



Threads



Conceito de *Thread*

- “Linha” de execução de um processo. Cada *thread* executa uma parte do programa.
 - Este mecanismo permite a execução concorrente de múltiplas tarefas dentro do mesmo processo.
 - Recursos do processo são partilhados por todas as *threads*: memória, ficheiros abertos.
- Exemplo: processador de texto XYZ.
 - Uma *thread* para leitura de dados e actualização do ecrã
 - Uma *thread* para executar o corrector ortográfico
 - Uma *thread* para cada pedido de impressão.
 - etc.
- Linux: man 7 pthreads



Criação de *threads*

- `#include <pthread.h>`
- `int pthread_create(
pthread_t *tidp,
const pthread_attr_t *attr,
void *(*start_rtn)(void *),
void *arg);`
 - `tidp` – usado pelo `pthread_create()` para armazenar o identificador (*handle*) da thread.
 - `attr` – atributos da thread. Com NULL são usados os defaults.
 - `start_rtn` – apontador para a função a ser executada na thread.
 - `arg` – apontador para os parâmetros a passar à função `start_rtn`

pthread_create





Exemplo 1 – impressão simultânea de 'x's e 'o's

```
void myprint(char c) {
    while (1)
        fputc (c, stderr);
}

void* thread_main(void* arg) {
    myprint('x');
    return NULL;
}

int main () {
    pthread_t tid;

    pthread_create(&tid, NULL, thread_main, NULL);

    myprint('o');
    return 0;
}
```



Terminação de *threads*

- Execução numa *thread* termina nas seguintes situações:
 - Quando a função principal da *thread* termina.
 - Quando é executada a função **pthread_exit()** nessa *thread*.



Terminação de threads

- `void pthread_exit(void *rval_ptr);`
 - *rval_ptr* é um apontador para a posição de memória onde estão armazenados os dados a serem retornados pela *thread*
 - passar NULL, no caso de não ser necessário retornar informação
 - NÃO deverão ser retornados apontadores para variáveis automáticas (locais).
 - Caso uma *thread* execute a função `exit()`, o processo termina, incluindo todas as suas restantes *threads*.



Terminação de threads

- Por *default*, a *thread* fica no estado “zombie”:
 - Termina execução, mas mantém em memória o valor de retorno e bloqueia o identificador da thread
 - Acumulação de threads zombie pode impedir a criação de novas threads.
- Duas alternativas para evitar *threads* “zombie”:
 - Usar função **pthread_join()**, para ler o valor de retorno da thread.
 - **pthread_join()** é uma função bloqueante.
 - Usar função **pthread_detach()**, para indicar ao sistema que não se pretende ler o valor de retorno da *thread*.
 - Função não bloqueante, pode ser chamada a qualquer altura.



Aguardar pela conclusão de uma *thread*

- `int pthread_join(pthread_t thread, void **rval_ptr);`
 - *rval_ptr* deverá indicar o endereço de um apontador onde, por sua vez, será guardado o endereço da posição de memória onde estão armazenados os dados retornados pela *thread*. Caso não se pretenda recolher essa informação, deverá ser passado a constante NULL.
 - Caso a *thread* esteja no estado *detached* (através dos atributos passados a *pthread_create* ou da chamada a *pthread_detach*) a função falha.

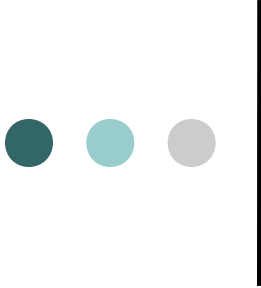
Exemplo 2 - impressão simultânea de 30k 'x's e 20k 'o's

```
void myprint2(char c, int n) {  
    for(int i=0;i<n;++i)  
        fputc(c, stderr);  
}
```

```
void* tm_print(void* args){  
    myprint2(?, ?);  
    return NULL;  
}
```

```
int main (){  
    pthread_t tid1, tid2;  
    pthread_create (&tid1, NULL, tm_print, ?);  
    pthread_create (&tid2, NULL, tm_print, ?);  
    //impedir que o processo termine antes de ambas  
    //as threads terminarem  
    pthread_join(tid1, NULL);  
    pthread_join(tid2, NULL);  
    return 0;  
}
```

Mesma função executada em simultâneo, com diferentes argumentos.



