POSIX Threads

System Programming

2019 여름 계절학기

한양대학교 공과대학 컴퓨터소프트웨어학부
홍석준

Motivation

❑ Monitoring file descriptors

- A separated process
  - Children do not share any variables
- select(), poll()
  - Blocking calls
- Nonblocking I/O with polling
  - Sometimes forces hard-coding of the timing for I/O check
- POSIX asynchronous I/O
  - The handler use only async-signal-safe funcitons
- A separate thread
  - Simpler than the other approaches

Why thread?

- ❑ **There are many reasons to program with threads. In the context of this class, there are two important ones:**
  - They allow you to deal with asynchronous events efficiently.
  - They allow you to get parallel performance on a shared-memory multiprocessor.
- ❑ **You'll find threads to be a big help in writing an operating system.**

What are threads?

- ❑ **Each thread is a unit of execution, which consists of a stack and CPU state(i.e, registers)**
- ❑ **Multiple threads resemble**
  - – Multiple processes, except that multiple threads within a task use the same code, globals and heap
- ❑ **Thus, while two processes in Unix can only communicate through the operating system (e.g. through files, pipes or sockets).**
- ❑ **Two threads in a task can communicate through memory.**

What are threads?

- ❑ **When you program with threads, you assume that they execute simultaneously.**
  - In other words, it should appear to you as if each thread is excuting on its own CPU, and that all the threads share the same memory.

What are threads?

- ❑ **On a single processor, multithreading generally occurs by time-division multiplexing (as in multitasking)**
  - The processor switches between different threads
  - Time shared and multiprocessor threading with a process scheduler.
- ❑ **On a multiprocessor or multi-core system the threads or task actually do run at the same time, with each processor or core running a particular thread or task.**
- ❑ **Many modern operating system directly support both**
  - time sliced and multiprocessor threading with a process scheduler.
  - The operating system kernel allows programmers to manipulate threads via the system call interface

Processes vs. threads

- **Threads are distinguished from traditional multitasking operating system processes in that processes.**
  - are typically independent.
  - carry considerable state information
  - have separate address spaces, and
  - interact only through system-provided inter-process communication mechanisms.

Processes vs. threads

- ❑ **A process is the heaviest unit of kernel scheduling.**
- ❑ **Processes own resources allocated by the operating system.**
  - – Resources include memory, file descriptors, sockets.
- ❑ **Processes do not share address spaces or file resources**
  - – Except through explicit methods such as inheriting file descriptors or shared memory segments or mapping the same file in a shared way.
- ❑ **Processes are typically preemptively multitasked.**

Thread management

- **POSIX Thread functions**
  - pthread_cancel
  - pthread_create
  - pthread_detach
  - pthread_equal
  - pthread_exit
  - pthread_join
  - pthread_self
- **Most functions return 0 if successful and a nonzero error if unsuccessful**
- **None of the POSIX thread functions returns EINTR and they do not have to be restarted if interrupted.**

Referencing threads by ID

```
#include <pthread.h>
pthread_t pthread_self(void);
int pthread_equal(pthread_t tid1,pthread_t tid2);
```

❑ **Pthread_self**

- Find out its own ID

❑ **Pthread_equal**

- Pthread_t may be a structure
- Returns nonzero value if they equal, 0 otherwise

Creating a thread

```
#include <pthread.h>
int pthread_create(pthread_t *restrict tidp,
  const pthread_attr_t *restrict attr, void
  *(*start_rtn)(void *), void * restrict arg);
```

- ❑ **Automatically makes the thread runnable without requiring a separate start operation**
- ❑ **Parameters**
  - “Thread” : the ID of the newly created thread
  - “attr” : represents an attribute object NULL for default attributes.
  - “start_routine” : the name of a function
  - “arg” : a single parameter taken by “start_routine”
- ❑ **Return values**
  - 0 if successful nonzero if unsuccessful

Detaching

```
#include <pthread.h>
int pthread_detach(pthread_t tid);
```

❑ Pthread_detach

- If a thread is not a detached thread. It does not release its resource when it exits
- This function sets a thread's interenal options to specify that storage for the thread can be reclaimed when the thread exits
- Detached thread do not report their status when they exit.
- Return 0 if successful, nonzero if unsuccessful.

