Threads

Jinkyu Jeong (jinkyu@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu
Today’s Topics

- Why threads?
- Threading issues


## Processes

### Heavy-weight

- A process includes many things:
  - An address space (all the code and data pages)
  - OS resources (e.g., open files) and accounting info.
  - Hardware execution state (PC, SP, registers, etc.)
- Creating a new process is costly because all of the data structures must be allocated and initialized
  - Linux: over 100 fields in `task_struct`
  (excluding page tables, etc.)
- Inter-process communication is costly, since it must usually go through the OS
  - Overhead of system calls and copying data
Concurrent Servers: Processes

- Web server example
  - Using fork() to create new processes to handle requests in parallel is overkill for such a simple task.

```c
While (1) {
    int sock = accept();
    if ((pid = fork()) == 0) {
        /* Handle client request */
        } else {
            /* Close socket */
        }
}
```
Cooperating Processes

- Example
  - A web server, which forks off copies of itself to handle multiple simultaneous tasks
  - Any parallel program on a multiprocessor

- We need to:
  - Create several processes that execute in parallel
  - Cause each to map the same address space to share data (e.g., shared memory)
  - Have the OS schedule these processes in parallel

- This is very inefficient!
  - Space: PCB, page tables, etc.
  - Time: creating OS structures, fork and copy address space, etc.
Rethinking Processes

- **What’s similar in these cooperating processes?**
  - They all share the same code and data (address space)
  - They all share the same privilege
  - They all share the same resources (files, sockets, etc.)

- **What’s different?**
  - Each has its own hardware execution state: PC, registers, SP, and stack.
Key Idea (1)

- Separate the concept of a process from its execution state
  - Process: address space, resources, other general process attributes (e.g., privileges)
  - Execution state: PC, SP, registers, etc.

  - This execution state is usually called
    - a thread of control,
    - a thread, or
    - a lightweight process (LWP)
Key Idea (2)

User space 

Kernel space 

Process 1  Process 1  Process 1

Thread 

Kernel 

(a) 

Process 

Thread 

Kernel 

(b)
Key Idea (3)

- Single-threaded process
  - Thread
  - Registers
  - Stack

- Multithreaded process
  - Thread
  - Registers
  - Stack

- Code
- Data
- Files

- Code
- Data
- Files
What is a Thread?

- A thread of control (or a thread)
  - A sequence of instructions being executed in a program.
  - Usually consists of
    - a program counter (PC)
    - a stack to keep track of local variables and return addresses
    - registers
  - Threads share the process instructions and most of its data.
    - A change in shared data by one thread can be seen by the other threads in the process
  - Threads also share most of the OS state of a process.
Using threads

- We can create a new thread for each request.

```c
webserver ()
{
    While (1) {
        int sock = accept();
        thread_fork (handle_request, sock);
    }
}
handle_request (int sock)
{
    /* Process request */
    close (sock);
}
```
Multithreading

- **Benefits**
  - Creating concurrency is cheap.
  - Improves program structure.
  - Throughput
    - By overlapping computation with I/O operations
  - Responsiveness (User interface / Server)
    - Can handle concurrent events (e.g., web servers)
  - Resource sharing
  - Economy
  - Utilization of multiprocessor architectures
    - Allows building parallel programs.
Processes vs. Threads

- Processes vs. Threads
  - A thread is bound to a single process.
  - A process, however, can have multiple threads.
  - Sharing data between threads is cheap: all see the same address space.
  - Threads become the unit of scheduling.
  - Processes are now containers in which threads execute.
  - Processes become static, threads are the dynamic entities.
Process Address Space

- **0xFFFFFFFF**
- **address space**
- **0x00000000**

Diagram:
- Stack (dynamically allocated mem)
- Heap (dynamically allocated mem)
- Static data (data segment)
- Code (text segment)

Arrows:
- SP (Stack Pointer)
- PC (Program Counter)
Address Space with Threads

0xFFFFF

address space

0x00000000

PC

thread 1 stack

thread 2 stack

thread 3 stack

heap
(dynamically allocated mem)

static data
(data segment)

code
(text segment)

