CS 3113

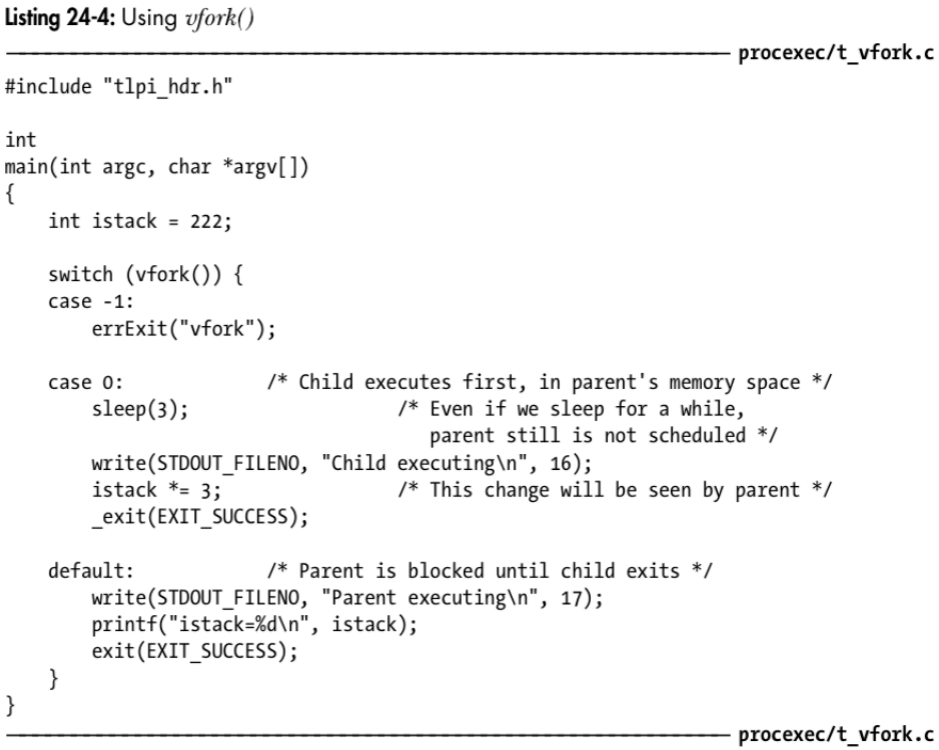
Clone + Pthreads

Spring 2020

Outline

- vfork
- Clone
- Pthreads

vfork()

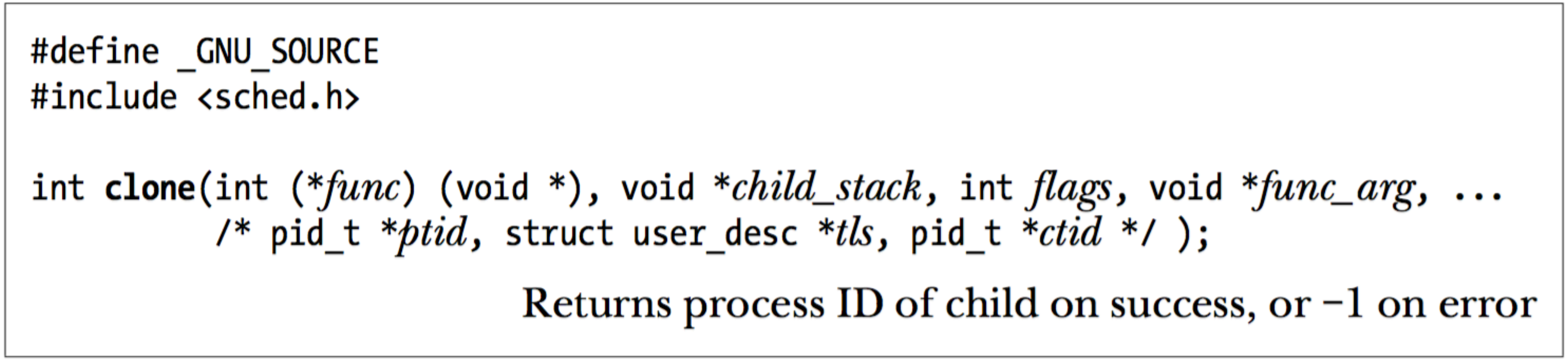


Listing 24-4: Using *vfork()*

```
————————————————————————————————————————————————— procexec/t_vfork.c
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int istack = 222;

    switch (vfork()) {
    case -1:
        errExit("vfork");

    case 0:             /* Child executes first, in parent's memory space */
        sleep(3);               /* Even if we sleep for a while,
                                   parent still is not scheduled */
        write(STDOUT_FILENO, "Child executing\n", 16);
        istack *= 3;            /* This change will be seen by parent */
        _exit(EXIT_SUCCESS);

    default:            /* Parent is blocked until child exits */
        write(STDOUT_FILENO, "Parent executing\n", 17);
        printf("istack=%d\n", istack);
        exit(EXIT_SUCCESS);
    }
}
————————————————————————————————————————————————— procexec/t_vfork.c
```

- Unlike fork(), no duplication of virtual memory (page tables)
- Parents memory is shared until exec or exit are called
- Any changes to the stack or heap of the parent are seen in the parent
- The child of vfork() is guaranteed to be called.

Clone



```
#define _GNU_SOURCE
#include <sched.h>

int clone(int (*func) (void *), void *child_stack, int flags, void *func_arg, ...
          /* pid_t *ptid, struct user_desc *tls, pid_t *ctid */ );
```

