permite que programa continue respondendo, mesmo que parte esteja bloqueada ou realizando operação demorada

**Desempenho de E/S (*I/O throughput*):**

- E/S tradicional bloqueia processo
- Com *threads*, apenas a *thread* responsável pela operação é bloqueada, permitindo a sobreposição de processamento com E/S

**Portabilidade:**

- POSIX *threads* são portáveis
- Bibliotecas para *threads* são emuladas em sistemas monoprocessados. Em computadores paralelos permitem explorar o uso de todos os processadores.

# Quando usar *Threads*

- Para poder beneficiar-se do uso de *threads*, programas devem ser organizados em **tarefas independentes**, que podem ser executadas de maneira **concorrente**
- Aspectos das **aplicações** que favorecem o uso de *threads*:
  - **Bloqueio** por possivelmente longos períodos de tempo
  - Uso de muitos **ciclos** de CPU
  - Possibilidade de sobrepor processamento e E/S
  - Tratamento de **eventos assíncronos**
    - Espera por eventos (servidores de rede, e.g.)
    - Apresentam **interação** (aplicações de tempo real, e.g.)
    - Uso da função ***select( )***
  - Possuem **prioridades** incomuns (maiores ou menores)
  - Podem ser executadas em **paralelo** com outras tarefas

# Modelos de Programação

- ***Manager/worker*** (**mestre/escravo**):
  - *Thread* **mestre** (***manager***) atribui operações para outras *threads* **escravas**
  - **Mestre** normalmente trata operações de **entrada de dados** (solicitações), distribuindo o serviço entre os demais
  - **Escravos** podem ser criados estática ou dinamicamente
- ***Pipeline***:
  - Tarefa é quebrada em suboperações, executadas em série, mas de maneira **concorrente**, por *threads* diferentes
- ***Peer***:
  - Semelhante ao modelo mestre/escravo mas, depois de criar os escravos, *thread* principal participa na execução do trabalho

# Posix Threads: pthreads API

***API Pthreads*** apresenta 3 grupos de funções:

- Funções para **gerenciar** *threads*
  - Tratam da criação e manipulação de *threads* (criar e destruir) e do ajuste de seus atributos (*joinable*, *scheduling*, etc.)
- Funções para **sincronizar** a execução de *threads* e bloquear o acesso a recursos
  - ***Mutexes***: manipulam um mecanismo de controle de exclusão mútua, mutex, e seus atributos.
  - ***Condition variables***: incluem funções para criar, destruir, esperar e tratar sinais baseada em valores de variáveis.
  - Outras: (*sem)*, *barrier*, *rw_locks*, *spins*, *keys*, ...
- Funções para gerenciar o **escalonamento** de *threads*

# Criação de *Threads*

*int* ***pthread_create*** *(pthread_t * thread,*
*const pthread_attr_t * attr,*
*void * (*start_routine)(void *),*
*void *arg);*

Ex: *result =* ***pthread_create****(&th, NULL, função, NULL);*

- Quando um programa é iniciado com ***exec***, uma única *thread* é criada (***initial thread*** ou ***main thread***).
- *Threads* adicionais são criadas com ***pthread_create***
- ***ptrhead_create( ):*** cria uma nova *thread*, especificando a **função** que deve ser executada
- Semelhante à combinação de *fork* e *exec*, com espaço de endereçamento compartilhado
- Retorna um *thread id* em *thread*
- Se ***attr*** é NULL, usa parâmetros *default*

# *Threads*: atributos

Ao invés de usar valores *default*, estrutura ***pthread_attr_t*** pode ser ajustada e passada como parâmetro para a criação de *threads*.

**Atributos:** *detachstate, schedpolicy, schedparam, inheritsched* e *scope*

***Detachstate***:

- PTHREAD_CREATE_JOINABLE (*default*), ou PTHREAD_CREATE_DETACHED
- O parâmetro ***joinable*** permite que outras *threads* esperem pela conclusão de uma *thread*, recuperando a condição de saída
- Quando ***detached***, recursos de uma *thread* são liberados imediatamente em sua conclusão e *pthread_join* não pode ser usado para sincronização (espera pelo fim)
- *pthread_detach( )* permite ajustar estado para ***detached*** em tempo de execução.

***Schedpolicy:***

- Seleciona a política de escalonamento para a *thread*
- SCHED_OTHER: escalonamento **normal**, não tempo-real, é a política *default*
- SCHED_RR: *realtime*, *round-robin*
- SCHED_FIFO: *realtime*, *first-in first-out*
- Políticas SCHED_RR e SCHED_FIFO são permitidas apenas para processos com privilégios de superusuário
- Política de escalonamento pode ser alterada em tempo de execução com o comando *pthread_setschedparam( )*

# *Threads*: atributos

***Schedparam***

- Contém parâmetros de escalonamento: **prioridade** (***default=0***).
- É relevante apenas para as políticas **SCHED_RR** e **SCHED_FIFO**.
- Pode ser alterada em tempo de execução com o comando *pthread_setschedparam( )*

