CS 5523 Operating Systems: Thread and Implementation

Instructor: Dr. Tongping Liu

Thank Dr. Dakai Zhu, Dr. Palden Lama, and Dr. Tim Richards (UMASS) for providing their slides.
Outline

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

- Thread implementations
  - User threads: e.g., Pthreads and Java threads
  - Kernel threads: e.g., Linux tasks
  - Map user- and kernel-level threads
  - Lightweight process and scheduler activation

- Other issues with threads: process creation and signals etc.

- Threaded programs
  - Thread pool
  - Performance vs. number of threads vs. CPUs and I/Os
Traditional Process: Single Activity

- Multi-programming
  - More processes share CPU

- One process: address space to hold process image
  - System resources, e.g., files, I/O devices

- Execution states
  - Processor registers (e.g., PC register)
  - 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 the 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

- Responsiveness
  - Part of blocked

- Resource Sharing
  - Memory, open files, etc.

- Economy
  - Creation and switches

- Scalability
  - Increase parallelism
Process: Traditional View

- 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**

- Stack
- Shared libraries
- Run-time heap
- Read/write data
- Read-only code/data
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

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

**Stack**
Process with Two Threads

Thread 1
- Program context:
  - Data registers
  - Condition codes
  - Stack pointer (SP)
  - Program counter (PC)

- Stack

Thread 2
- Program context:
  - Data registers
  - Condition codes
  - Stack pointer (SP)
  - Program counter (PC)

- Stack

Code, data, and kernel context
- Shared libraries
- Run-time heap
- Read/write data
- Read-only code/data

Kernel context:
- VM structures
- Descriptor table
- Brk pointer
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 as thread control
    - ✓ Context switches: processes are more expensive than for threads
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!
Thread Implementations: Issues

- 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.
- **Who and where** to manage threads
  - **User space**: managed by applications
  - **Kernel space**: managed by OS
    - ✓ all modern OSes have kernel level support
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.
Multithreading Models: One-to-One

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

- Provide programmers with API for creating and managing threads

- Two primary ways of implementing
  - **User-level library**
    - Entirely in user space
    - Everything is done using function calls (not system calls)
  - **Kernel-level library supported by the OS**
    - 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)
Pthreads: POSIX Thread

- **POSIX**
  - Portable Operating System Interface [for Unix]
  - Standardized programming interface

- **Pthreads**
  - Thread implementations adhering to POSIX standard
  - API specifies 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
## Pthreads: Thread/Synchronization APIs

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

- **Thread management**
  - Routines for creating, detaching, joining, etc.
  - Routines for setting/querying thread attributes

- **Mutexes**: abbreviation for "mutual exclusion"
  - Routines for creating, destroying, locking/unlocking
  - Functions to set or modify attributes with mutexes.

- **Conditional variables**
  - Communications for threads that share a mutex
  - Functions to create, destroy, wait and signal based on specified variable values
  - Functions to set/query condition variable attributes

- **Synchronization**
  - Routines that manage read/write locks and barriers
Thread Creation

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

- **1st** argument is the ID of the new thread
- **2nd** argument is a pointer to `pthread_attr_t`
- **3rd** argument is `thread (function/method) name`
- **4th** argument is a pointer to the arguments for the thread’s method/function
An Example: testthread.c

// testthread.c
#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);
        }
    }

    //to 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);
}

int main(int argc, char *argv[]){
    ......
}
```
Thread joins and exits

- Join with a non-detached thread by using
  
  ```c
  pthread_join ( pthread_t thread, void **status)
  ```

  (All threads are created non-detached by default, so they are “joinable” by default)

- Exit from threads:
  - If threads use `exit()`, 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/](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 function and thread in a controlled way, but not terminating other threads. (NOT recommended)
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)
Other Issues: *Process Creation in Thread*

- What will happen if one thread in a process call `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 performed blocking system call?!
Using Pthread Library

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

- Linux
  - `gcc -lpthread // C`
  - `g++ -lpthread // C++`

```
gcc testthread.c -o test -lpthread
```
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 \( \times \) is \textit{shared} if and only if multiple threads reference some instance of \( \times \)
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

Thread 1
Program context:
- Data registers
- Condition codes
- Stack pointer (SP)
- Program counter (PC)

stack

Thread 2
Program context:
- Data registers
- Condition codes
- Stack pointer (SP)
- Program counter (PC)

stack

Code, data, and kernel context

- shared libraries
- run-time heap
- read/write data
- read-only code/data

Kernel context:
- VM structures
- Descriptor table
- brk pointer
```c
char **ptr; /* global */

int main()
{
    int i;
    pthread_t tid;
    char *msgs[2] = {
        "Hello from foo",
        "Hello from bar"
    };
    ptr = msgs;

    for (i = 0; i < 2; i++)
        Pthread_create(&tid,
                        NULL,
                        thread,
                        (void *)i);

    Pthread_exit(NULL);
}

/* 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.
Mapping Variable Instances to Memory

