Threads (SGG 4)

Instructor: Dr. Tongping Liu

Outline
- Motivation and thread basics
  - Resources requirements: thread vs. process
- Thread implementations/Thread Model
  - User threads: e.g., Pthreads and Java threads
  - Kernel threads: e.g., Linux tasks
  - Map user- and kernel-level threads
- Threading Functions: thread creation/join/exit
- Other multithreaded issues: fork, signals etc.
- Threads Memory Model

Traditional Process: Single Activity
- Multi-programming
  - Multiple processes share the CPU
- One process: address space to hold process image
  - System resources, e.g., files, I/O devices
- Execution states
  - Processor registers
  - Protection mode (user/kernel)
  - Priority
- Single activity ➔ single thread

Example: A Text Editor with Multi-Activity
- Process approach on data
  - P1: read from keyboard
  - P2: format document
  - P3: write to disk
- The processes will access the same set of data.
  - How do the processes exchange data?

Context Switch for Processes - costly
Context Switches of Processes: Expensive

- Context switch between processes
  - Save processor registers for current process
  - Load the new process’s registers
- Switch address spaces – expensive
  - Hardware cache
  - Memory pages (e.g., TLB content)

Ideal Solution for this Example: Threads

- Three activities within one process
  - Single address space
  - Same execution environment
  - Data shared easily
- Switch between activities
  - Only running context
  - No change in address space

Thread vs. Process

- Resource Sharing
  - Memory, open files, etc.
- Performance
  - Creation and switches
- Scalability
  - Increase parallelism

Process: Traditional View

- Process = process context + code, data, and stack

- Code, data, and stack
  - Program context:
    - Data registers
    - Condition codes
    - Stack pointer (SP)
    - Program counter (PC)
  - Kernel context:
    - VM structures
    - Descriptor table
    - Brk pointer

- Stack
  - SP
  - Brk
  - PC
  - Stack
  - Read/write data
  - Read-only code/data
  - Shared libraries
  - Run-time heap
Process: Alternative View

Process = thread + code, data, and kernel context

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

- Code, data, and kernel context
  - shared libraries
  - run-time heap
  - read/write data
  - read-only code/data

- Stack
  - SP
  - PC

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

Process with Two Threads

- Thread 1
  - Program context:
    - Data registers
    - Condition codes
    - Stack pointer (SP)
    - Program counter (PC)
  - shared libraries
  - run-time heap
  - read/write data
  - read-only code/data

- Stack
  - SP
  - PC

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

- Thread 2
  - Program context:
    - Data registers
    - Condition codes
    - Stack pointer (SP)
    - Program counter (PC)
  - shared libraries
  - run-time heap
  - read/write data
  - read-only code/data

- Stack
  - SP
  - PC

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

Resources of Threads

- **Shared** resources among threads
  - Address space (e.g., code, globals, heap)
  - Global variables
  - Opened Files and other resources etc.

- **Separated** resources for each thread
  - Machine states: registers (e.g., PC)
  - Running stacks
  - Private data

Outline

- Motivation and thread basics
  - Resources requirements: thread vs. process
- Thread implementations/Thread Model
  - User threads: e.g., Pthreads and Java threads
  - Kernel threads: e.g., Linux tasks
  - Map user- and kernel-level threads
- Threading Functions: thread creation/join/exit
- Other multithreaded issues: fork, signals etc.
- Threads Memory Model
Thread Implementation

- Process usually starts with a single thread
- Thread management: required operations
  - Creation: procedure/method for the new thread to run
  - Scheduling: runtime properties/attributes
  - Destruction: release resources
- Thread Synchronization
  - join, wait, etc.