***Inheritsched***

- Indica se a política e a prioridade de escalonamento são definidas pelos parâmetros, ou herdadas da *main thread*
- **PTHREAD_EXPLICIT_SCHED** e **PTHREAD_INHERIT_SCHED** *(default)*

***Scope***

- Define o âmbito de contenção (concorrência) da *thread* pelo uso de CPU
- **PTHREAD_SCOPE_SYSTEM**: *thread* compete pela CPU com todos os outros processos sendo executados no sistema (valor *defautl*)
- **PTHREAD_SCOPE_PROCESS**: *thread* compete apenas com outras *threads* do mesmo processo

# *Threads*: atributos

- *int* ***pthread_attr_init****(pthread_attr_t *attr);*
- *int* ***pthread_attr_destroy****(pthread_attr_t *attr);*

- *int* ***pthread_attr_setdetachstate****(pthread_attr_t *attr, int detachstate);*
- *int* ***pthread_attr_getdetachstate****(const pthread_attr_t *attr, int *detachstate);*
- *int* ***pthread_attr_setschedpolicy****(pthread_attr_t *attr, int policy);*
- *int* ***pthread_attr_getschedpolicy****(const pthread_attr_t *attr, int *policy);*
- *int* ***pthread_attr_setschedparam****(pthread_attr_t *attr, const struct sched_param *param);*
- *int* ***pthread_attr_getschedparam****(const pthread_attr_t *attr, struct sched_param *param);*
- *int* ***pthread_attr_setinheritsched****(pthread_attr_t *attr, int inherit);*
- *int* ***pthread_attr_getinheritsched****(const pthread_attr_t *attr, int *inherit);*
- *int* ***pthread_attr_setscope****(pthread_attr_t *attr, int scope);*
- *int* ***pthread_attr_getscope****(const pthread_attr_t *attr, int *scope);*

- O ajuste de atributos de *threads* é realizado preenchendo um objeto ***pthread_attr_t*** passado como argumento na função de criação da *threadt pthread_create( ).*
- *pthread_attr_init( )* inicia um objeto ***pthread_attr_t*** e o preenche com valores *default*.

# Término de *Threads*

***void pthread_exit(void * return_value);***

- ***pthread_exit( )***: termina uma *thread*, que também pode ser encerrada com o retorno da função executada.
- Semelhante a *exit( )*
- Função *exit( )* termina (*kills*) todas as *threads* e termina o processo
- Se a *thread* corrente é a última, processo termina
- Uso de *return( )* na função da *thread* (*start_routine*) é equivalente a *pthread_exit( )*
- Uso de *return( )* na função *main( )* (*initial thread*) é equivalente a chamar a função *exit()*

# *Thread Cancellation*

- Mecanismo de **cancelamento** (***thread cancellation***) permite terminar execução de qualquer outra *thread* do processo.
- Cancelamento é pertinente quando diversas *threads* atuam de maneira **segmentada** na busca de uma solução:
  - Demais *threads* devem ser canceladas quando alguma conclui a atividade
  - Alguma *thread* encontra erro que impede prosseguimento da atividade

*Ex*: usuário interrompe navegador que usava diversas *threads* para transferir arquivos associados a uma página

- Cada *thread* controla seu estado de cancelamento (***cancelability state***)
- *Threads* são originalmente canceláveis
- *Threads* podem instalar (*push*) e remover (*pop*) **rotinas de tratamento** (***cleanup handlers***) para execução ao serem canceladas.
- **Pontos de cancelamento** (*cancellation points*) são pontos na execução do programa em que verifica-se se há pedido de cancelamento pendente. Em caso positivo, execução da *thread* é encerrada.

# *Thread Cancellation*

*Threads* estão sujeitas a pedidos de cancelamento, que são **deferidos** até que uma função que é ponto de cancelamento seja executada. Esse comportamento padrão pode ser alterado.

*int* ***pthread_setcancelstate****(int state, int *oldstate);*

*int* ***pthread_setcanceltype****(int type, int *oldtype);*

***pthread_setcancelstate()*** ajusta o estado de cancelamento (*cancelability state*):

- **PTHREAD_CANCEL_ENABLE**: *thread* é cancelável (estado padrão, inclusive para a *thread* inicial). *Cancelability type* determina como *thread* reagirá ao pedido de cancelamento recebido.
- **PTHREAD_CANCEL_DISABLE**: *thread* não é cancelável. Se *cancellation request* for recebido, o cancelamento fica bloqueado até que o estado de cancelamento (*cancelability*) seja habilitado.

***pthread_setcanceltype()*** ajusta o tipo de cancelamento da *thread* (*cancelability type*):

- PTHREAD_CANCEL_DEFERRED: pedido de cancelamento recebido é preterido até a chamada de função que é *cancellation point* (tipo padrão, inclusive para *thread* inicial).
- PTHREAD_CANCEL_ASYNCHRONOUS: *thread* pode ser cancelada a qualquer momento, tipicamente assim que receber o pedido de cancelamento.