Returns process ID of child on success, or –1 on error

Listing 28-3: Using *clone()* to create a child process

procexec/t_clone.c

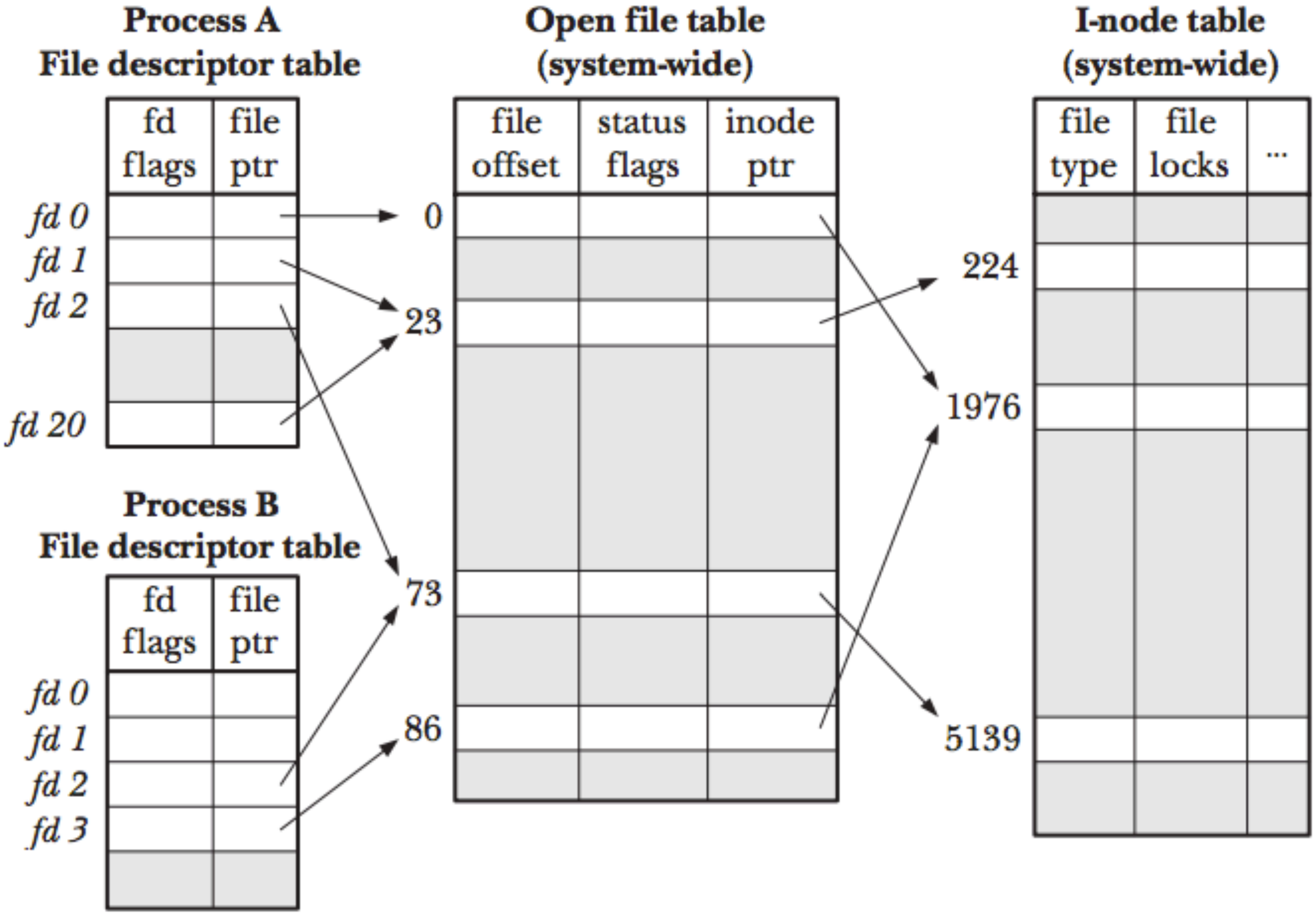


Figure 5-2: Relationship between file descriptors, open file descriptions, and i-nodes

