Creating and Using Threads

CS 475

Traditional View of a Process

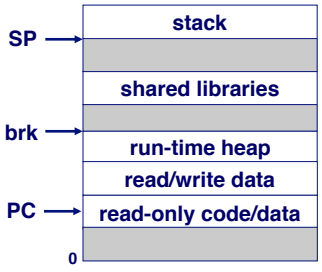
Process = process context + code, data, and stack

Process context

Program context:
- Data registers
- Condition codes
- Stack pointer (SP)
- Program counter (PC)

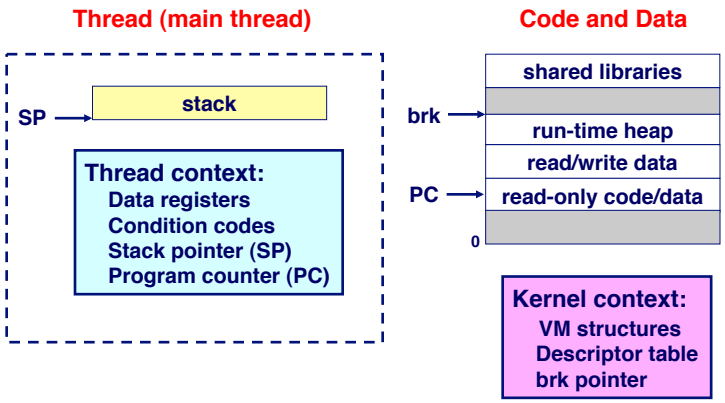
Kernel context:
- VM structures
- Descriptor table
- brk pointer

Code, data, and stack



Alternate View of a Process

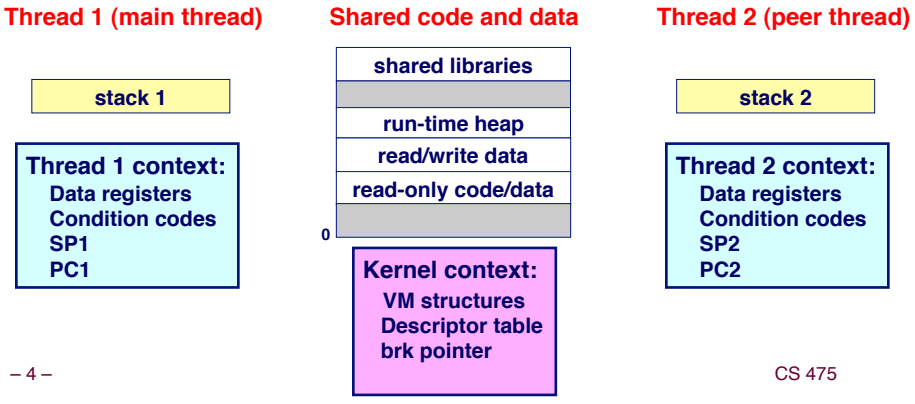
Process = thread + code, data, and kernel context



A Process With Multiple Threads

Multiple threads can be associated with a process

- Each thread has its own logical control flow (sequence of PC values)
- Each thread shares the same code, data, and kernel context
- Each thread has its own thread id (TID)

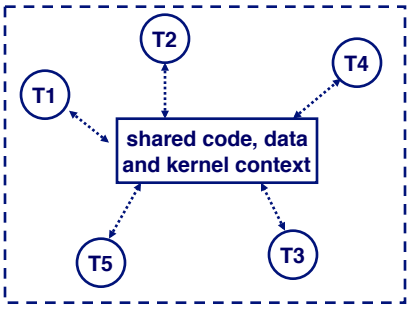


Logical View of Threads

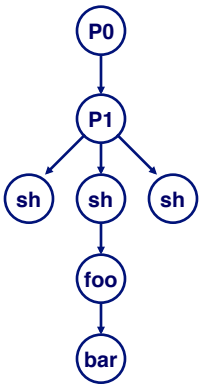
Threads associated with a process form a pool of peers.