# Thread Cancellation

*void* ***pthread_cleanup_push*** *(void (*function) (void *), void *arg);*
*void* ***pthread_cleanup_pop*** *(int execute);*

- *Handlers* são executados (LIFO) quando:
  - *thread* é cancelada por outra
  - *thread* termina (*pthread_exit* ou retorno da função), ou
  - *pthread_cleanup_pop* é chamada com o parâmetro *execute* diferente de zero
- Execução da *thread* é terminada depois da execução das rotinas de tratamento. Neste caso, valor **PTHREAD_CANCELED** é retornado no caso de chamadas à função *pthread_join*( )

*void* ***pthread_testcancel****(void);*

- ***pthread_testcancel*** verifica se há um pedido de cancelamento pendente e o trata
- É útil em situações em que há longos trechos de código sem chamadas a funções que são pontos de cancelamento
- Se o cancelamento está desabilitado, ou não há cancelamentos pendentes, nada ocorre

# Thread Cancellation

*void **pthread_cleanup_push_defer_np** (void (*routine)(void *), void *arg);*
*void **pthread_cleanup_pop_restore_np** (int execute);*

- *These functions are the same as pthread_cleanup_push(3) and pthread_cleanup_pop(3), except for the differences noted on this page.*
- *Like pthread_cleanup_push(3), pthread_cleanup_push_defer_np() pushes routine onto the thread's stack of cancellation clean-up handlers. In addition, it also saves the thread's current cancelability type, and sets the cancelability type to "deferred" (see pthread_setcanceltype(3)); this ensures that cancellation clean-up will occur even if the thread's cancelability type was "asynchronous" before the the call.*
- *Like pthread_cleanup_pop(3), pthread_cleanup_pop_restore_np() pops the top-most clean-up handler from the thread's stack of cancellation clean-up handlers. In addition, it restores the thread's cancelability type to its value at the time of the matching pthread_cleanup_push_defer_np().*
- *The caller must ensure that calls to these functions are paired within the same function, and at the same lexical nesting level. Other restrictions apply, as described in pthread_cleanup_push(3).*

```
pthread_cleanup_push_defer_np(routine, arg);
pthread_cleanup_pop_restore_np(execute);
```

É equivalente a:

```
int oldtype;
pthread_cleanup_push(routine, arg);
pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, &oldtype);
...
pthread_setcanceltype(oldtype, NULL);
pthread_cleanup_pop(execute);
```

# Cancellation Points

Funções de *threads* POSIX que são pontos de cancelamento:

- *pthread_join(), pthread_cond_wait(), pthread_cond_timedwait(), pthread_testcancel(), sem_wait(), sigwait()*

http://www.opengroup.org/onlinepubs/000095399/functions/xsh_chap02_09.html#tag_02_09_05

Funções que são pontos de cancelamento:

- *accept(), connect(), aio_suspend(), clock_nanosleep(), sleep(), usleep(), nanosleep(), pause(), open(), close(), creat(), fcntl(), lockf(), fdatasync(), fsync(), getmsg(), getpmsg(), putmsg(), putpmsg(), mq_receive(), mq_send(), mq_timedreceive(), mq_timedsend(), msgrcv(), msgsnd(), msync(), pread(), pwrite(), select(), pselect(), poll(), pthread_cond_timedwait(), pthread_cond_wait(), pthread_join(), pthread_testcancel(), read(), readv(), recv(), recvfrom(), recvmsg(), sem_timedwait(), sem_wait(), send(), sendmsg(), sendto(), sigpause(), sigsuspend(), sigtimedwait(), sigwait(), sigwaitinfo(), system(), tcdrain(), wait(), waitid(), waitpid(), write(), writev()*

Pontos de cancelamento também podem ocorrer em:

- *access(), asctime(), asctime_r(), catclose(), catgets(), catopen(), closedir(), closelog(), ctermid(), ctime(), ...*

# *Threads*: tratamento de sinais

- *Threads* **compartilham tratadores de sinais** do processo
- Contudo, **cada *thread*** pode ter sua própria **máscara** de sinais

**Tipos de sinais:**

- ***Asynchronous***: entregue para **alguma** *thread* que não o está bloqueando
- ***Synchronous***: entregue para a *thread* que o causou. Ex. SIGFPE, SIGBUS, SIGSEGV, SIGILL
- ***Directed***: entregue para uma *thread* específica (*pthread_kill( )*)

- ***int pthread_kill(pthread_t thread, int sig);***
- Possibilita o envio de um sinal para uma ***thread*** **específica** do **processo corrente**, que está executando a chamada**.**

- **Sinais** enviados para processo com o comando ***kill( )*** afetam **todas as *threads*** (**o processo como um todo**).
  - Qualquer *thread* que não tenha bloqueado o sinal pode tratá-lo