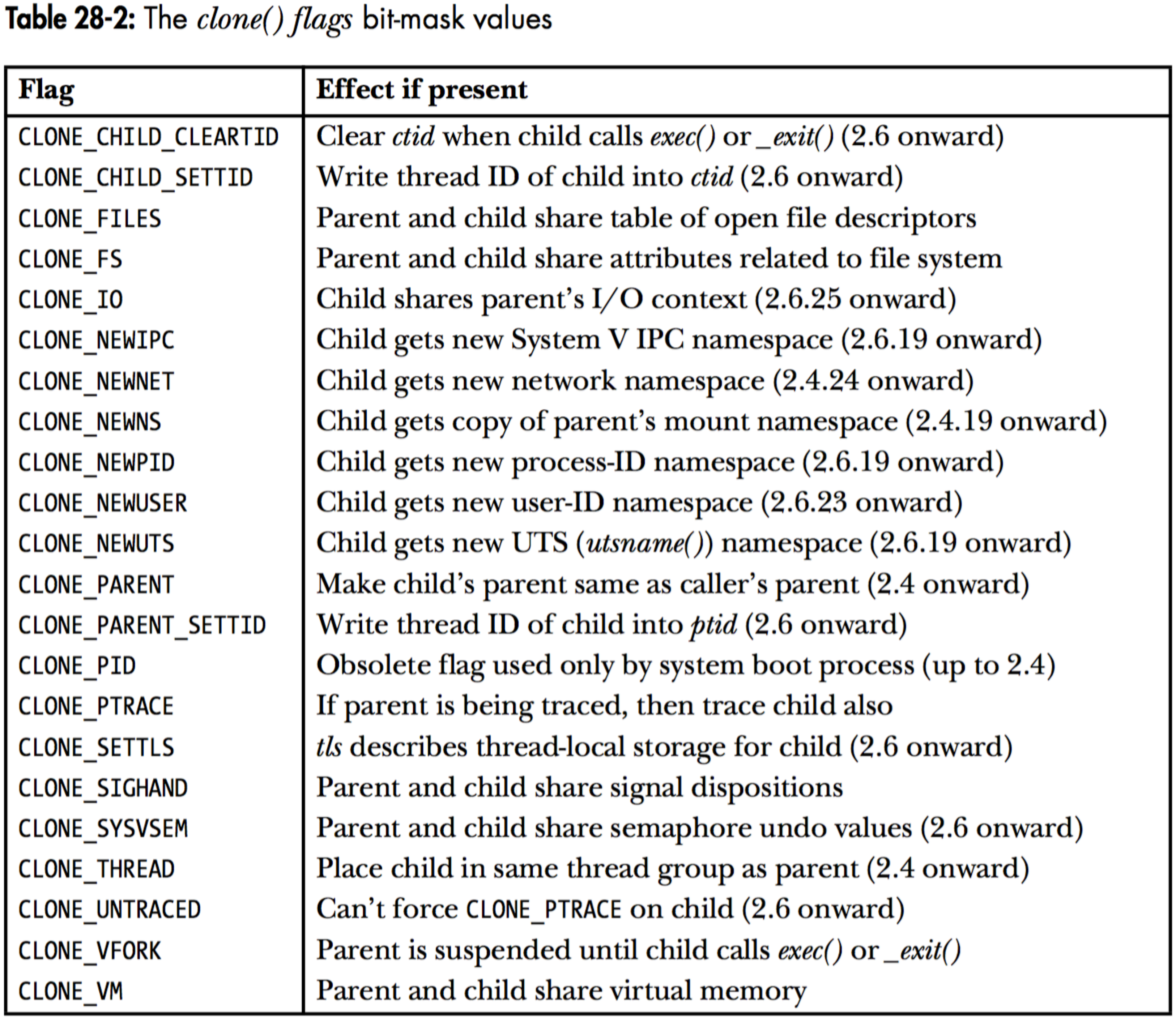


Table 28-2: The *clone() flags* bit-mask values

Flag	Effect if present
`CLONE_CHILD_CLEARTID`	Clear *ctid* when child calls *exec()* or *_exit()* (2.6 onward)
`CLONE_CHILD_SETTID`	Write thread ID of child into *ctid* (2.6 onward)
`CLONE_FILES`	Parent and child share table of open file descriptors
`CLONE_FS`	Parent and child share attributes related to file system
`CLONE_IO`	Child shares parent's I/O context (2.6.25 onward)
`CLONE_NEWIPC`	Child gets new System V IPC namespace (2.6.19 onward)
`CLONE_NEWNET`	Child gets new network namespace (2.4.24 onward)
`CLONE_NEWNS`	Child gets copy of parent's mount namespace (2.4.19 onward)
`CLONE_NEWPID`	Child gets new process-ID namespace (2.6.19 onward)
`CLONE_NEWUSER`	Child gets new user-ID namespace (2.6.23 onward)
`CLONE_NEWUTS`	Child gets new UTS (*utsname()*) namespace (2.6.19 onward)
`CLONE_PARENT`	Make child's parent same as caller's parent (2.4 onward)
`CLONE_PARENT_SETTID`	Write thread ID of child into *ptid* (2.6 onward)
`CLONE_PID`	Obsolete flag used only by system boot process (up to 2.4)
`CLONE_PTRACE`	If parent is being traced, then trace child also
`CLONE_SETTLS`	*tls* describes thread-local storage for child (2.6 onward)
`CLONE_SIGHAND`	Parent and child share signal dispositions
`CLONE_SYSVSEM`	Parent and child share semaphore undo values (2.6 onward)
`CLONE_THREAD`	Place child in same thread group as parent (2.4 onward)
`CLONE_UNTRACED`	Can't force `CLONE_PTRACE` on child (2.6 onward)
`CLONE_VFORK`	Parent is suspended until child calls *exec()* or *_exit()*
`CLONE_VM`	Parent and child share virtual memory