Thread Libraries

- Provide programmers with API for creating and managing threads
- Two primary implementations:
  - User-level library
    - Entirely in user space
    - Everything is done using function calls (not system calls)
  - Kernel-level threads
    - Code and data structures for threads are in kernel space
    - Function calls result in system calls to kernel
- Examples:
  - POSIX Threads: Pthreads
  - Java threads (JVM uses host system threads)

Multithreading Model

- Who and where to manage threads
  - User space: managed by applications
  - Kernel space: managed by OS
    - all modern O/Ses have kernel level support
- Thread Modeling:
  - How to map user-level threads to kernel-level threads

Multithreading Models: Many-to-One

- Many user-level threads mapped to a single kernel thread
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads
**Many-to-One Model**

**Pros:**
- Cheap synchronization and cheap thread creation

**Cons:**
- Blocking-problem. A thread calling block system call will block the whole process
- No concurrency.
- Need two levels of scheduling

---

**Multithreading Models: Many-to-Many Model**

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows NT/2000 with the ThreadFiber package

---

**Many-to-Many Model**

**Pros:**
- Cheap Resource, not all user threads should create a kernel thread
- Synchronization mainly at user-level
- Context switch may not involve system calls

**Cons:**
- Difficult cooperation between kernel scheduler and user scheduler
- How to decide the number of kernel threads?

---

**Multithreading Models: One-to-One**

- Each user-level thread maps to kernel thread
- Examples
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 and later

  This is the most popular model
**One-to-one Model**

**Pros:**
- Scalable parallelism (concurrency)
- Thread will not block a whole process

**Cons:**
- Expensive synchronization (system call is required if a lock can't be acquired)
- Expensive creation (3.5 slower)
- Kernel resource, e.g., stack and kernel structure

**One-to-one Threads vs. Processes**

- **Threads and processes: similarities**
  - Each has its own logical control flow
  - Each can run concurrently with others
  - Each is context switched (scheduled) by the kernel

- **Threads and processes: differences**
  - Threads share code and data, processes (typically) do not
  - Threads are less expensive than processes
  - Process control (creation and exit) is more expensive than thread control
  - Context switches: processes are more expensive than for threads

**Pros and Cons of Thread-Based Designs**

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

- **Cons:**
  - Unintentional sharing can introduce subtle and hard-to-reproduce errors!

**Outline**

- Motivation and thread basics
  - Resources requirements: thread vs. process
- Thread implementations/Thread Model
  - User threads: e.g., Pthreads and Java threads
  - Kernel threads: e.g., Linux tasks
  - Map user- and kernel-level threads
- Threading Functions: thread creation/join/exit
- Other multithreaded issues: fork, signals etc.
- Threads Memory Model
Pthreads: POSIX Thread

- POSIX
  - Portable Operating System Interface [for Unix]
  - Standardized programming interface
- Pthreads
  - Thread implementations adhering to POSIX standard
  - APIs specify behavior of the thread library: defined as a set of C types and procedure calls
  - Common in UNIX OS (Solaris, Linux, Mac OS X)
- Support for thread creation and synchronization

Thread Call

<table>
<thead>
<tr>
<th>Thread Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pthread_create</td>
<td>Create a new thread in the caller’s address space</td>
</tr>
<tr>
<td>pthread_exit</td>
<td>Terminate the calling thread</td>
</tr>
<tr>
<td>pthread_join</td>
<td>Wait for a thread to terminate</td>
</tr>
<tr>
<td>pthread_mutex_init</td>
<td>Create a new mutex</td>
</tr>
<tr>
<td>pthread_mutex_destroy</td>
<td>Destroy a mutex</td>
</tr>
<tr>
<td>pthread_mutex_lock</td>
<td>Lock a mutex</td>
</tr>
<tr>
<td>pthread_mutex_unlock</td>
<td>Unlock a mutex</td>
</tr>
<tr>
<td>pthread_cond_init</td>
<td>Create a condition variable</td>
</tr>
<tr>
<td>pthread_cond_destroy</td>
<td>Destroy a condition variable</td>
</tr>
<tr>
<td>pthread_cond_wait</td>
<td>Wait on a condition variable</td>
</tr>
<tr>
<td>pthread_cond_signal</td>
<td>Release one thread waiting on a condition variable</td>
</tr>
</tbody>
</table>