# *Threads*: tratamento de sinais

- ***int pthread_sigmask(int how, const sigset_t *newmask, sigset_t *oldmask);***
- *pthread_sigmask( )* é semelhante a *sigprocmask( )*
- Tratadores de sinal (gerenciados com *sigaction*) estão associados ao processo
- Máscaras de sinais são individualizadas para cada *thread*

*// select any of pending signals from SET or wait for any to arrive:*

- ***int sigwait(const sigset_t *set, int *sig);***

- Estratégias para **tratamento de sinais**:
  - Definição de tratadores para sinal
  - Uso de máscara durante execução da rotina
  - Seleção de *threads* dedicadas ao tratamento de sinais específicos

# *Threads*: Uso da pilha (*stack*)

- Alocações de variáveis estáticas da *thread* ocorre na **pilha**
- POSIX **não** determina o tamanho da **pilha** para cada *thread*, que é dependente de implementação
- Limites do SO: *ulimit -s, getrlimit ( ) - RLIMIT_STACK*
- Exceder o limite da pilha é problema crítico
- Programas seguros e portáveis devem preocupar-se com o tamanho da pilha, usando ***pthread_attr_setstacksize***
- ***pthread_attr_getstackaddr*** e ***pthread_attr_setstackaddr*** permitem determinar a localização da pilha em regiões específicas da memória

- *pthread_attr_getstacksize (attr, stacksize)*
- *pthread_attr_setstacksize (attr, stacksize)*
- *pthread_attr_getstackaddr (attr, stackaddr)*
- *pthread_attr_setstackaddr (attr, stackaddr)*

# *Threads*: funções gerais

- ***pthread_self ( )***
  - retorna um identificador da *thread*
- ***pthread_equal (thread1,thread2)***
  - Compara 2 identificadores de *thread*.
  - Retorno 0 indica que *threads* são diferentes
  - Operador *C* "= =" não deve ser usado para a comparação, uma vez que ***thread IDs*** são objetos.
- ***pthread_once (once_control, init_routine)***
  - Executa a função *init_routine* uma única vez em um processo
  - Primeira chamada faz com que a função seja executada, sendo que chamadas subsequentes não têm efeito.
  - *init_routine* é tipicamente uma função de inicialização
  - Parâmetro **once_control** requer inicialização**:**
    - *pthread_once_t once_control = PTHREAD_ONCE_INIT;*
- ***pthread_yield ( )***
  - Faz *thread* liberar o processador, retornando à fila de prontos

# *Threads*: Sincronização

- *Threads* possuem diversos mecanismos de sincronização:
  - ***Joins***: fazem com que uma *thread* espere até que outra complete (termine)
  - ***Semaphores***:sincroniza *threads* em função de valor de contador
  - ***Mutexes***: bloqueios para preservar **seções críticas** ou obter **acesso exclusivo** a recursos
  - ***Condition variables***: *pthread_cond_t*
  - ***Barriers***: *pthread_barrier_t*
  - ***Reader Writer locks***: *pthread_rwlock_t*
  - ***Thead Specific***: *pthread_key_t*
  - ***Spin***: *pthread_spinlock_t*

# *Detach e Join*

*int **pthread_join** (pthread_t thread, void ** status);*
*int **pthread_tryjoin_np** (pthread_t thread, void **retval);*
*int **pthread_timedjoin_np** (pthread_t thread, void **retval, const struct timespec *abstime);*

*The pthread_tryjoin_np() function performs a nonblocking join with the thread thread, returning the exit status of the thread in *retval. If thread has not yet terminated, then instead of blocking, as is done by pthread_join(3), the call returns an error.*

*The pthread_timedjoin_np() function performs a join-with-timeout. If thread has not yet terminated, then the call blocks until a maximum time, specified in abstime. If the timeout expires before thread terminates, the call returns an error. The abstime argument is a structure of the following form, specifying an absolute time measured since the Epoch (see time(2)):*

```
struct timespec {
    time_t tv_sec;     /* seconds */
    long   tv_nsec;    /* nanoseconds */
};
```

- Comando ***join*** é executado para esperar o fim de uma *thread*, de maneira semelhante a ***wait( )/waitpid( )*** para processos.
  - Identificador da *thread* é especificado; **não** é possível esperar por **qualquer** uma
  - *Threads* podem ser ***detached*** ou ***joinable (default POSIX)***

# Detach e Join

*int* ***pthread_join****(pthread_t thread, void ** status);*
*int* ***pthread_tryjoin_np****(pthread_t thread, void **retval);*
*int* ***pthread_timedjoin_np****(pthread_t thread, void **retval, const struct timespec *abstime);*

- Atributo na criação determina se *thread* é ***joinable*** ou ***detached***:
  - Declarar variável ***pthread_attr_t***
  - Iniciar variável com o comando ***pthread_attr_init( )***
  - Ajustar atributo ***detached status*** com ***pthread_attr_setdetachstate( )***
  - Ao final, liberar a variável com ***pthread_attr_destroy( )***