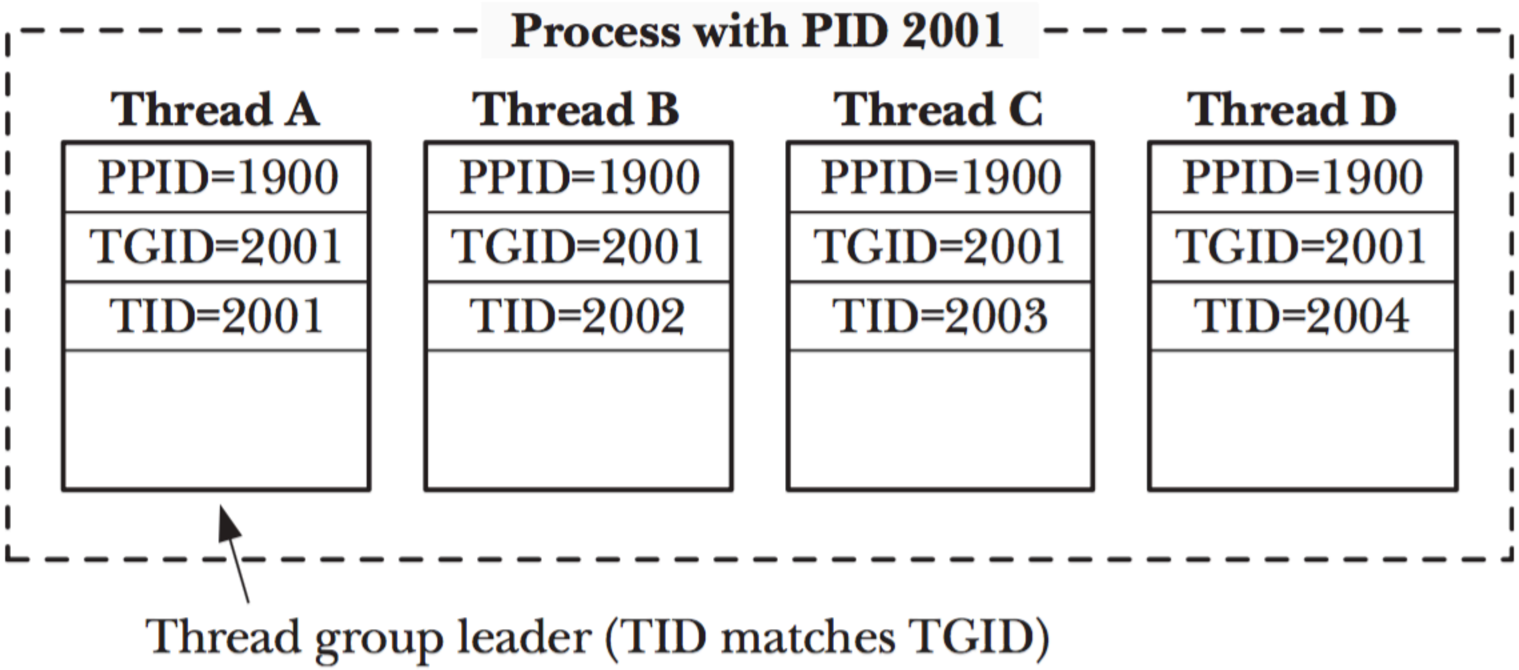


Figure 28-1: A thread group containing four threads

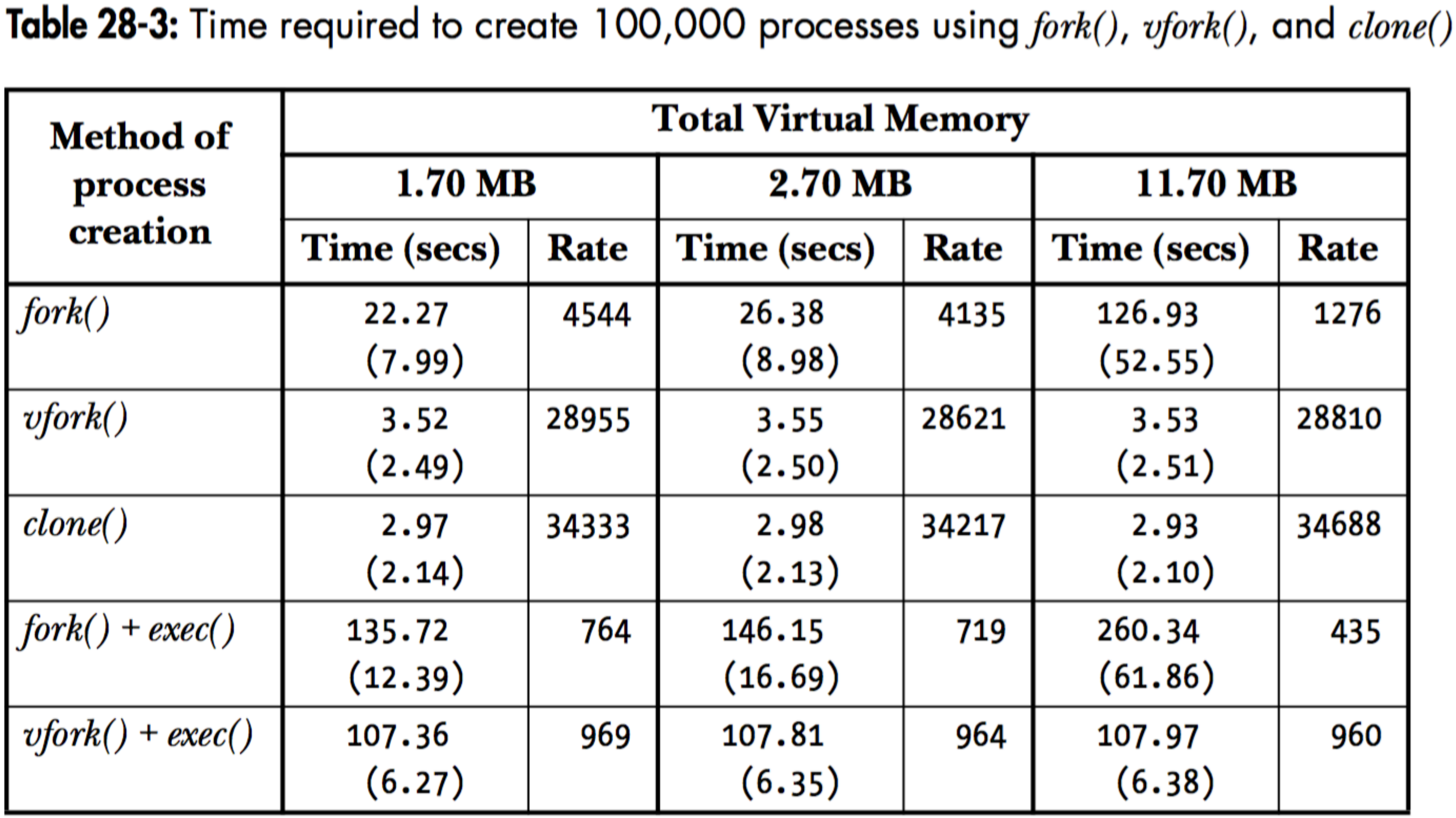


Table 28-3: Time required to create 100,000 processes using *fork()*, *vfork()*, and *clone()*

Method of process creation	Total Virtual Memory					
	1.70 MB		2.70 MB		11.70 MB	
	Time (secs)	Rate	Time (secs)	Rate	Time (secs)	Rate
fork()	22.27 (7.99)	4544	26.38 (8.98)	4135	126.93 (52.55)	1276
vfork()	3.52 (2.49)	28955	3.55 (2.50)	28621	3.53 (2.51)	28810
clone()	2.97 (2.14)	34333	2.98 (2.13)	34217	2.93 (2.10)	34688
fork() + *exec()*	135.72 (12.39)	764	146.15 (16.69)	719	260.34 (61.86)	435
vfork() + *exec()*	107.36 (6.27)	969	107.81 (6.35)	964	107.97 (6.38)	960

