CS 450: Operating Systems
Lecture 5: More Threads

Spring 2014, J. Sasaki
Dept of Computer Science
Illinois Institute of Technology

The Story So Far….

Threads

- Threads have process context.
  - Share resources, faster context switching, simpler communication; can help organize work of process.
- Processes don't share context.
  - Require interprocess communication; longer context switches. Harder to coordinate.…

Using Threads

- Thread library provides API for creating & using threads.
    - POSIX: pthread_create, pthread_join()
    - No specified implementation for POSIX threads — can vary by OS …
- Threading module for python3
    - Provides class Thread

Thread Data

5

Thread Data

- A created thread shares global data with its creator.
- A created thread can have its own local data.
  - In C, use local variables in the task function
  - In python, add attributes to `threading.local()`.)

Lec05_thread1.c:

```
#include <pthread.h>  // pthread_...
#include <stdio.h>
#include <stdlib.h>      // exit

void *task(void *arg);  // prototype

int gv = 1;             // Global variable

int main(void) {
   // retcode = 0 if thread creation succeeded
   pthread_t thd;
   int retcode;

   int my_x;  // Main's local variable
```


7

```
// Create thread and have it run task
//
retcode = pthread_create(&thd, NULL, task, NULL);
if (retcode != 0) {
   fprintf(stderr,
      "Thread creation failed with code %d!!\n",
      retcode );
   exit(1);
}

// Set & print value of local & global variables
//
my_x = 1234;
gv = 5678;
printf("Main:  &my_x = %-14p  my_x = %d\n",
   &my_x, my_x );
printf("Main:  &gv   = %-14p  gv   = %d\n",
   &gv, gv );
```

```
    // Wait for child to finish, then reprint local & global
    // variables (only global variable will have changed).
    //
    pthread_join(thd, NULL);
    printf("Main has waited:\n");
    printf("Main:  &my_x = %-14p  my_x = %d\n", &my_x, my_x);
    printf("Main:  &gv   = %-14p  gv   = %d\n", &gv, gv);
}

// Task run by thread; this one changes its local
// variable and prints its value
//
void *task(void *arg) {
    int my_x;     // local variable
    my_x = 9012;

    printf("Child: &my_x = %-14p  my_x = %d\n", &my_x, my_x);
    printf("Child: &gv   = %-14p  gv   = %d\n", &gv, gv);

    return NULL;
}
```


9

Lec05_thread2.py:

```
import threading
from threading import Thread

gv = 1;   # Global variable

def main():
   my_x = 1234
   global gv
   gv = 5678
   print("Main:  my_x = {}".format(my_x))
   print("Main:  gv   = {}".format(gv))
```

```
    thd = Thread(target=task, \
       args=(), \
       name="child" )
    thd.start()
    thd.join()
    print("Main has waited")
    print("Main:  my_x = {}".format(my_x))
    print("Main:  gv   = {}".format(gv))

# end of main
```

```
# Thread task stores local data in
# threading.local()
#
def task():
   mydata = threading.local()
   mydata.x = 9012;

   print("Child: my_x = {}".format(mydata.x))
   global gv
   gv = 3456
   print("Child: gv   = {}".format(gv))
```

Thread Implementation

Thread Control Block

- Similar to Process Control Block
- Doesn't include info shared by threads of the process
  - Address space
  - List of open files
  - Any CPU state common to all threads
- TCB switches faster than PCB switches

User & Kernel Threads

- User(-level) thread: Thread created by user-level program (e.g. pthread_create).
  - Runs in user mode
- Kernel thread: Thread of the kernel
  - Analogous to process thread — “Multithreaded kernel”
  - Runs in supervisor mode

Who Manages Threads?

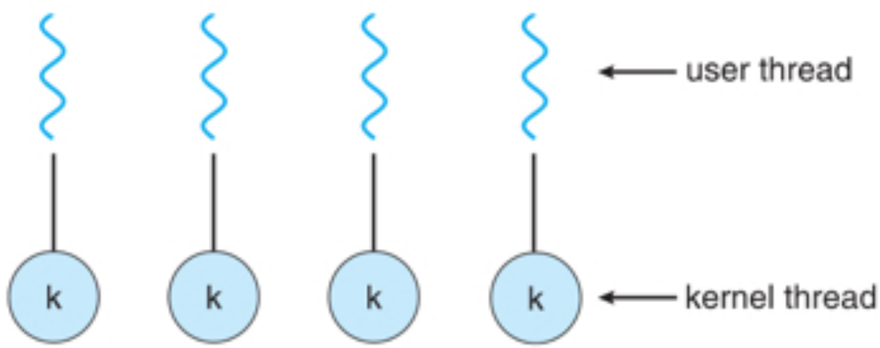
- Kernel(-supported) threads are managed directly by OS
  - Runs user code, in user mode
  - User-level thread scheduled by kernel
  - User thread analogous to a process
- Smart OS scheduler can try to run more threads of program with many threads

