CSE 120
Principles of Operating Systems
Spring 2020

Lecture 4: Threads

Yiying Zhang
Process Creation -- UNIX interfaces (1)

- `fork()` system call creates a duplicate of the original process

```
main()
{
  ...
  foo()
  ...
  I = fork()
}
foo()
{
  ...
}
```

Process

```
main()
{
  ...
  foo()
  ...
  I = fork()
}
foo()
{
  ...
}
```

Process
Process Creation: Unix interfaces (2)

• Wait a second. How do we actually start a new program?

\[ \text{int exec(char *prog, char *argv[])} \]

• `exec()` system call used after a `fork` to replace the process’ code/address space with a new program
  - Important: BOTH code and data, i.e., the whole address space is replaced!
exec("b.out")

// a.out
main()
{
... 
foo()
... 
exec("b.out")
}

foo()
{
... 
}

Process

Afterwards, only one thing about the process was kept, which is?
Process Creation: Unix interfaces (2)

- `exec()`
  - Stops the current process
  - Loads the program “prog” into the process’ address space
  - Initializes hardware context and args for the new program
  - Places the PCB onto the ready queue
  - Note: It **does not** create a new process

- What does it mean for `exec` to return?
Process Creation – UNIX interfaces

UNIX system calls related to process creation/termination:

- **fork** – create a copy of this process
  - Clone would have been a better name!
- **exec** – replace this process with a new program
- **fork()** is used to create a new process, **exec** is used to load a program into the address space
- **exit/kill** – (potentially) end a running process
- **wait** – wait for child process to finish
Process Termination

• All good processes must come to an end. But how?
  ♦ Unix: `exit(int status)`
• Essentially, free resources and terminate
  ♦ Terminate all threads (next lecture)
  ♦ Close open files, network connections
  ♦ Release allocated memory (and VM pages out on disk)
  ♦ Remove PCB from kernel data structures, delete
• Note that a process does not need to clean up itself
  ♦ Why does the OS have to do it?
What happens when a parent process disappears?
all child processes are killed by the OS, or
all child processes reset init as their parent
wait() a second...

- Often it is convenient to pause until a child process has finished
  - Think of executing commands in a shell
- Unix `wait()`
  - Suspends the current process until any child process ends
  - `waitpid()` suspends until the specified child process ends
#include <stdio.h>
void main()
{
    int pid; int was = 3;
    pid = fork(); /* fork another process */

    if (pid == 0) { /* child process */
        sleep(2); printf("child: was = %d\n", was);
        execlp("/bin/ls", "ls", NULL);}
    else { /* pid > 0; parent process */
        was = 4;
        printf("parent: child process id = %d; was=%d\n", pid, was);
        wait(NULL); exit(0);
    }
}
```c
#include <stdio.h>

void main()
{
    int ret_of_fork;   int was = 3;
    ret_of_fork= fork(); /* fork another process */

    if (ret_of_fork== 0) { /* child process */
        sleep(2); was=9;  printf("child: was = %d\n", was);
        execlp("/bin/ls", "ls", NULL);       was = 10;
        printf("It’s me, your child was = %d\n", was);
    }
    else { /* pid > 0; parent process */
        was = 4;
        printf("parent: child process id %d was=%d\n", pid, was);
        wait(NULL);   exit(0);
    }
}
```
Unix Shells

while (1) {
    char *cmd = read_command();
    int child_pid = fork();
    if (child_pid == 0) {
        Manipulate STDIN/OUT/ERR file descriptors for pipes, redirection, etc.
        exec(cmd);
        panic("exec failed");
    } else {
        waitpid(child_pid);
    }
}

Process Summary

• What are the units of execution?
  ♦ Processes

• How are those units of execution represented?
  ♦ Process Control Blocks (PCBs)

• How is work scheduled in the CPU?
  ♦ Process states, process queues, context switches

• What are the possible execution states of a process?
  ♦ Running, ready, waiting

• How does a process move from one state to another?
  ♦ Scheduling, I/O, creation, termination

• How are processes created?
  ♦ fork/exec
Processes

• Recall that a process includes many things
  ♦ An address space (defining all the code and data pages)
  ♦ OS resources (e.g., open files) and accounting information
  ♦ Execution state (PC, SP, regs, etc.)

• Creating a new process is costly because of all of the data structures that must be allocated and initialized
  ♦ Recall `task_struct` in Linux

• Communicating between processes is also costly
  ♦ How to communicate? Each process is an island
  ♦ The OS needs to intervene to bridge the gap
  ♦ OS provides system calls to support Inter-Process Communication (IPC)
How do processes communicate?