- Unlike processes which form a tree hierarchy

Threads associated with process foo



Process hierarchy



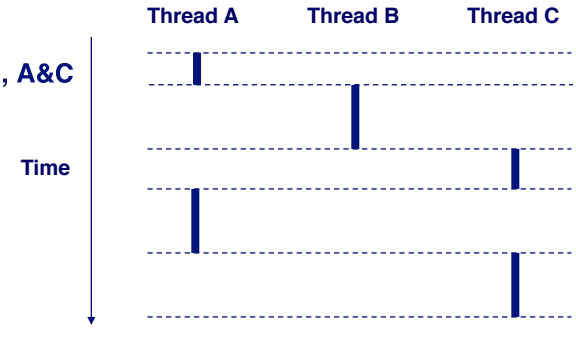
Concurrent Thread Execution

Two threads run concurrently (are concurrent) if their logical flows overlap in time.

Otherwise, they are sequential.

Examples:

- Concurrent: A & B, A&C
- Sequential: B & C



Threads vs. Processes

How threads and processes are similar

- Each has its own logical control flow.
- Each can run concurrently.
- Each is context switched.

How threads and processes are different

- Threads share code and data, processes (typically) do not.
- Threads are somewhat less expensive than processes.
  - Process control (creating and reaping) is twice as expensive as thread control.
  - Linux/Pentium III numbers:
    - » ~20K cycles to create and reap a process.
    - » ~10K cycles to create and reap a thread.

Creating and Using threads

Pthreads Multi-threading Library

- Supported on Linux, MacOS X, Windows
- pthread_create, pthread_join, pthread_self, pthread_exit, pthread_detach

Java

- provides a Runnable interface and a Thread class as part of standard Java libraries
  - users program threads by implementing the Runnable interface or extending the Thread class

Java Threads

Java threads may be created by:

- Extending Thread class
- Implementing the Runnable interface

Java threads are managed by the JVM.

Java thread constructor and management methods

Thread(ThreadGroup group, Runnable target, String name)
Creates a new thread in the *SUSPENDED* state, which will belong to *group* and be identified as *name*; the thread will execute the *run()* method of *target*.

setPriority(int newPriority), getPriority()
Set and return the thread's priority.

run()
A thread executes the *run()* method of its target object, if it has one, and otherwise its own *run()* method (*Thread* implements *Runnable*).

start()
Change the state of the thread from *SUSPENDED* to *RUNNABLE*.

sleep(int millisecs)
Cause the thread to enter the *SUSPENDED* state for the specified time.

yield()
Enter the *READY* state and invoke the scheduler.

destroy()
Destroy the thread.

Creating threads

```
class Simple implements Runnable {
   public void run() {
     System.out.println("this is a thread");
   }
}

Runnable s = new Simple();
Thread t = new Thread(s);
t.start();
```

Alternative strategy: Extend Thread class (not recommended unless you are creating a new type of Thread)

Pthreads

a POSIX standard (IEEE 1003.1c) API for thread creation and synchronization.

API specifies behavior of the thread library, implementation is up to development of the library.

Common in UNIX operating systems.

Posix Threads (Pthreads) Interface

Pthreads: **Standard interface for ~60 functions that manipulate threads from C programs.**

- Creating and reaping threads.
  - `pthread_create`
  - `pthread_join`
- Determining your thread ID
  - `pthread_self`
- Terminating threads
  - `pthread_cancel`
  - `pthread_exit`
  - `exit` [terminates all threads] , `ret` [terminates current thread]
- Synchronizing access to shared variables
  - `pthread_mutex_init`
  - `pthread_mutex_[un]lock`
  - `pthread_cond_init`
  - `pthread_cond_[timed]wait`

The Pthreads "hello, world" Program

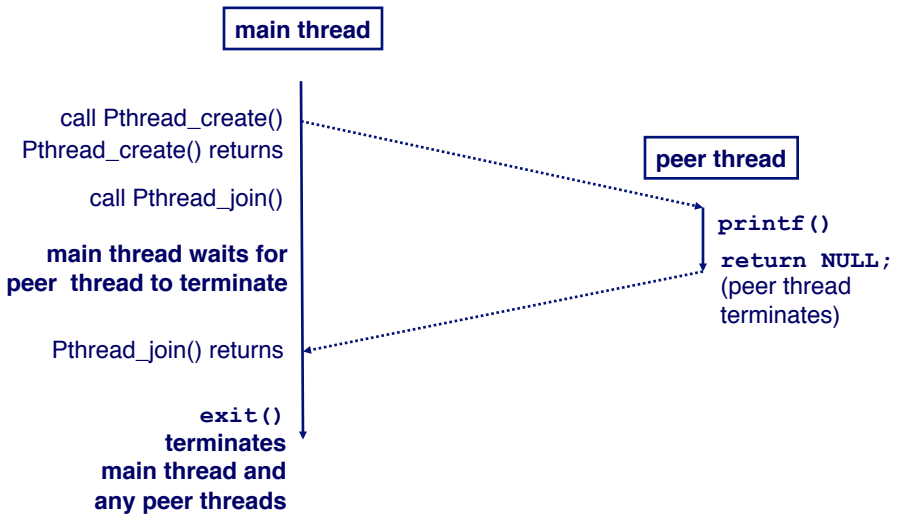
```
/*
 * hello.c - Pthreads "hello, world" program
 */
#include "csapp.h"

void *thread(void *vargp);

int main() {
  pthread_t tid;

  Pthread_create(&tid, NULL, thread, NULL);
  Pthread_join(tid, NULL);
  exit(0);
}

/* thread routine */
void *thread(void *vargp) {
  printf("Hello, world!\n");
  return NULL;
}
```