***int pthread_detach(pthread_t thread);***

- Função ***pthread_detach( )*** pode ser usada para mudar o *status* de uma *thread* para *detached*, mesmo se criada *joinable*
- Quando uma *joinable thread* termina, seu **identificador** e ***status* de saída** são mantidos até que outra *thread* execute *pthread_join*.
- ***pthread_detach( )*** faz com que recursos de uma *thread* sejam imediatamente liberados em sua conclusão

# Sincronização

- Problema: múltiplas *threads* acessando a mesma variável compartilhada
- Risco de corromper o dado se o acesso não for sincronizado
- Exemplo: ajuste de soma global

```
/* todas as threads tentam atualizar soma_global*/
soma_global+= partial_list_ptr[i];
```

- Considere duas *threads*
  - valor inicial de **soma_global** é 100
  - valores de **partial_list_ptr[i]** são 50 e 25 nas *threads* t1 e t2
- Dependendo da ordem de execução das *threads*, **soma_global** pode ser 150 ou 125
- Resultado deveria ser o mesmo da execução sequencial

# Seção Crítica

- Segmento de código que só pode ser executado por uma *thread* por vez
- Em *pthreads,* seções críticas podem ser implementadas usando *mutex*:
  - *Mutexes* possuem dois estados apenas:
    - Livre (*locked*) e
    - Bloqueado (*unlocked*)
  - A qualquer momento, somente uma *thread* pode bloquear (obter) um *mutex*
  - Bloqueio do *mutex* é feito de forma atômica
- *Thread* requisita *mutex* no início da seção crítica, ficando bloqueada até conseguir obtê-lo
- *Mutex* é liberado no final da seção crítica

# *Mutex*

- Objeto ***mutex*** (*short for "mutual exclusion"*) é usado para facilitar o bloqueio de recursos
- ***Mutexes*** podem ser bloqueados por apenas **uma** *thread* de cada vez
- Se 2 ou mais *threads* tentam bloquear o mesmo ***mutex***, apenas uma é bem-sucedida; demais são bloqueadas até a liberação do ***mutex***
- ***Mutexes*** são usados geralmente para:
  - **Proteger o acesso** a trechos de código (**região crítica**)
  - Garantir **exclusão mútua** na manipulação de recursos
- Para serem acessíveis, ***mutexes*** devem ser declarados como **variáveis globais**

# *Mutex*: criação e destruição

- ***int pthread_mutex_init*** *(pthread_mutex_t *mutex, __const pthread_mutexattr_t *mutexattr)*
- ***int pthread_mutex_destroy*** *(pthread_mutex_t *mutex)*: destrói *mutex*

- ***pthread_mutexattr_init (attr)***: cria atributos para *mutex*
- ***pthread_mutexattr_destroy (attr)***: destrói atributos de *mutex*

- Variáveis ***mutex*** (***pthread_mutex_t***) devem ser inicializadas

Inicialização **estática** (na declaração):

- ***pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;***

Inicializaçao **dinâmica**, ajustando atributos:

- ***pthread_mutex_init(&mutex, &attr); pthread_mutex_init(&mutex,NULL);***

- Parâmetro ***attr*** pode ser NULL (para parâmetros *default*), ou do tipo *pthread_mutexattr*, usado para seleção de parâmetros
- ***Mutexes*** são criados inicialmente no estado **liberado (*unlocked)***

# *Mutex*: atributos

- **Tipo** do ***mutex*** determina se este pode ser **bloqueado** novamente pela *thread* que já o bloqueou
- Tipos (*/usr/include/pthread.h*):

```
enum
{
  PTHREAD_MUTEX_TIMED_NP,
  PTHREAD_MUTEX_RECURSIVE_NP,
  PTHREAD_MUTEX_ERRORCHECK_NP,
  PTHREAD_MUTEX_ADAPTIVE_NP
#ifdef __USE_UNIX98
  ,
  PTHREAD_MUTEX_NORMAL = PTHREAD_MUTEX_TIMED_NP,
  PTHREAD_MUTEX_RECURSIVE = PTHREAD_MUTEX_RECURSIVE_NP,
  PTHREAD_MUTEX_ERRORCHECK = PTHREAD_MUTEX_ERRORCHECK_NP,
  PTHREAD_MUTEX_DEFAULT = PTHREAD_MUTEX_NORMAL
#endif
#ifdef __USE_GNU
  /* For compatibility.  */
  , PTHREAD_MUTEX_FAST_NP = PTHREAD_MUTEX_TIMED_NP
#endif
};
```