• At process creation time
  ♦ Parents get one chance to pass everything at fork()

• OS provides generic mechanisms to communicate
  ♦ Shared Memory: multiple processes can read/write same physical portion of memory; implicit channel
    » System call to declare shared region
    » No OS mediation required once memory is mapped
  ♦ Message Passing: explicit communication channel provided through send()/receive() system calls
    » A system call is required

• IPC is, in general, expensive due to the need for system calls
  ♦ Although many OSes have various forms of lightweight IPC
Concurrent Programs

• Recall our Web server example that forks off copies of itself to handle multiple simultaneous requests
  ♦ Or any parallel program that executes on a multiprocessor

• To execute these programs we need to
  ♦ Create several processes that execute in parallel
  ♦ Cause each to map to the same address space to share data
    » They are all part of the same computation
  ♦ Schedule these processes in parallel (logically or physically)

• This situation is very inefficient
  ♦ **Space**: PCB, page tables, etc.
  ♦ **Time**: create data structures, fork and copy addr space, etc.
Rethinking Processes

• For some cases, forked processes are cooperative
  ♦ They all share the same code and data (address space)
  ♦ They all share the same privileges
  ♦ They all share the same resources (files, sockets, etc.)

• What don’t they share?
  ♦ Each has its own execution state: PC, SP, and registers

• **Key idea**: Why don’t we separate the concept of a process from its execution state?
  ♦ **Process**: address space, privileges, resources, etc.
  ♦ **Execution state**: PC, SP, registers

• Exec state also called **thread of control**, or **thread**
Threads

- Modern OSes (Windows, Unix, OS X) separate the concepts of processes and threads
  - The thread defines a sequential execution stream within a process (PC, SP, registers)
  - The process defines the address space and general process attributes (everything but threads of execution)
- A thread is bound to a single process
  - Processes, however, can have multiple threads
- Threads become the basic unit of scheduling
  - Processes are now the containers in which threads execute
Single and Multithreaded Processes
Threads in a Process

- Stack (T1)
- Stack (T2)
- Stack (T3)
- Heap
- Static Data
- Code

Thread 1
- PC (T1)

Thread 2
- PC (T2)

Thread 3
- PC (T3)
Process/Thread Separation

- Separating threads and processes makes it easier to support multithreaded applications
  - Concurrency does not require creating new processes
- Concurrency (multithreading) can be very useful
  - Improving program structure
  - Handling concurrent events (e.g., Web requests)
  - Writing parallel programs
- So multithreading is even useful on a uniprocessor
  - Although today even cell phones are multicore
Process Control Block

- Process management info
  - State (ready, running, blocked)
  - PC & Registers
  - CPU scheduling info (priorities, etc.)
  - Parent info

- Memory management info
  - Segments, page table, stats, etc
  - Code, data, heap, execution stack

- I/O and file management
  - Communication ports, directories, file descriptors, etc.
Thread Control Block

- Shared information
  - Process info: parent process
  - Memory: code/data segments, page table, and stats
  - I/O and file: comm ports, open file descriptors

- Private state
  - State (ready, running and blocked)
  - PC, Registers
  - Execution stack
Threads: Concurrent Servers

- Using fork() to create new processes to handle requests in parallel is overkill
- Recall our forking Web server:

```c
while (1) {
    int sock = accept();
    if ((child_pid = fork()) == 0) {
        Handle client request
        Close socket and exit
    } else {
        Close socket
    }
}
```
Threads: Concurrent Servers

- Instead, we can create a new thread for each request

```c
web_server() {
    while (1) {
        int sock = accept();
        thread_fork(handle_request, sock);
    }
}

handle_request(int sock) {
    Process request
    close(sock);
}
```
Thread Design Space

- One Thread per Process
  - One Address Space
    - (MSDOS)

- Many Threads per Process
  - One Address Space
    - (Java VM)

- One Thread per Process
  - Many Address Spaces
    - (Early Unix)

- Many Threads per Process
  - Many Address Spaces
    - (Solaris, Linux, NT, MacOS)
Scheduling Threads

• No longer just scheduling processes, but threads
  ♦ Kernel scheduler used to pick among PCBs
  ♦ Now what?

• We have basically two options
  ♦ Kernel explicitly selects among threads in a process
  ♦ Hide threads from the kernel, and have a user-level scheduler inside each multi-threaded process

• Why do we care?
  ♦ Think about the overhead of switching between threads
  ♦ Who decides which thread in a process should go first?
  ♦ What about blocking system calls?
An Analogy: Family Car Rental

- Scenario (a day is 9am-5pm)
  - Avis rents a car to 2 family, Round-robin daily
  - Each family has 4 members, round-robin every 2 hrs

- Two ways of doing it:
  - Global scheduler: Avis schedules family for each day, family schedules among its members
  - Local scheduler: Avis schedules among 8 members
Thread Implementations

- User-level thread implementation
- Kernel-level thread implementation
User-Level Thread Implementation

• User-level threads are managed entirely by a run-time system (a.k.a. user-level thread library)
  ♦ Creation / scheduling
  ♦ No kernel intervention (kernel sees single entity)

• Invisible to kernel
  ♦ A thread represented inside process by a PC, registers, stack, and small thread control block (TCB)
  ♦ Creating a new thread, switching, and synchronizing threads are done via **user-level procedure call**
  ♦ User-level thread operations **100x faster** than kernel thread