Thread Creation

```c
pthread_t threadID;
pthread_create(&threadID, *attr, methodName, *para);
```

- 1st argument is thread ID of the new thread
- 2nd argument is a pointer to pthread_attr_t
- 3rd argument is thread function/method
- 4th argument is a pointer to the arguments for the thread’s method/function

Thread Function: Work for the Thread

- Need to have a special format
- Parameter: single parameter with the type (void *)
  - Basically allow any pointer to be passed
  - Point to a structure (with any number of arguments)
- Return value: a pointer to void
- Example:
  ```c
  void *myfunc(void *vPtr) { ... }; //thread function
  int error; pthread_t tid; int fd;
  if (error = pthread_create(&tid, NULL, myfunc, (void*) fd))
    fprintf(stderr, "Failed to create thread: %s",
            strerror(error));
  ```
Passing Arguments to Threads: int

- Passing an int:
  ```c
  int i=100;
  pthread_create(..., myfunc, (void*)i))
  ```

- Retrieve the parameter as an int:
  ```c
  void *myfunc(void *vptr) {
      int value = (int)vptr;
      ...
  }
  ```

Passing Arguments to Threads: String

- Passing a C-String:
  ```c
  char *str = "CS3733 OS";
  pthread_create(..., myfunc, (void*)str)
  ```

- Retrieve the parameter as an int:
  ```c
  void *myfunc(void *vptr) {
      char *str = (char *)vptr;
      ...
  }
  ```

Passing Arguments to Threads: Array

- Passing a C-String:
  ```c
  int ary[100];
  pthread_create(..., myfunc, (void*)ary)
  ```

- Retrieve the parameter as an int:
  ```c
  void *myfunc(void *vptr) {
      int *ary = (int *)vptr;
      //use as array: ary[x]
  }
  ```

Issues for Thread Arguments Passing

- How to safely pass data to newly created threads
  - Non-deterministic start-up and scheduling ⇒ newly created threads may NOT run immediately !!!
  - Make sure that all passed data is thread safe, i.e., arguments cannot be changed by other threads

- Some notes:
  - Calling thread uses new data or global data for each new thread
  - Arguments for each new thread remain intact throughout the program

How and Where should you declare the parameters (int, string, or array)?
Passing Arguments: Erroneous Case 1

```c
void *myfunc(void *vptr_value) {
    int value = *((int *) vptr_value);
    pthread_exit(NULL);
}
pthread_t launch_thread(void) {
    pthread_t tid; int i = 1024; //local variable
    pthread_create(&tid, NULL, myfunc, &i);
    return tid;
} //other dummy functions
int main() {
    pthread_t tid = launch_thread();
    // Other dummy functions
    pthread_join(tid, NULL);
    return 0;
}
```

What could be the problem?

Passing Arguments: Erroneous Case 2

```c
void *myfunc(void *vptr_value) {
    int value = *((int *) vptr_value);
    pthread_exit(NULL);
}
pthread_t launch_thread(void) {
    pthread_t tid; int i = 1024; //static global variable
    pthread_create(&tid, NULL, myfunc, &i);
    return tid;
}
int main() {
    pthread_t tid = launch_thread();
    pthread_join(tid, NULL);
    return 0;
}
```

What could be the problem?

Passing Arguments: Erroneous Case 3

```c
void *myfunc(void *vptr_value) {
    int value = *((int *) vptr_value);
    pthread_exit(NULL);
}
pthread_t launch_thread(void) {
    pthread_t tid;
    int *iPtr = (int *)malloc(sizeof(int));
    *iPtr = 1024;
    pthread_create(&tid, NULL, myfunc, iPtr);
    return tid;
}
int main() {
    pthread_t tid = launch_thread();
    pthread_join(tid, NULL);
    return 0;
}
```

What could be the problem?