Thread stack

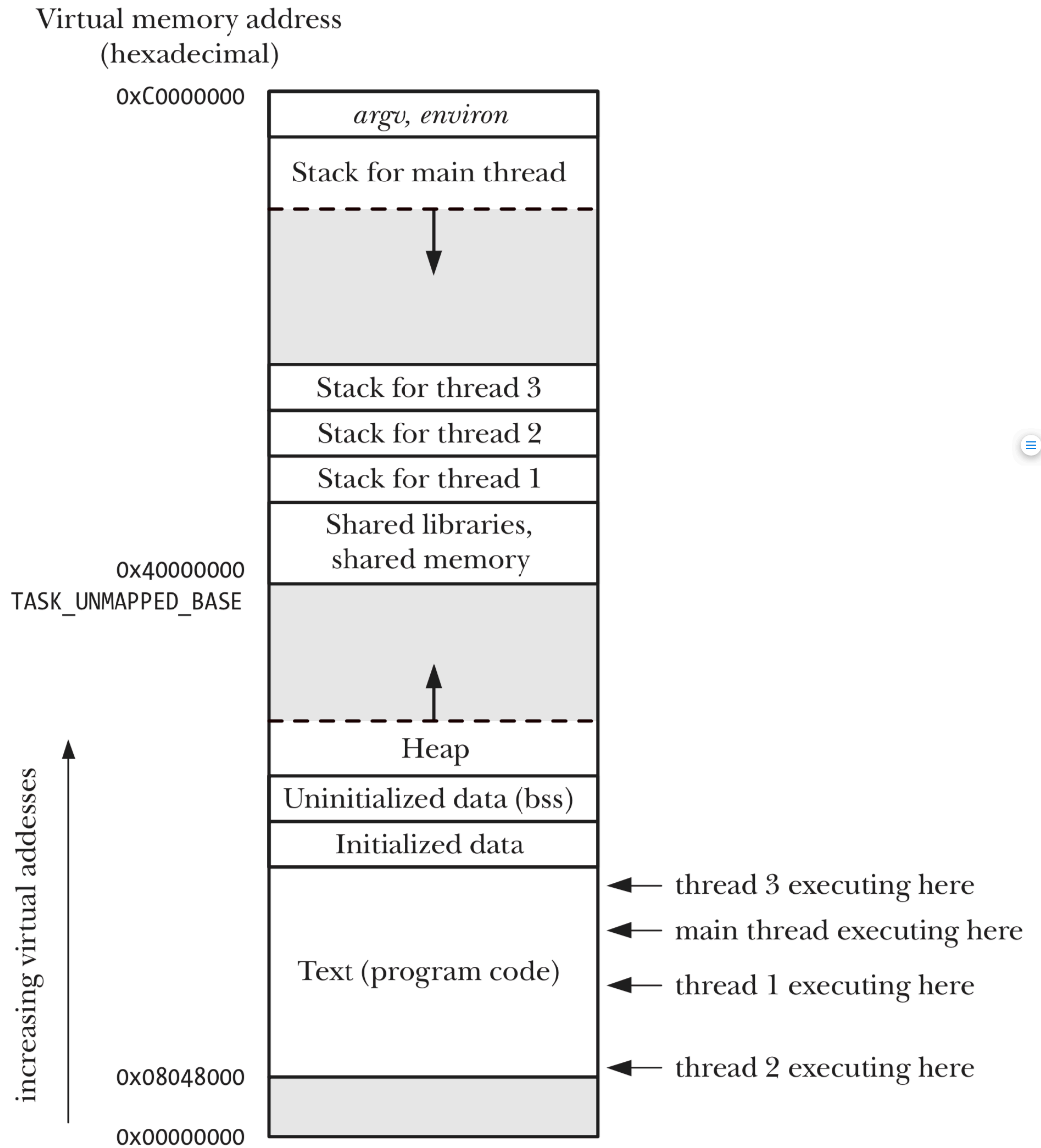


Figure 29-1: Four threads executing in a process (Linux/x86-32)

Pthreads

POSIX threads.

Standardized in 95 as part of SUSv3

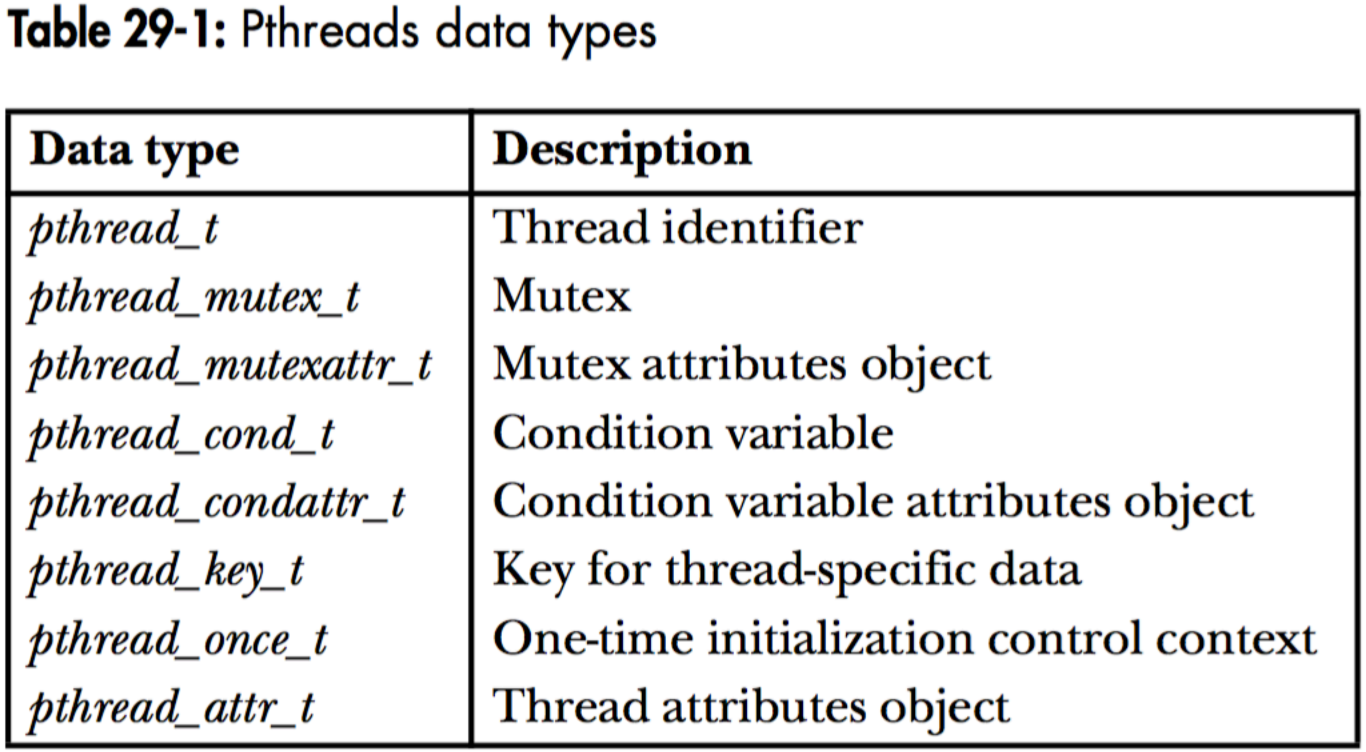
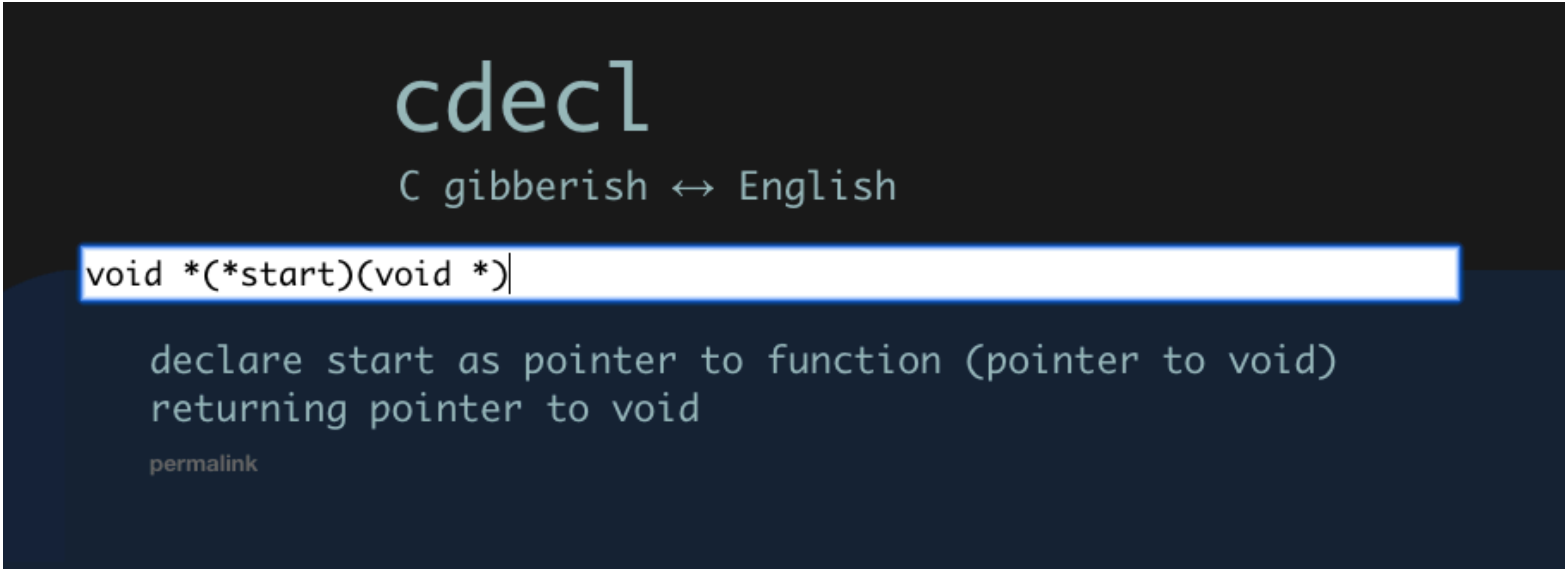