SP (T1)

SP (T2)

SP (T3)

PC (T1)

PC (T2)

PC (T3)
# threads per addr space: | # of addr spaces: | One | Many
---|---|---|---
One | | MS/DOS Early Macintosh | Traditional UNIX
Many | Many embedded Oses (VxWorks, uClinux, ..) | Mach, OS/2, Linux, Windows, Mac OS X, Solaris, HP-UX
Threads Interface (1)

- **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.
Threads Interface (2)

- POSIX-style threads
  - Pthreads
  - DCE threads (early version of Pthreads)
  - Unix International (UI) threads (Solaris threads)
    - Sun Solaris 2, SCO Unixware 2

- Microsoft-style threads
  - Win32 threads
    - Microsoft Windows 98/NT/2000/XP
  - OS/2 threads
    - IBM OS/2
Pthreads (1)

- Thread creation/termination

```c
int pthread_create (pthread_t *tid,
                   pthread_attr_t *attr,
                   void *(start_routine)(void *),
                   void *arg);

void pthread_exit (void *retval);

int pthread_join (pthread_t tid,
                  void **thread_return);
```
Pthreads (2)

- Mutexes

```c
int pthread_mutex_init
    (pthread_mutex_t *mutex,
     const pthread_mutexattr_t *mattr);

void pthread_mutex_destroy
    (pthread_mutex_t *mutex);

void pthread_mutex_lock
    (pthread_mutex_t *mutex);

void pthread_mutex_unlock
    (pthread_mutex_t *mutex);
```
### Condition variables

- **int pthread_cond_init**
  ```c
  (pthread_cond_t *cond,
   const pthread_condattr_t *cattr);
  ```

- **void pthread_cond_destroy**
  ```c
  (pthread_cond_t *cond);
  ```

- **void pthread_cond_wait**
  ```c
  (pthread_cond_t *cond,
   pthread_mutex_t *mutex);
  ```

- **void pthread_cond_signal**
  ```c
  (pthread_cond_t *cond);
  ```

- **void pthread_cond_broadcast**
  ```c
  (pthread_cond_t *cond);
  ```
Threading Issues (1)

- fork() and exec()
  - When a thread calls fork(),
    - Does the new process duplicate all the threads?
    - Is the new process single-threaded?
  - Some UNIX systems support two versions of fork().
    - In Pthreads,
      » fork() duplicates only a calling thread.
    - In the Unix International standard,
      » fork() duplicates all parent threads in the child.
      » fork1() duplicates only a calling thread.
  - Normally, exec() replaces the entire process.
Threading Issues (2)

- Thread cancellation
  - The task of terminating a thread before it has completed.
  - Asynchronous cancellation
    - Terminates the target thread immediately.
    - What happens if the target thread is holding a resource, or it is in the middle of updating shared resources?
  - Deferred cancellation
    - The target thread is terminated at the cancellation points.
    - The target thread periodically check if it should be cancelled.
  - Pthreads API supports both asynchronous and deferred cancellation.
Threading Issues (3)

- Signal handling
  - Where should a signal be delivered?
    - To the thread to which the signal applies.
      - for synchronous signals.
  - To every thread in the process.
  - To certain threads in the process.
    - Typically only to a single thread found in a process that is not blocking the signal.
    - Pthreads: per-process pending signals, per-thread blocked signal mask
  - Assign a specific thread to receive all signals for the process.
    - Solaris 2
Threading Issues (4)

- **Using libraries**
  - **errno**
    - Each thread should have its own independent version of the errno variable.
  - **Multithread-safe (MT-safe)**
    - A set of functions is said to be multithread-safe or reentrant, when the functions may be called by more than one thread at a time without requiring any other action on the caller’s part.
    - Pure functions that access no global data or access only read-only global data are trivially MT-safe.
    - Functions that modify global state must be made MT-safe by synchronizing access to the shared data.