• The kernel only sees one scheduling entity (which has many user threads inside)
User-Level Threads Mapping to Kernel Threads

- Definition: **Kernel thread** is the kernel scheduling unit

- In user thread implementation, all user threads of the same process are effectively mapped to one kernel thread

- Examples
  - *pthread*: PTHREAD_SCOPE_PROCESS
Context switching user-level threads

• If belonging to the same process
  ♦ Handled by the dispatcher in the thread library
    » Only need to store/load the TCB information
  ♦ OS does not do anything

• If belonging to different processes
  ♦ Like an ordinary context switch of two processes
    » Handled by OS (drop in/out of the kernel)
    » OS needs to load/store PCB information and TCB information
User-Level Thread Limitations

• What happens if a thread invokes a syscall?
  ♦ A blocking syscall blocks the whole process!

• User-level threads are invisible to the OS
  ♦ They are not well integrated with the OS

• As a result, the OS can make poor decisions
  ♦ Scheduling a process with idle threads
  ♦ Blocking a process whose thread initiated an I/O, even though the process has other threads that can execute
  ♦ Unscheduling a process with a thread holding a lock
Kernel-Level Thread Implementation

- OS now manages threads and processes
  - All thread operations are implemented in the kernel
  - Kernel performs thread creation, scheduling, and management (each thread is a scheduling entity)
  - The OS schedules all of the threads in the system
- OS-managed threads are called *kernel-level threads* or *lightweight processes*
- Scheduler deals in threads
  - PCBs are no longer scheduled
  - If a thread blocks, another thread in the same process can run
Kernel Thread Implementation

- Each user thread maps to a kernel thread
- Examples: Windows family, Linux

- Slow to create and manipulate
  + Integrated with OS well (e.g., a blocking syscall will not block the whole process)
Kernel vs. User Threads

• Kernel-level threads
  ♦ Integrated with OS (informed scheduling)
  ♦ Slower to create, manipulate, synchronize

• User-level threads
  ♦ Faster to create, manipulate, synchronize
  ♦ Not integrated with OS (uninformed scheduling)

• Understanding the differences between kernel and user-level threads is important
  ♦ Correctness, performance
Kernel and User Threads

• Or use **both** kernel and user-level threads
  ♦ Can associate a user-level thread with a kernel-level thread
  ♦ Or, multiplex user-level threads on top of kernel-level threads

• Java Virtual Machine (JVM) (also C#, others)
  ♦ Java threads are user-level threads
  ♦ On older Unix, only one “kernel thread” per process
    » Multiplex all Java threads on this one kernel thread
  ♦ On modern OSes
    » Can multiplex Java threads on multiple kernel threads
    » Can have more Java threads than kernel threads
Three multithreading models

- Many-to-One
- One-to-One
- Many-to-Many
Many-to-One

- Many user-level threads mapped to single kernel entity (kernel thread)
- Used in *user thread implementation*
- Drawback: blocking sys call blocks the whole process
One-to-One

- Each user thread maps to kernel thread
- Used in *kernel thread implementation*
- May lead to too many kernel threads
Many-to-Many model

- Allows many user threads to be mapped to many kernel threads
- Allows OS to create a sufficient number of kernel threads running in parallel
  - When one blocks, schedule another user thread
- Examples:
  - Go goroutines: Go schedules an arbitrary number of goroutines onto an arbitrary number of kernel threads
Many-to-Many Model

![Diagram showing a Many-to-Many Model]

- User thread
- Kernel thread
Threads Summary

• The operating system as a large multithreaded program
  ♦ Each process executes as a thread within the OS
• Multithreading is also very useful for applications
  ♦ Efficient multithreading requires fast primitives
  ♦ Processes are too heavyweight
• Solution is to separate threads from processes
  ♦ Kernel-level threads much better, but still significant overhead
  ♦ User-level threads even better, but not well integrated with OS
• Now, how do we get our threads to correctly cooperate with each other?
  ♦ Synchronization…
Next time...

- Read Chapters 28, 29
- HW1 and PR0 due 4/14
Implementing Threads

• Implementing threads has several issues
  ♦ Interface
  ♦ Context switch
  ♦ Preemptive vs. non-preemptive
  ♦ Scheduling
  ♦ Synchronization (next lecture)

• Focus on user-level threads
  ♦ Kernel-level threads are similar to original process management and implementation in the OS
  ♦ What you will be dealing with in Nachos
  ♦ Not only will you be using threads in Nachos, you will be implementing more thread functionality
public class KThread {
    int status;
    String name;
    Runnable target;
    TCB tcb;
    int id;
    <Methods>
};
Nachos Thread API

- **KThread.fork**
  - Run a new thread (also “create”)
- **KThread.sleep**
  - Stop the calling thread (also “stop”, “block”, “suspend”)
- **KThread.ready**
  - Start the given thread (also “start”, “resume”)
- **KThread.yield**
  - Voluntarily give up the processor
- **KThread.join**
  - Block until another thread finishes (Project 1)
- **KThread.finish**
  - Terminate the calling thread (also “exit”, “destroy”)