Table 29-1: Pthreads data types

Data type	Description
pthread_t	Thread identifier
pthread_mutex_t	Mutex
pthread_mutexattr_t	Mutex attributes object
pthread_cond_t	Condition variable
pthread_condattr_t	Condition variable attributes object
pthread_key_t	Key for thread-specific data
pthread_once_t	One-time initialization control context
pthread_attr_t	Thread attributes object

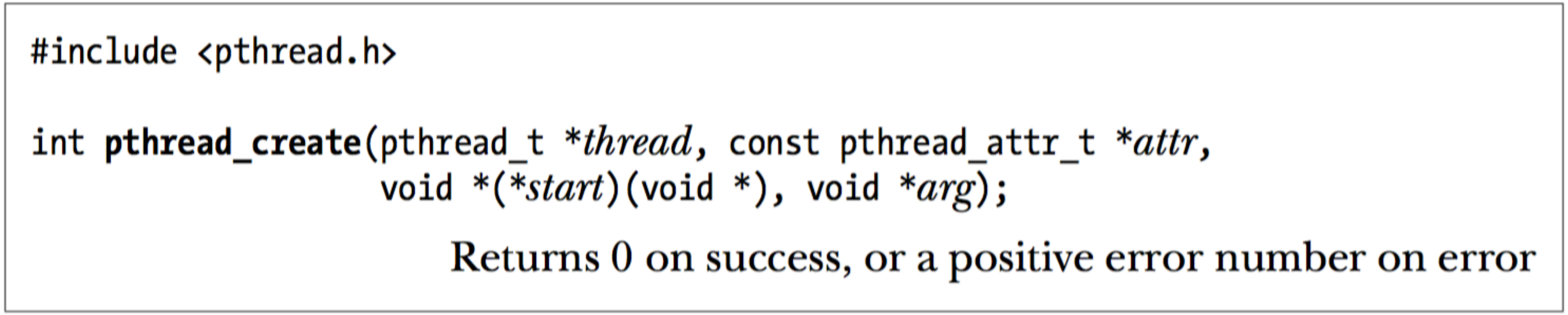


```
void *(*start)(void *)
```

declare start as pointer to function (pointer to void) returning pointer to void

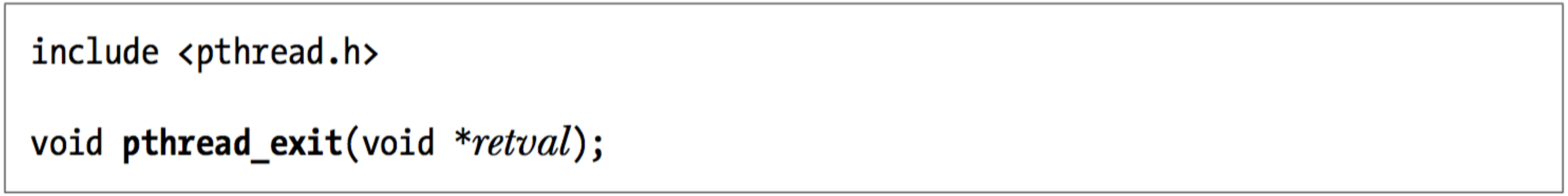
http://parrt.cs.usfca.edu/doc/**how-to-read-C-declarations.html**