Exemplo 2 (cont.)

```
typedef struct {  
    char c;  
    int n;  
} args_t;
```

```
int main () {  
    pthread_t tid1, tid2;  
    args_t t1_args, t2_args;  
    t1_args.c = 'x';  
    t1_args.n = 30000;  
    pthread_create (&tid1, NULL, tm_print, &t1_args);  
    t2_args.c = 'o';  
    t2_args.n = 20000;  
    pthread_create (&tid2, NULL, tm_print, &t2_args);  
  
    pthread_join(tid1, NULL);  
    pthread_join(tid2, NULL);  
    return 0;  
}  
  
void* tm_print(void* args) {  
    args_t* p = (args_t*) args;  
    myprint2(p->c, p->n);  
    return NULL;  
}
```



Modo *detached*

- `int pthread_detach(pthread_t thread);`
 - Thread não ficará *zombie*.
 - Deixa de ser possível aguardar pela conclusão da thread usando o `pthread_join`.



Identificação de threads

- `pthread_t pthread_self(void);`
 - Obtém o handle da *thread* onde a função é executada.
 - Por exemplo, se uma *thread* quiser ficar no estado detached pode fazer `pthread_detach(pthread_self())`
- `int pthread_equal(pthread_t t1, pthread_t t2);`
 - Verifica se os identificadores de threads se referem à mesma *thread*.



Cancelamento de threads

- Uma *thread* pode:
 - permitir o seu cancelamento em qualquer ponto do programa (cancelamento assíncrono).
 - permitir o seu cancelamento apenas em determinadas partes do programa (correspondentes à execução de determinadas funções). Este é o comportamento *default* (cancelamento síncrono).
 - Não permitir o cancelamento.
- Resultados podem ser pouco previsíveis (principalmente no caso assíncrono).
 - E.g., cancelamento durante uma escrita.



Cancelamento de threads (cont.)

- `int pthread_setcancelstate(int state, int *oldstate);`
 - ENABLE/DISABLE
- `int pthread_setcanceltype(int type, int *oldtype);`
 - DEFERRED/ASYNCHRONOUS
- `int pthread_cancel(pthread_t tid);`
- Qualquer *thread* pode solicitar o cancelamento (terminação) de outra *thread*.



Sinais e threads

- Aquando da entrega de um sinal, o sistema escolhe arbitrariamente uma das threads que não o tenham bloqueado.
- Cada *thread* tem a sua própria máscara de sinais.
 - Cada *thread* herda a máscara de sinais da *thread* que a criou.
- Existem funções para fazer um tratamento “mais fino” dos sinais para programas *multi-thread*.
 - `int pthread_sigmask(int how, const sigset_t *newmask, sigset_t *oldmask);`
 - `int pthread_kill(pthread_t thread, int signo);`
 - `int sigwait(const sigset_t *set, int *sig);`
 - Para usar esta função, deve-se bloquear os sinais em todas as outras *threads*



Threads vs processos

- Ambos permitem multi-tarefa.
- Criação de *threads* e comutação entre diferentes *threads* é em geral mais rápida.
 - No entanto, os sistemas Unix podem usar o método copy-on-write para minimizar o tempo de criação do processo, para evitar que o *fork()* tenha que copiar imediatamente todos os dados do processo pai (só são “copiados” à medida que são alterados).
- Maior eficiência no intercâmbio de dados (e sincronização) entre *threads* vs maior probabilidade de um erro numa *thread* afetar drasticamente todas as outras.
 - Para o bem e para o mal, todas as *threads* acedem à mesma memória, a memória do processo (cf. com cópia de variáveis feita pelo *fork()*).



Threads e Processos: analogias

Processes Threads

fork pthread_create
(create a new flow of control)

exit pthread_exit
(exit from an existing flow of control)

waitpid pthread_join
(get exit status from flow of control)

getpid pthread_self
(get ID for flow of control)