Global var: 1 instance (ptr [data])

char **ptr; /* global */

int main()
{
    int i;
    pthread_t tid;
    char *msgs[2] = {
        "Hello from foo",
        "Hello from bar"
    };
    ptr = msgs;

    for (i = 0; i < 2; i++)
        Pthread_create(&tid,
                        NULL,
                        thread,
                        (void *)i);
    ....
}

Local vars: 1 instance (i.m, msgs.m)

Local var: 2 instances (myid.p0 [peer thread 0’s stack], myid.p1 [peer thread 1’s stack])

/* thread routine */
void *thread(void *vargp)
{
    int myid = (int)vargp;
    static int cnt = 0;

    printf("[%d]: %s (svar=%d)\n", myid, ptr[myid], ++cnt);
}

Local static var: 1 instance (cnt [data])

sharing.c
Mapping Variable Instances to Memory

Global var: 1 instance (ptr [data])

Local vars: 1 instance (i.m, msgs.m)

Local var: 2 instances (myid.p0 [peer thread 0’s stack], myid.p1 [peer thread 1’s stack])

Variable instance | Referenced by main thread? | Referenced by peer thread 0? | Referenced by peer thread 1?
--- | --- | --- | ---
ptr | yes | yes | yes
cnt | no | yes | yes
i.m | yes | yes | yes
msgs.m | yes | no | yes
myid.p0 | no | yes | yes
myid.p1 | no | no | yes

sharing.c
Shared Variable Analysis

Which variables are shared?

<table>
<thead>
<tr>
<th>Variable</th>
<th>Referenced by instance</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>
<td>yes</td>
</tr>
<tr>
<td>cnt</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>i.m</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>msgs.m</td>
<td>yes</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>
<td>no</td>
</tr>
<tr>
<td>myid.p1</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Answer: A variable \( x \) is shared iff multiple threads reference at least one instance of \( x \). Thus:

- \( \text{ptr, cnt, and msgs are shared} \)
- \( i \) and \( \text{myid are not shared} \)
Multithreaded Programs

- Use multiple threads to improve performance

How should the server handle the incoming requests?
Thread Pool

- Pool of threads
  - Threads in a pool where they wait for work

- Advantages:
  - Usually slightly faster to service a request with an existing thread than create a new thread
  - Allows the number of threads in the application(s) to be bound to the size of the pool

- Adjust thread number in pool
  - According to usage pattern and system load
Thread Pool Example: Web server

![Diagram of a thread pool example for a web server]

```c
while(TRUE) {
    getNextRequest(&buf);
    handoffWork(&buf);
}
```

```c
while(TRUE) {
    waitForWork(&buf);
    lookForPageInCache(&buf,&page);
    if(pageNotInCache(&page)) {
        readPageFromDisk(&buf,&page);
    }
    returnPage(&page);
}
```
Thread Pool Implementation

- Work queue
  - Fixed number of threads

- Other possible problems
  - Deadlock
  - Resource thrashing
  - Thread leakage
  - Overload
Performance of Threaded Programs

- Suppose that the processing of each request
  - Takes $X$ seconds for computation; and
  - Takes $Y$ seconds for reading data from I/O disk

- For single-thread program/process
  - A single CPU & single disk system
  - What is the maximum throughput (i.e., the number of requests can be processed per second)?

Example: suppose that each request takes
2ms for computation
8ms to read data from disk

$\frac{1000}{10\text{ms}} = 100$
Multi-thread performance improvement

- Single CPU & single disk system
- How many threads should be used?
- What is the maximum throughput (i.e., the number of requests can be processed per second)?

Example: suppose that each request takes
  2ms for computation
  8ms to read data from disk

Assuming that we have 8 cores and 1 disk

\[
1000/8\text{ms} = 125
\]
Performance of Threaded Programs (cont)

- What about $m$-CPU and $n$-disk system
  - Maximum throughput and # of threads? (X: computation, Y: IO for each task)
  - **Throughput** $\Rightarrow 1 / \max(X/m, Y/n)$
  - if $(X/m < Y/n)$, $\# = n + m'$
    where $m' = \min\{k | X/k \leq Y/n, 1 \leq k \leq m\}$;
  - Similarly, if $(X/m > Y/n)$, $\# = m + n'$
    where $n' = \min\{k | X/m \geq Y/k, 1 \leq k \leq n\}$;

- Other issues:
  - When I/O disk is bottleneck, adding more CPUs will NOT help improve the throughput
  - What about heterogeneous disks and CPUs?!
Summary

- Thread basics
  - Resources requirements: thread vs. process
- Thread implementations
  - User threads: e.g., Pthreads and Java threads
  - Kernel threads: e.g., Linux tasks
  - Map user- and kernel-level threads
  - Lightweight process and scheduler activation
- Other issues with threads
  - process creation and signals etc.
- Threaded programs
  - Thread pool
  - Performance vs. number of threads vs. CPUs and I/Os
For next lecture

- **Concurrency and Synchronization**
  - SGG Chapters 6 and 7
  - TS Chapter 6