Multiple threads: Bad Parameters

```c
#include <pthread.h>
#include <stdio.h>
#include <string.h>

int main(void) {
    int *para;
    para = (int *)malloc(sizeof(int));
    // para = i;
    for (i = 0; i < NTHREADS; i++) {
        if (error = pthread_create(&tid, NULL, myfunc, arg[i])) {
            fprintf(stderr, "Failed to create thread %s
", strerror(error));
            exit(1);
        }
        pthread_join(tid, NULL);
    }
    printf("All threads done\n");
    return 0;
}
```

Program 12.6: badparameters.c, page 439
An Example: testthread.c

```c
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 5

#define NUM_THREADS 3
... ...
int main(int argc, char *argv[]){
    pthread_t threads[NUM_THREADS];
    int rc;
    long t;
    for(t=0;t<NUM_THREADS;t++){
        printf("In main: creating thread %ld\n", t);
        rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
        if(!rc){
            printf("ERROR: return code from pthread_create() is %d\n", rc);
            exit(-1);
        }
    }
    return value; use pthread_join() wait for other thread; and then return ...
}
```

An Example (cont.)

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 5

void *PrintHello(void *threadid){
    long tid;
    tid = (long)threadid;
    printf("Hello World! It's me, thread #%ld\n", tid);
    pthread_exit(NULL); // return NULL;
}
```

Thread joins and exits

- Joining a non-detached thread by using
  ```c
  int pthread_join ( pthread_t thread, void **retval)
  ```
  (All threads are created non-detached by default, so they are "joinable" by default)

- Exit from threads:
  - If threads use `exit()`, the process terminates.
  - A thread (main, or another thread) can exit by calling `pthread_exit()`, this does not terminate the whole process.

- More information about Pthread programming
  - https://computing.llnl.gov/tutorials/pthreads/

void pthread_exit(void *retval):

- Not necessary for normal exits

- Some special cases:
  - Allows you to exit a thread from any depth in the call stack and return a value via `retval`
  - `pthread_exit()` can terminate your main thread in a controlled way, but not terminating other threads. (NOT recommended)
  - By default, when the main thread returns, it terminates the entire process.
Linux Threads

- Linux uses the term **task** (rather than process or thread) when referring to a flow of control.
- Linux provides `clone()` system call to create threads
  - A set of flags, passed as arguments to the `clone()` system call determine how much sharing is involved (e.g., open files, memory space, etc.).
- Linux: 1-to-1 thread mapping
  - NPTL (Native POSIX Thread Library)

Outline

- Motivation and thread basics
  - Resources requirements: thread vs. process
- Thread implementations/Thread Model
  - User threads: e.g., Pthreads and Java threads
  - Kernel threads: e.g., Linux tasks
  - Map user- and kernel-level threads
- Threading Functions: thread creation/join/exit
- Other multithreaded issues: fork, signals etc.
- Threads Memory Model

Fork() in a Multithreaded Program

- What will happen if one thread in a process invokes `fork()` to create a new process?
  - How many threads in the new process?
- Duplicate only the invoking thread
  - `exec()`: will load another program
  - Everything will be replaced anyway
- Duplicate all threads
  - What about threads perform a blocking system call?
Signal Handling in Multithreading

- Signal handlers are per process, signal masks are per thread
- We can’t predict which thread will handle the signal

struct task_struct {
    /* Signal handlers: */
    struct signal_struct *signal;
    struct sighand_struct *sighand;
    sigset_t sigset_t;
    sigset_t sigset_t;
    struct sigpending pending;
}

All the signals fields are stored per thread. Actually, there is no structure for the process, and all threads on the same process point to the file descriptor table. So the kernel needs to choose a thread to deliver the signal to.

devarea.com/linux-handling-signals-in-a-multithreaded-application

Using Pthread Library

- In the program:
  - #include <pthread.h>
- To compile, link with the pthread library

Linux
- gcc -lpthread // C, Linux
- gcc -lpthread // C++

- You can specify per-thread signal handler with fcntl:
  - F_SETOWN: allows the caller to direct I/O availability signals (SIGIO, SIGURG) to a specific thread, process, or process group

gcc testthread.c -o test -lpthread

inus signals, e.g. SIGSEGV, SIGFPE, SIGBUS, SIGILL, will be caught by current thread

- You can use pthread_kill() to send a signal directly to a thread inside the same process
Outline

- Motivation and thread basics
  - Resources requirements: thread vs. process

- Thread implementations/Thread Model
  - User threads: e.g., Pthreads and Java threads
  - Kernel threads: e.g., Linux tasks
  - Map user- and kernel-level threads

- Threading Functions: thread creation/join/exit
- Other multithreaded issues: fork, signals etc.

- Threads Memory Model

Shared Variables in Threaded C Programs

- Question: Which variables in a threaded C program are shared?
  - The answer is not as simple as "global variables are shared" and "stack variables are private"

- Requires answers to the following questions:
  - What is the memory model for threads?
  - How are variables mapped to memory?
  - How many threads might reference each variable?

A variable $x$ is shared if and only if multiple threads reference some instance of $x$

Threads Memory Model

- Conceptual model:
  - Multiple threads run in the same context of a process
  - Each thread has its own separate thread context
    - Thread ID, stack, stack pointer, PC, and GP registers
  - All threads share the remaining process context
    - Code, data, heap, and shared library segments
    - Open files and installed handlers

- Operationally, this model is not strictly enforced:
  - Register values are truly separate and protected, but...
  - Any thread can read and write the stack of any other thread

Process with Two Threads
Example Program to Illustrate Sharing