Thread attributes (usually NULL)

*Thread arguments (void *p)*

*return value (void **p)*

Execution of Threaded“hello, world”



Example

```
/*
 * lifecycle.c
 * Demonstrate the "life cycle" of a typical thread. A thread is
 * created, and then joined.
 */
#include <pthread.h>
#include "errors.h"

/* Thread start routine. */

void *thread_routine (void *arg)
{
    return arg;
}
```

```
main (int argc, char *argv[])
{
    pthread_t thread_id;
    void *thread_result;
    int status;

    status = pthread_create (
        &thread_id, NULL, thread_routine, NULL);
    if (status != 0)
        err_abort (status, "Create thread");
     status = pthread_join (thread_id, &thread_result);
    if (status != 0)
        err_abort (status, "Join thread");
    if (thread_result == NULL)
        return 0;
    else return 1;
}
```


Creating and using threads

- Thread states
  - Ready
  - Running
  - Blocked
  - Terminated
- the main thread is special
- detaching a thread has no impact on a running thread except the system to knows that it can free up resources being used by that thread when it terminates

Example: using threads

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

typedef struct alarm_tag {
    int         seconds;
    char        message[64];
} alarm_t;

void *alarm_thread (void *arg)
{
    alarm_t *alarm = (alarm_t*)arg;
    int status;

    status = pthread_detach (pthread_self ());
```



```
    if (status != 0)
        err_abort (status, "Detach thread");
    sleep (alarm->seconds);
    printf ("(%d) %s\n", alarm->seconds, alarm->message);
    free (alarm);
    return NULL;
}
int main (int argc, char *argv[])
{
    int status;
    char line[128];
    alarm_t *alarm;
    pthread_t thread;

    while (1) {
        printf ("Alarm> ");
        if (fgets (line, sizeof (line), stdin) == NULL) exit (0);
        if (strlen (line) <= 1) continue;
```

```
alarm = (alarm_t*)malloc (sizeof (alarm_t));
if (alarm == NULL)
    errno_abort ("Allocate alarm");

/*
 * Parse input line into seconds (%d) and a message
 * (%64[^\n]), consisting of up to 64 characters
 * separated from the seconds by whitespace.
 */
if (sscanf (line, "%d %64[^\n]",
    &alarm->seconds, alarm->message) < 2) {
    fprintf (stderr, "Bad command\n");
    free (alarm);
} else {
    status = pthread_create (
        &thread, NULL, alarm_thread, alarm);
    if (status != 0)
        err_abort (status, "Create alarm thread");
```


Issues With Thread-Based Servers

Must run "detached" to avoid memory leak.

- At any point in time, a thread is either *joinable* or *detached.*
- *Joinable* thread can be reaped and killed by other threads.
  - must be reaped (with `pthread_join`) to free memory resources.
- *Detached* thread cannot be reaped or killed by other threads.
  - resources are automatically reaped on termination.
- Default state is joinable.
  - use `pthread_detach(pthread_self())` to make detached.

Must be careful to avoid unintended sharing.

All functions called by a thread must be *thread-safe*

- *(next lecture)*

Pros and Cons of Thread-Based Designs

+ Easy to share data structures between threads
  - e.g., logging information, file cache.
+ Threads are more efficient than processes.

--- Unintentional sharing can introduce subtle and hard-to-reproduce errors!
  - The ease with which data can be shared is both the greatest strength and the greatest weakness of threads.
  - (next lecture)