User Threads

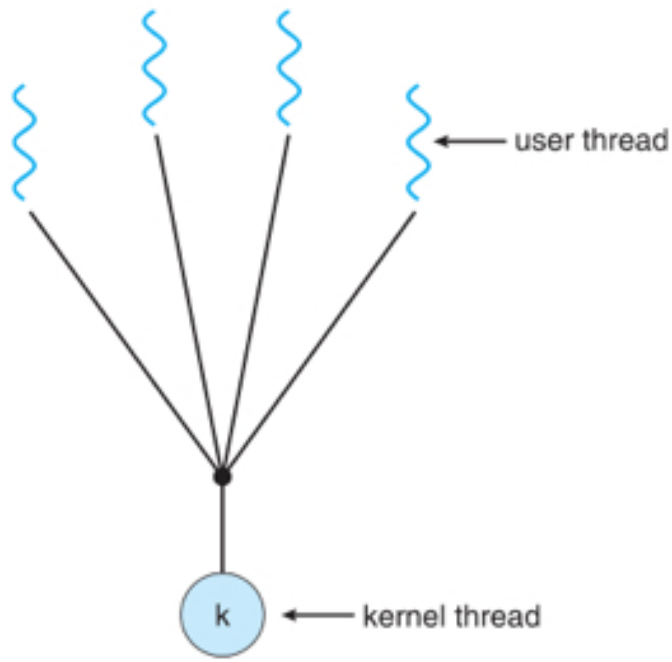
- User threads are managed by user-level library
  - Kernel schedules your process
  - Your process schedules your thread
  - Much less/no special reliance on kernel
  - Duplication of effort vs portability …

One-to-One Thread Model

- User-level thread created and run as a kernel-supported thread
  - Lots of concurrency, possibly parallelism
  - But: Limit on # threads supported by kernel?

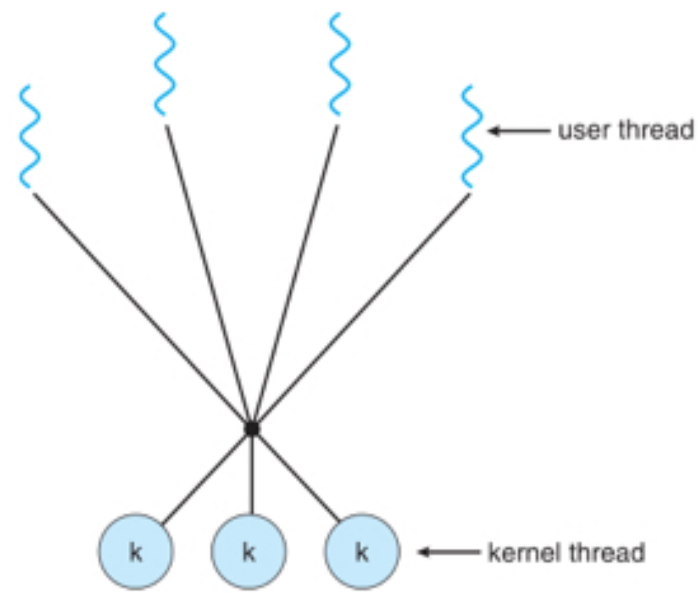


Many-to-One Model



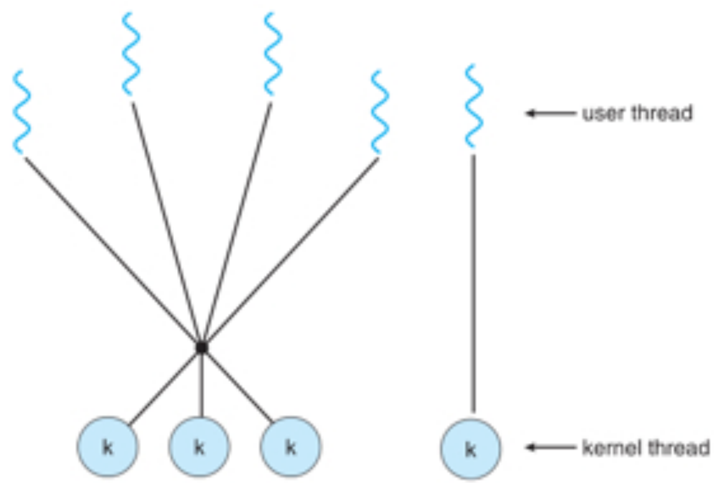
- Multiple user-level threads for 1 kernel thread.
- Kernel thread runs code that schedules user threads
- Less concurrency, parallelism
- Less popular

Many-to-Many Model



- Multiple kernel threads used to run multiple user threads
- Max # kernel threads less of a problem
- More concurrency, parallelism than many-to-one
- Overhead, complexity

Two-Level Model



- Combines 1-to-1 and many-to-many models
- Less popular