https://cdecl.org/

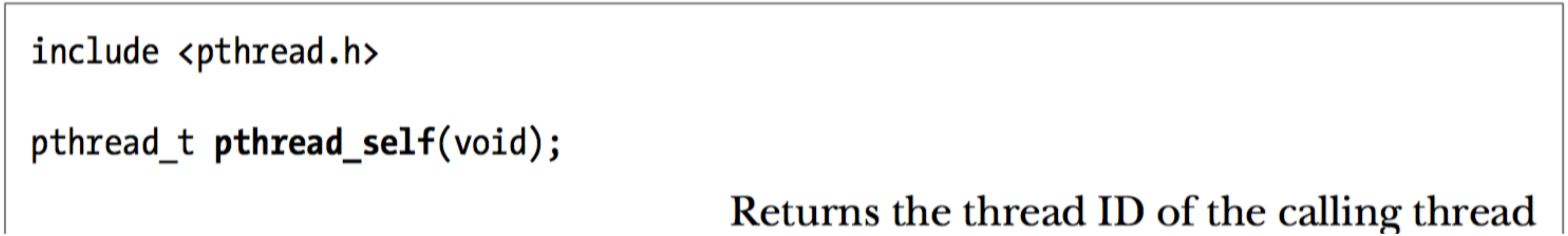


```
#include <pthread.h>

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

Returns 0 on success, or a positive error number on error

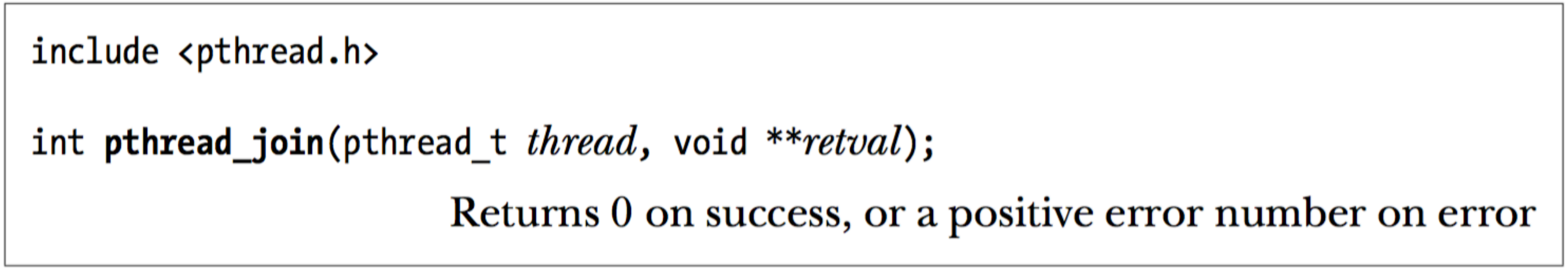


```
include <pthread.h>

void pthread_exit(void *retval);
```



```
include <pthread.h>

pthread_t pthread_self(void);
```

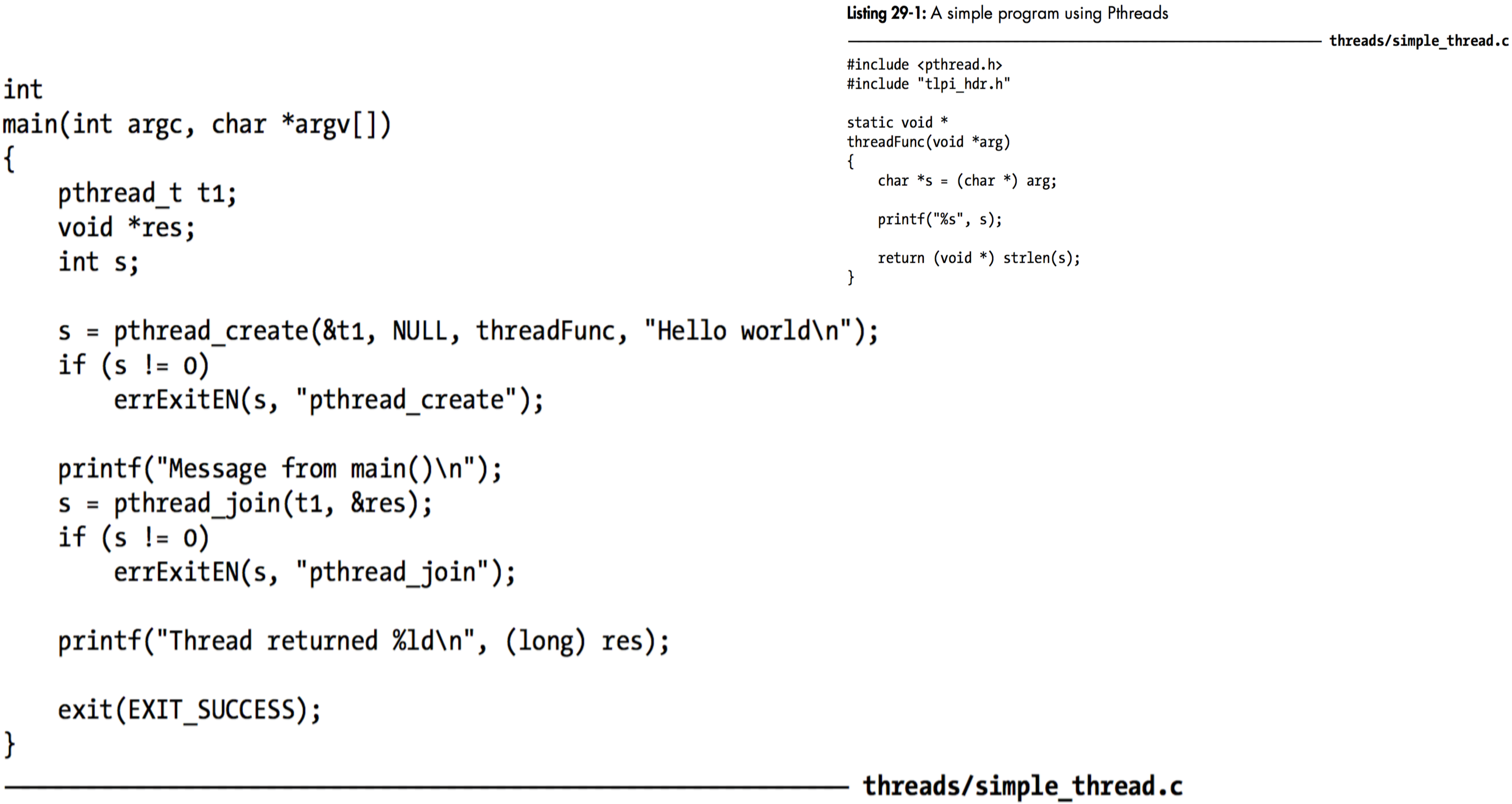
Returns the thread ID of the calling thread