# *Mutex*: atributos

```
pthread_mutex_t timedmutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t recmutex = PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP;
pthread_mutex_t errchkmutex = PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP;
pthread_mutex_t adaptivemutex = PTHREAD_ADAPTIVE_MUTEX_INITIALIZER_NP;

/* /usr/include/pthread.h */
#if __WORDSIZE == 64
# define PTHREAD_MUTEX_INITIALIZER \
  { { 0, 0, 0, 0, 0, 0, { 0, 0 } } }
# ifdef __USE_GNU
#  define PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP \
  { { 0, 0, 0, 0, PTHREAD_MUTEX_RECURSIVE_NP, 0, { 0, 0 } } }
#  define PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP \
  { { 0, 0, 0, 0, PTHREAD_MUTEX_ERRORCHECK_NP, 0, { 0, 0 } } }
#  define PTHREAD_ADAPTIVE_MUTEX_INITIALIZER_NP \
  { { 0, 0, 0, 0, PTHREAD_MUTEX_ADAPTIVE_NP, 0, { 0, 0 } } }
# endif
#else
# define PTHREAD_MUTEX_INITIALIZER \
  { { 0, 0, 0, 0, 0, { 0 } } }
# ifdef __USE_GNU
#  define PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP \
  { { 0, 0, 0, PTHREAD_MUTEX_RECURSIVE_NP, 0, { 0 } } }
#  define PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP \
  { { 0, 0, 0, PTHREAD_MUTEX_ERRORCHECK_NP, 0, { 0 } } }
#  define PTHREAD_ADAPTIVE_MUTEX_INITIALIZER_NP \
  { { 0, 0, 0, PTHREAD_MUTEX_ADAPTIVE_NP, 0, { 0 } } }
# endif
#endif
```

# *Mutex*: atributos

***pthread_mutexattr_t*** (dependem da implementação):

- ***Protocol***: *Specifies the protocol used to prevent priority inversions for a mutex.*
- ***Prioceiling***: determina o **teto de prioridade** (***priority ceiling***) de um *mutex*.
  - *pthread_mutex_getprioceiling( )*
  - *pthread_mutex_setprioceiling( )*
- ***Process-shared***: trata do **compartilhamento** de um *mutex*.
  - **PTHREAD_PROCESS_SHARED** permite que um ***mutex*** seja manipulado por qualquer processo que tem acesso à **memória** em que ele está alocado, mesmo que em em memória compartilhada por múltiplos processos.
  - **PTHREAD_PROCESS_PRIVATE** (valor *default*) faz com que um mutex possa ser manipulado apenas por threads associadas ao mesmo processo.
  - *int pthread_mutexattr_getpshared( )*
  - *int pthread_mutexattr_setpshared( )*

  - *int pthread_mutexattr_getrobust ( )*
  - *int pthread_mutexattr_getrobust_np ( )*
  - *int pthread_mutexattr_setrobust ( )*
  - *int pthread_mutexattr_setrobust_np ( )*

# *Mutex*: bloqueio e liberação

- *int* ***pthread_mutex_lock*** *(pthread_mutex_t *__mutex)*
  - É usada para bloquer uma variável *mutex.*
  - *Thread* é bloqueada se mutex já está bloqueado.
  - *"Only mutex itself may be used for performing synchronization. The result of referring to copies of mutex in calls to pthread_mutex_xxx is undefined".*
- *int* ***pthread_mutex_trylock*** *(pthread_mutex_t *__mutex)*
  - Tenta bloquear um *mutex*.
  - Caso já esteja bloqueado, erro *busy* é retornado.
  - É comumente usada para prevenir *deadlocks*.
- ***int pthread_mutex_timedlock (pthread_mutex_t *__restrict __mutex,***
  - Tenta bloquear um *mutex*.
  - Retorna sucesso caso *mutex* esteja liberado
  - Caso *mutex* já esteja bloqueado, bloqueia *thread* até que seja liberado ou até que o ***timeout*** especificado como parâmetro expire.
  - Caso ***timeout*** expire, erro ETIMEDOUT é retornado.
- ***int pthread_mutex_unlock (pthread_mutex_t *__mutex)***
  - Libera um *mutex*, se executado pela *thread* que o bloqueou.
  - Erros: *mutex* já está liberado, *mutex* está bloqueado por outra *thread*

# Exclusão Mútua

## Solução usando *mutex*

```
#include <pthread.h>
...
// mutex para controle de de acesso a variável compartilhada
pthread_mutex_t soma_value_lock;

int soma_global;
...

void *soma(void *list_ptr){
   int *partial_list_ptr, soma_local, i;
   partial_list_ptr = (int *)list_ptr;
   for (i = 0; i < values_per_thread; i++){
      pthread_mutex_lock(&values_per_thread);
      soma_global+= partial_list_ptr[i];
      pthread_mutex_unlock(&values_per_thread);
   }
  ...
}
int  main(int argc, char *argv[]) {
   pthread_mutex_init(&soma_value_lock, NULL);
   // cria múltiplas threads
   …
}
```

Serializa acesso à variável compartilhada: perda de paralelismo...