```c
char **ptr; /* global */
int main()
{
  int i;
  pthread_t tid;
  char *msg[2] = {
    "Hello from foo",
    "Hello from bar"
  };
  ptr = msg;
  for (i = 0; i < 2; i++)
    pthread_create(&tid, NULL, thread, (void *)i);
}

/* thread routine */
void *thread(void *vargp)
{
  int myid = (int)vargp;
  static int cnt = 0;
  printf("[%d]: %s (svar=%d)\n", myid, ptr[myid], ++cnt);
}
```

Peer threads reference main thread’s stack indirectly through global `ptr` variable.

Mapping Variable Instances to Memory

- **Global variables**
  - Def: Variable declared outside of a function
  - Virtual memory contains exactly one instance of any global variable

- **Local variables**
  - Def: Variable declared inside function without static attribute
  - Each thread stack contains one instance of each local variable

- **Local static variables**
  - Def: Variable declared inside function with the static attribute
  - Virtual memory contains exactly one instance of any local static variable.
Shared Variable Analysis

- **Which variables are shared?**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Referenced by main thread?</th>
<th>Referenced by peer thread 0?</th>
<th>Referenced by peer thread 1?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptr</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>cnt</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>i,n</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>msgs,m</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>myid,p0</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>myid,p1</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

- **Answer:** A variable is shared iff multiple threads reference it. Thus:
  - ptr, cnt, and msgs are shared
  - i and myid are not shared

Outline

- **Motivation and thread basics**
  - Resources requirements: thread vs. process
- **Thread implementations/Thread Model**
  - User threads: e.g., Pthreads and Java threads
  - Kernel threads: e.g., Linux tasks
  - Map user- and kernel-level threads
- **Threading Functions:** thread creation/join/exit
- **Other multithreaded issues:** fork, signals etc.
- **Threads Memory Model
- **Multithreading performance**
  - Thread pool
  - Performance vs. number of threads vs. CPUs and I/Os

Multithreaded Programs

- **Use multiple threads to improve performance**

**Summary**

- **Motivation and thread basics**
  - Resources requirements: thread vs. process
- **Thread implementations/Thread Model**
  - User threads: e.g., Pthreads and Java threads
  - Kernel threads: e.g., Linux tasks
  - Map user- and kernel-level threads
- **Threading Functions:** thread creation/join/exit
- **Other multithreaded issues:** fork, signals etc.
- **Threads Memory Model**