Joining

```
#include <pthread.h>
int pthread_join(pthread_t thread, void
   **rval_ptr);
```

- ❑ **pthread_join**
  - Suspends the calling the target thread terminates
  - A nondetached thread resources are not released until another thread calls pthread_join or the entire processes exit
- ❑ **Parameters**
  - “thread” : a target thread
  - “rval_ptr” : a location for a pointer to the return status. If NULL the caller does not retrieve the status
- ❑ **Return values**
  - 0 if successful, nonzero if unsuccessful

Exiting

```
#include <pthread.h>
void pthread_exit(void *rval_ptr);
```

❑ pthread_exit

- Causes the calling thread to terminate
  - Difference form process exit() ?
- ‘return’ implicitly calls pthread_exit
- ‘value_ptr’ : available to successful pthread_join

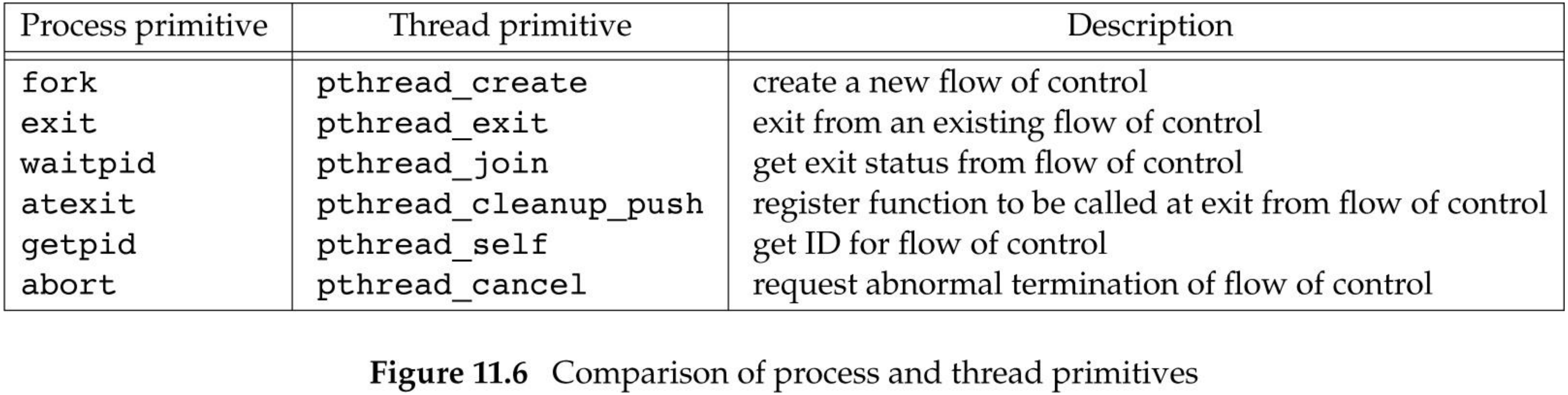
Cancellation

```
#include <pthread.h>
int pthread_cancel(pthread_t tid);
```

❑ **pthread_cancel**

- Requests that another thread be canceled
- Does not cause the caller to block while the cancelation completes
- Return 0 if successful, nonzero if unsuccessful
- The result depends on the target thread's state and type
  - PTHREAD_CANCEL_ENABLE : receives the request
  - PTHREAD_CANCEL_DISABLE : the request is held pending