# Produtor-Consumidor com Mutex

- *Thread* produtor **produz** dados e os insere em *buffer* compartilhado
- *Thread* **consumidor** retira dados do buffer e executa
- Restrições
  - Produtor não deve sobrescrever *buffer* enquanto dado anterior não tiver sido retirado pelo consumidor
  - Consumidor deve aguardar presença de dado no *buffer*

# *Overhead* devido ao bloqueio

- *Mutexes* forçam a serialização
  - Uma *thread* de cada vez executa a seção crítica
- Seções críticas devem breves
- *Overhead* pode ser reduzido aumentando a sobreposição de fases de computação e de espera

```
int pthread_mutex_trylock(pthread_mutex_t
  *mutex_lock)
```

  - Obtém o semáforo caso esteja livre
  - Retorna EBUSY caso esteja bloqueado
  - *Thread* pode fazer outra coisa se *mutex* estiver indisponível

# Variáveis de Condição

- Permitem que um *thread* bloqueie a si mesmo até que uma condição seja satisfeita
- Evento é sinalizado através do estado da variável de condição
- Pode haver diversas *threads* bloqueadas aguardando condição
  - Quando a variável de condição for sinalizada, uma ou mais *threads* serão desbloqueadas
- Uma variável de condição pode estar associada a vários predicados, mas comumente está associada a apenas **um predicado** e a um *mutex*
  - *Mutex* serve para que o teste da variável seja uma operação atômica

# Sincronização com Variáveis de Condição

- Função básica de espera:

```
int pthread_cond_wait(pthread_cond_t *cond,
      pthread_mutex_t *mutex);
```

- Chamar essa função faz com que *thread* seja bloqueada até que a condição seja satisfeita
  - *Thread* bloqueia *mutex* indicado e testa a variável de condição
  - Se a condição for FALSA, *thread* é bloqueada
    - Mas antes deve liberar o *mutex* para que outra *thread* possa ser executada e ajustar o valor da variável de condição

- Quando *thread* tornar a condição verdadeira
  - Essa *thread* deve sinalizar a variável de condição para
    - acordar uma única *thread* em espera, ou
    - acordar todas as *threads* em espera
  - Quando *thread* liberar *mutex* ele será passado para próxima *thread* que estiver esperando

# Variáveis condicionais

- *Mutex* provêm sincronização no acesso a dados
- **Variáveis condicionais** (*condition variables*) provêm sincronização baseada no **valor** de alguma **variável**

- Sem variáveis condicionais, programa deve fazer uma **varredura** constante para verificar se condição está satisfeita.
- Uma variável condicional é usada junto com um *mutex* para bloquear uma *thread* em função do valor de alguma condição.

Uso típico:

*Main thread*
- Declara e inicializa variáveis globais que requerem acesso sincronizado (ex. *count*)
- Declara e inicializa uma *condition variable*
- Declara e inicializa um *mutex*
- Cria *threads* A e B

*Thread A*
- Executa até o ponto em que uma condição deva ser satisfeita (Ex. *count* deve ter valor específico)
- Bloqueia *mutex* associado à *conditional variable* e verifica o valor da variável global (*count*)
- Executa *pthread_cond_wait*() para esperar de maneira bloqueante até que condição seja sinalizada pela thread B. Chamada a *pthread_cond_wait* automaticamente desbloqueia o *mutex* de maneira atômica, que agora pode ser usado por B.
- Quando receber a indicação, é acordada. *Mutex* é bloqueado de maneira atômica.
- Explicitamente libera o *mutex* (*phtread_mutex_unlock*)
- Continua

*Thread B*
- Executa
- Bloqueia *mutex*
- Altera o valor da variável global monitorada pela *trhead* A.
- Verifica o valor da variável global esperada pela thread A. Se valor for satisfeito, sinaliza (*pthread_cond_signal*) *thread* A
- Libera o *mutex*.
- Continua

*Main thread*
- *join / Continue*

# Variáveis condicionais

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
int ***pthread_cond_init***(pthread_cond_t *cond, pthread_condattr_t *cond_attr);
int ***pthread_cond_signal***(pthread_cond_t *cond);
int ***pthread_cond_broadcast***(pthread_cond_t *cond);
int ***pthread_cond_wait***(pthread_cond_t *cond, pthread_mutex_t *mutex);
int ***pthread_cond_timedwait***(pthread_cond_t *cond, pthread_mutex_t *mutex, const struct timespec *abstime);
int ***pthread_cond_destroy***(pthread_cond_t *cond);

***pthread_cond_wait***()
- Bloqueia a *thread* que executa a chamada até que a condição especificada seja sinalizada.
- Deve ser executada enquanto o mutex está bloqueado.
- *Mutex* associado é automaticamente liberado enquanto a condição testada não está satisfeita.
- Quando sinal é recebido e *thread* é desbloqueada (acordada), *mutex* é bloqueado automaticamente para uso pela *thread*.
- Programador deve incluir chamada explícita para desbloquear mutex ao final da seção crítica (depois de chamar pthread_cond_signal())

***pthread_cond_signal***()
- Sinaliza (ou acorda) outra *thread* que estiver esperando pela variável condicional.
- Deve ser executada enquanto o *mutex* está bloqueado e deve liberar o *mutex* para que a função *pthread_cond_wait* seja completada.

