CSE 120
Principles of Operating Systems
Spring 2023
Lecture 6: Process Creation, Threads
Yiying Zhang
Process Creation —
UNIX interfaces (1)

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

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

Process

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

Process

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

Process
What does this program print?
Example Output

alpenglow (18) ~/tmp> cc t.c
alpenglow (19) ~/tmp> a.out
My child is 486
Child of a.out is 486
Duplicating Address Spaces

```c
child_pid = fork();
if (child_pid == 0) {
    printf("child");
} else {
    printf("parent");
}
```

**Parent**

- `child_pid = 486`

**Child**

- `child_pid = 0`

```c
child_pid = fork();
if (child_pid == 0) {
    printf("child");
} else {
    printf("parent");
}
```
Divergence

```c
child_pid = fork();
if (child_pid == 0) {
    printf("child");
} else {
    printf("parent");
}
```

Parent

```c
child_pid = fork();
if (child_pid == 0) {
    printf("child");
} else {
    printf("parent");
}
```

Child

child_pid = 486

child_pid = 0
Example Continued

alpenglow (18) ~/tmp> cc t.c
alpenglow (19) ~/tmp> a.out
My child is 486
Child of a.out is 486
alpenglow (20) ~/tmp> a.out
Child of a.out is 498
My child is 498

Why is the output in a different order?
Why fork()?

• Very useful when the child…
  ♦ Is cooperating with the parent
  ♦ Relies upon the parent’s data to accomplish its task

• Example: Web server
  
  ```c
  while (1) {
    int sock = accept();
    if ((child_pid = fork()) == 0) {
      Handle client request and exit
    } else {
      Close socket
    }
  }
  ```
Process Creation: Unix interfaces (2)

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

```c
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”)
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 swapped out pages)
  ♦ 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?
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);
    }
}
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);
        execvp("/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);
}

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
• 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

• Applications often want to execute several tasks in parallel
  ✩ When executing on a multi-core system, can get better perf
  ✩ These parallel tasks likely need to access some shared data and/or communicate with each other during their execution

• To execute these programs with processes, we need to
  ✩ Create several processes that execute in parallel (fork)
  ✩ Have each of them map a shared-data memory region (MMAP_SHARED)

• 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

single-threaded

multithreaded

thread
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 parallel applications
  - Concurrency does not require creating new processes
- Concurrency (multithreading) can be very useful
  - Improving program structure
  - Handling concurrent events
  - Writing parallel programs
- So multithreading is even useful on a uniprocessor
  - Although today even cell phones are multicore
• 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 sometimes 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);
    }
}
```

```c
disable_request(int sock) {
    Process request
    close(sock);
}
```

*Why would you still choose multi-processes to handle web requests?*
Thread Design Space

Address Space

Thread

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)
Threads Summary

• Many applications demand internal parallelism
  ♦ Efficient parallelism requires fast primitives
  ♦ Processes are too heavyweight

• Solution is to separate threads from processes

• Now, how do we get our threads (and processes with shared memory) to correctly cooperate with each other?
  ♦ Synchronization…
Next time...

- Read Chapters 28, 29
Backup Slides
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”)