Process vs Thread

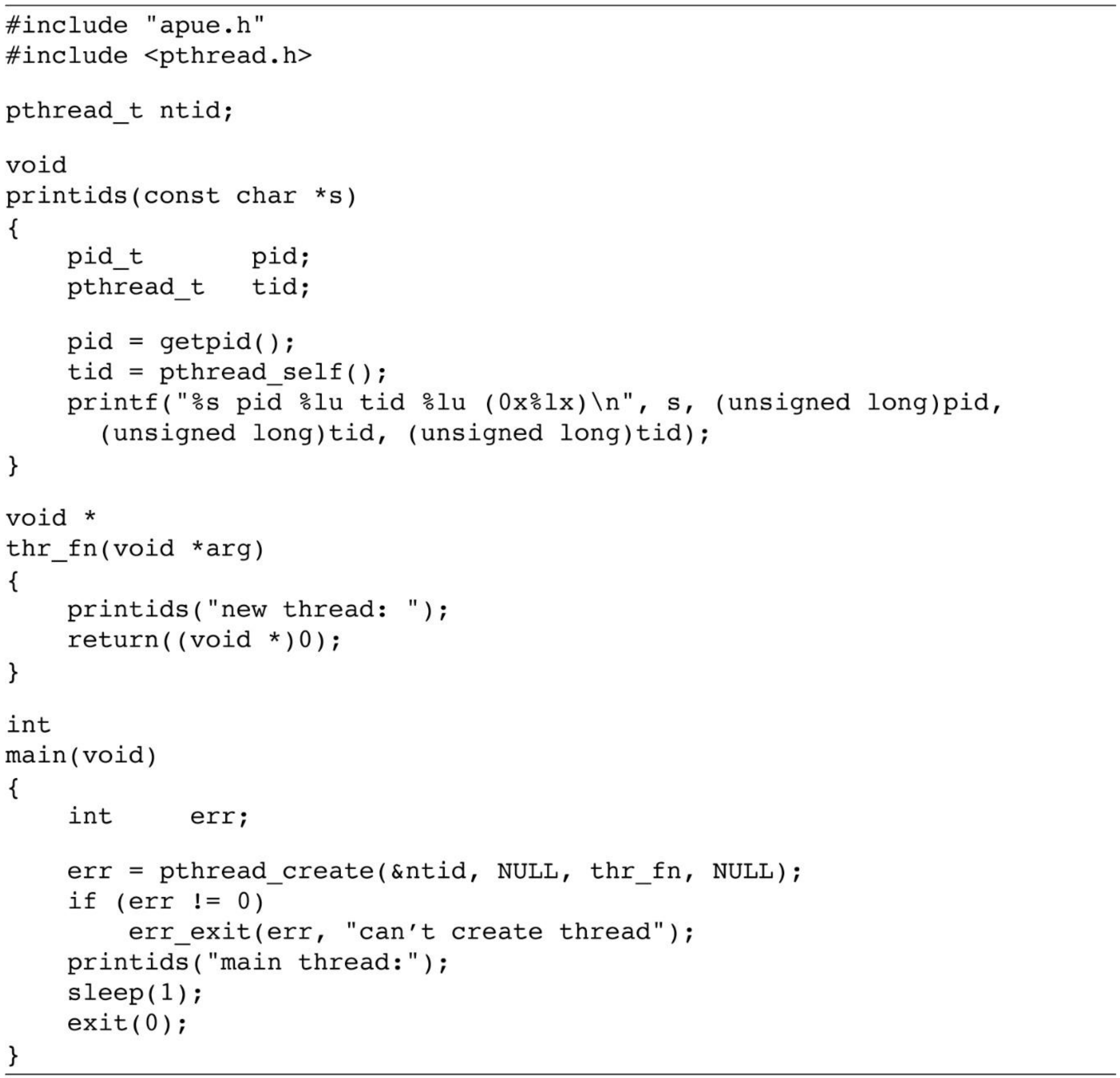


Process primitive	Thread primitive	Description
`fork`	`pthread_create`	create a new flow of control
`exit`	`pthread_exit`	exit from an existing flow of control
`waitpid`	`pthread_join`	get exit status from flow of control
`atexit`	`pthread_cleanup_push`	register function to be called at exit from flow of control
`getpid`	`pthread_self`	get ID for flow of control
`abort`	`pthread_cancel`	request abnormal termination of flow of control