```
include <pthread.h>

int pthread_join(pthread_t thread, void **retval);
```

Returns 0 on success, or a positive error number on error

Ch 29



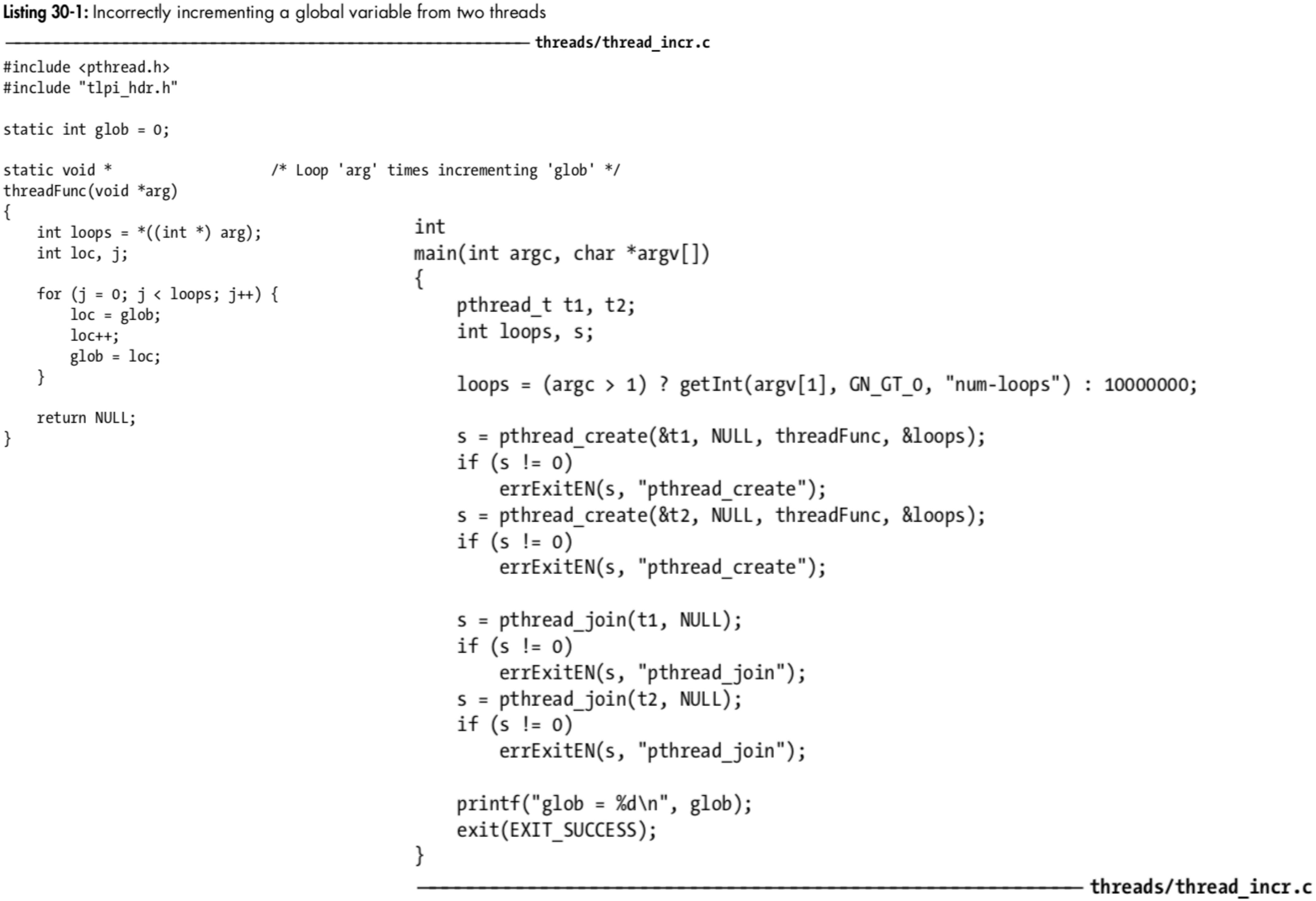
Listing 29-1: A simple program using Pthreads

threads/simple_thread.c

```
#include <pthread.h>
#include "tlpi_hdr.h"

static void *
threadFunc(void *arg)
{
    char *s = (char *) arg;

    printf("%s", s);

    return (void *) strlen(s);
}
```

```
int
main(int argc, char *argv[])
{
    pthread_t t1;
    void *res;
    int s;

    s = pthread_create(&t1, NULL, threadFunc, "Hello world\n");
    if (s != 0)
        errExitEN(s, "pthread_create");

    printf("Message from main()\n");
    s = pthread_join(t1, &res);
    if (s != 0)
        errExitEN(s, "pthread_join");

    printf("Thread returned %ld\n", (long) res);

    exit(EXIT_SUCCESS);
}
```

threads/simple_thread.c



Listing 30-1: Incorrectly incrementing a global variable from two threads

```
——————————————————————————————————————————————————————— threads/thread_incr.c
#include <pthread.h>
#include "tlpi_hdr.h"

static int glob = 0;

static void *                   /* Loop 'arg' times incrementing 'glob' */
threadFunc(void *arg)
{
    int loops = *((int *) arg);
    int loc, j;

    for (j = 0; j < loops; j++) {
        loc = glob;
        loc++;
        glob = loc;
    }

    return NULL;
}
```

```
int
main(int argc, char *argv[])
{
    pthread_t t1, t2;
    int loops, s;

    loops = (argc > 1) ? getInt(argv[1], GN_GT_0, "num-loops") : 10000000;

    s = pthread_create(&t1, NULL, threadFunc, &loops);
    if (s != 0)
        errExitEN(s, "pthread_create");
    s = pthread_create(&t2, NULL, threadFunc, &loops);
    if (s != 0)
        errExitEN(s, "pthread_create");

    s = pthread_join(t1, NULL);
    if (s != 0)
        errExitEN(s, "pthread_join");
    s = pthread_join(t2, NULL);
    if (s != 0)
        errExitEN(s, "pthread_join");

    printf("glob = %d\n", glob);
    exit(EXIT_SUCCESS);
}
——————————————————————————————————————————————————————— threads/thread_incr.c
```

Threads vs Processes

- Sharing between threads is easy.
- Sharing between processes required pipes or shared memory.

- In multi threading, shared variables need to be “thread-safe”
- A bug in one thread could corrupt memory in all the threads.

- Multi-threaded require careful design.