***pthread_cond_broadcast***()
- Deve ser usada (ao invés de *pthread_cond_signal*) para liberar mais de uma *threads* bloqueadas à espera de variável de condição.

# Produtor-Comsumidor com Variáveis de Condição (main)

```
pthread_cond_t cond_queue_empty, cond_queue_full;
pthread_mutex_t task_queue_cond_lock;
int task_available;
…
main() {
…
task_available = 0;
pthread_init();
pthread_cond_init(&cond_queue_empty, NULL);
pthread_cond_init(&cond_queue_full, NULL);
pthread_mutex_init(&task_queue_cond_lock, NULL);
/* create and join producer and consumer threads */
}
```

# Produtor

```
void *producer(void *producer_thread_data)
{
  int inserted;
  while (!done()) {
     create_task();
     pthread_mutex_lock(&task_queue_cond_lock);
     while (task_available == 1)
        pthread_cond_wait(&cond_queue_empty,
           &task_queue_cond_lock);
     insert_into_queue();
     task_available = 1;
     pthread_cond_signal(&cond_queue_full);
     pthread_mutex_unlock(&task_queue_cond_lock);
  }
}
```

Libera *mutex* ao entrar no *wait*

Readquire o *mutex* ao acordar

# Consumidor

```
void *consumer(void *consumer_thread_data)
{
   while (!done()) {
      pthread_mutex_lock(&task_queue_cond_lock);
      while (task_available == 0)
         pthread_cond_wait(&cond_queue_full,
            &task_queue_cond_lock);
      my_task = extract_from_queue();
      task_available = 0;
      pthread_cond_signal(&cond_queue_empty);
      pthread_mutex_unlock(&task_queue_cond_lock);
      process_task(my_task);
   }
}
```

# *Semaphores*

- POSIX 1003.1b ***semaphores***
- Diferentes da implementação de semáforos da API SystemV (***ipc***(5), ***semctl***(2) and ***semop***(2)).
- ***Semaphores*** são **contadores** para recursos compartilhados entre ***threads***.
- Função: sincronizar *threads* de maneira bloqueante (ou não).
- Operações básicas:
  - Incrementar o contador de maneira atômica.
  - Esperar até que o contador tenha valor diferente de 0 e decrementá-lo de maneira atômica.

# Semaphores

*int* ***sem_init*** *(sem_t *sem, int pshared, unsigned int value);*

- Inicia SEM. PSHARED indica se será compartilhado com outros processos.

*int* ***sem_destroy*** *(sem_t *sem);*

- Libera os recursos associados ao semóforo SEM.

sem_t ***sem_open*** (const char *name, int oflag, ...);

- Abre um *named semaphore* NAME com *flags* OFLAG.

*int* ***sem_close*** *(sem_t *sem);*

- Fecha o descritor de um *named semaphore* SEM.

*int* ***sem_unlink*** *(const char *name);*

- Remove o *named semaphore* NAME.

*int* ***sem_wait*** *(sem_t *sem);*

- Espera até que o semáforo SEM esteja liberado.

*int* ***sem_timedwait*** *(sem_t *restrict sem, const struct timespec *restrict abstime);*

- Similar a `sem_wait' mas limita espera até ABSTIME.

*int* ***sem_trywait*** *(sem_t *sem);*

- Testa se SEM está liberado.

*int* ***sem_post*** *(sem_t *sem);*

- Libera SEM.

*int* ***sem_getvalue*** *(sem_t *restrict sem, int *restrict sval);*

- Obtém o valor atual de SEM e o armazena em *SVAL.

# Barriers

- Barreiras permitem a sincronização de diversas *threads* em um ponto de execução
- Uma vez iniciada a barreira (*pthread_barrier_init*), execução das *threads* só prossegue quando o número estipulado de *threads* tiver chamado a função de sincronização (*pthread_barrier_wait*)

# Barreiras

*int* ***pthread_barrier_init*** *(pthread_barrier_t *barrier, const pthread_barrierattr_t *attr, unsigned count);*

- Inicia BARRIER com os atributos definidos por ATTR. Barreira é aberta quando COUNT *threads* tiverem atingido esse ponto.

*int* ***pthread_barrier_destroy*** *(pthread_barrier_t *barrier);*

- Destrói a barreira BARRIER.

*int* ***pthread_barrier_wait*** *(pthread_barrier_t *barrier);*

- Espera bloqueante na barreira BARRIER

*int* ***pthread_barrierattr_init*** *(pthread_barrierattr_t *attr);*

- Inicia o atributo de barreira ATTR

*int* ***pthread_barrierattr_destroy*** *(pthread_barrierattr_t *attr);*

- Destrói o atritubo de barreira ATTR.

*int* ***pthread_barrierattr_getpshared*** *(const pthread_barrierattr_t *attr, int *pshared);*

- Obtém o valor do *flag process-shared* do atributo de barreira ATTR

*int* ***pthread_barrierattr_setpshared*** *(pthread_barrierattr_t *attr, int pshared);*

- Ajusta o *flag process-shared* do atributo de barreira ATTR.