Figure 11.6 Comparison of process and thread primitives

Examples of thread program

- **Prog 11.2**

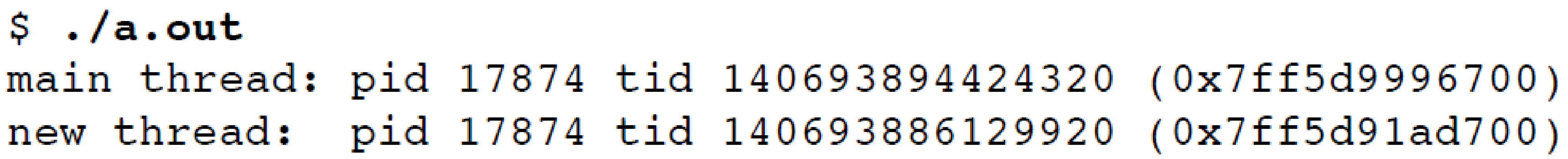


```
#include "apue.h"
#include <pthread.h>

pthread_t ntid;

void
printids(const char *s)
{
    pid_t       pid;
    pthread_t   tid;

    pid = getpid();
    tid = pthread_self();
    printf("%s pid %lu tid %lu (0x%lx)\n", s, (unsigned long)pid,
      (unsigned long)tid, (unsigned long)tid);
}

void *
thr_fn(void *arg)
{
    printids("new thread: ");
    return((void *)0);
}

int
main(void)
{
    int     err;

    err = pthread_create(&ntid, NULL, thr_fn, NULL);
    if (err != 0)
        err_exit(err, "can’t create thread");
    printids("main thread:");
    sleep(1);
    exit(0);
}
```

Figure 11.2 Printing thread IDs

Examples of thread program

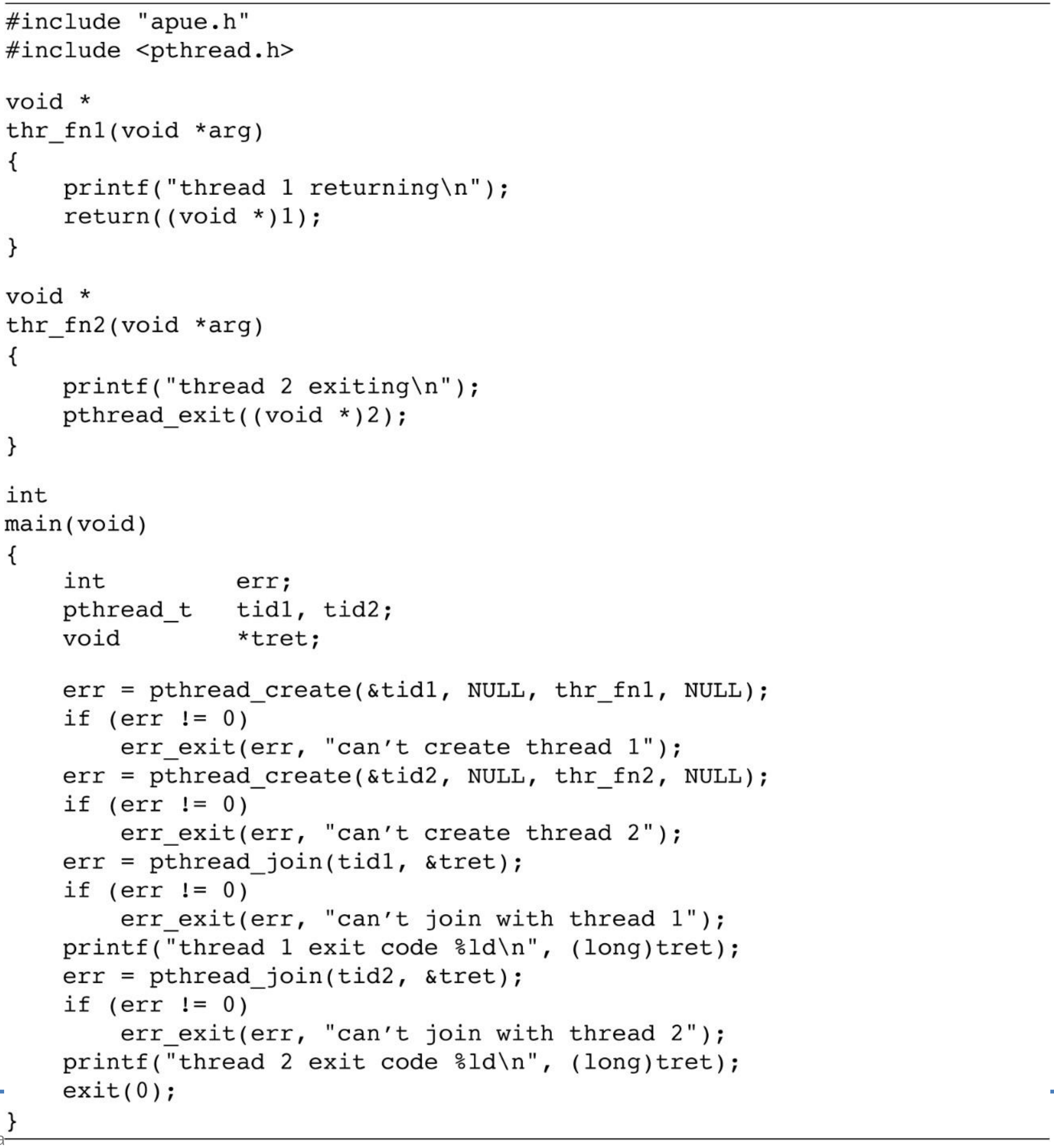
- **Prog 11.2 실행결과**



```
$ ./a.out
main thread: pid 17874 tid 140693894424320 (0x7ff5d9996700)
new thread:  pid 17874 tid 140693886129920 (0x7ff5d91ad700)
```

Examples of thread program

- **Prog 11.3**

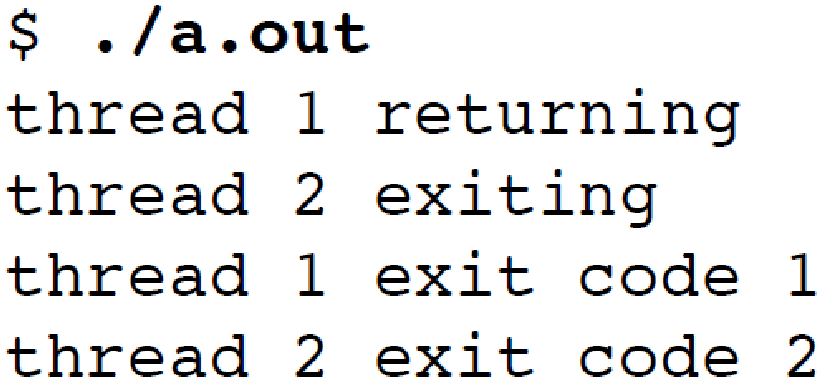


```
#include "apue.h"
#include <pthread.h>

void *
thr_fn1(void *arg)
{
    printf("thread 1 returning\n");
    return((void *)1);
}

void *
thr_fn2(void *arg)
{
    printf("thread 2 exiting\n");
    pthread_exit((void *)2);
}

int
main(void)
{
    int         err;
    pthread_t   tid1, tid2;
    void        *tret;

    err = pthread_create(&tid1, NULL, thr_fn1, NULL);
    if (err != 0)
        err_exit(err, "can’t create thread 1");
    err = pthread_create(&tid2, NULL, thr_fn2, NULL);
    if (err != 0)
        err_exit(err, "can’t create thread 2");
    err = pthread_join(tid1, &tret);
    if (err != 0)
        err_exit(err, "can’t join with thread 1");
    printf("thread 1 exit code %ld\n", (long)tret);
    err = pthread_join(tid2, &tret);
    if (err != 0)
        err_exit(err, "can’t join with thread 2");
    printf("thread 2 exit code %ld\n", (long)tret);
    exit(0);
}
```

Figure 11.3 Fetching the thread exit status

Examples of thread program

- ❑ **Prog 11.3 실행결과**



```
$ ./a.out
thread 1 returning
thread 2 exiting
thread 1 exit code 1
thread 2 exit code 2
```

Thank you for your attention !!

Q and A