Operating Systems

Lecture 4:
Abstractions I: Threads
Abstractions II: Processes
Abstractions III: IPC

Rachit Agarwal
Goal of Today’s Lecture

• Wrap up discussion on the first abstraction: thread
• Deeper dive into the second abstraction: process
• Introduction to the third abstraction: IPC abstractions
Recall: Four Fundamental OS Concepts

- **Thread**: Execution Context
  - A virtual core: a single, sequential execution context

- **Address space (with translation)**
  - Program's view of memory is distinct from physical memory

- **Process**: an instance of a running program
  - Address Space + One or more Threads + ...

- **Protection/Isolation**
  - Only the “system” can access certain resources
  - Combined with translation, isolates programs from each other
Recall: Threads

• **Virtual cores: illusion of infinite processors**
  • Each thread executes a sequence of instructions, in order, on a physical core

• **Why threads?**
  • *Statistical multiplexing*: improved utilization of physical cores

• **Challenges:**
  • synchronization (correctness), scheduling (performance)
Recall: Address space

- **Virtual address space:** illusion of infinite memory

- **Why virtual address space?**
  - *Statistical multiplexing:* improved utilization of physical memory
  - *Protection/Isolation* (not yet covered)
  - ....

- **Challenges?**
  - Efficient address translation
Recall: Process

• Execution environment with restricted rights: Illusion of a machine
  • One or more threads
  • Execution state: everything that can affect, or be affected by, a thread
    • Code, data, registers, call stack, files, sockets, etc.
  • Part of the process state is “owned” by individual threads
  • Part is shared among all threads in the process

• Why processes?
  • Statistical multiplexing: improved utilization of physical resources

• Challenges?
  • Protection/isolation/sharing
Recall: Protection/Isolation

• Virtualization (address space, in particular)

• Dual mode operations
  • Hardware provides at least two modes of operations:
    • Kernel mode (or “supervisor” / “protected” mode)
    • User mode
      • Processes execute in user mode
  • “Controlled” transitions between user mode and kernel mode
    • System calls, interrupts, exceptions
Recall: Need for Threads

- Consider the following program:
  ```
  main() {
    ComputePI();
    PrintClassList("classlist.txt");
  }
  ```

- The program would never print out class list:
  - **ComputePI** would never finish
Recall: With Threads

• Version of program with threads (loose syntax):

```c
main() {
    create_thread(ComputePI());
    create_thread(PrintClassList("classList.txt"));
}
```

• Now, you would actually see the class list
  • But only “now and then”
  • **Illusion: infinite number of processors (potentially varying speeds)**

• `create_thread`: Spawns a new thread running the given procedure
  • Should behave as if another CPU is running the given procedure
Questions?
Wrapping up Abstraction I: Threads
Multithreaded Programs

• When you compile a C program and run the executable
  • It creates a process that is executing that program

• Initially, this new process has one thread in its own address space
  • With code, globals, etc. as specified in the executable

• How can we make a multithreaded process?
  • A process can issues syscall to create new threads
  • These new threads are part of the process:
    • They share its address space
New Idea: Fork-Join Pattern

- Main thread *creates* (forks) collection of sub-threads passing them args to work on...
- ... and then *joins* with them, collecting results.
Memory Layout with Two Threads

- Two sets of CPU registers
- Two sets of stacks
- Issues:
  - How do we position stacks relative to each other?
  - What maximum size should we choose for the stacks?
  - What happens if threads violate this?
  - How might you catch violations?
Thread Abstraction

- **Illusion:** infinite number of processors, potentially varying speeds
- **Reality:** threads execute with variable “speed”
  - Why?
    - Depends on scheduling policies
- **Programs** must be designed to work with any schedule
## Programmer vs. Processor View

<table>
<thead>
<tr>
<th>Programmer’s View</th>
<th>Possible Execution</th>
<th>Possible Execution</th>
<th>Possible Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$x = x + 1;$</td>
<td>$x = x + 1;$</td>
<td>$x = x + 1$</td>
<td>$x = x + 1$</td>
</tr>
<tr>
<td>$y = y + x;$</td>
<td>$y = y + x;$</td>
<td>$\ldots$</td>
<td>$\ldots$</td>
</tr>
<tr>
<td>$z = x + 5y;$</td>
<td>$z = x + 5y;$</td>
<td>thread is suspended</td>
<td>thread is suspended</td>
</tr>
<tr>
<td></td>
<td></td>
<td>other thread(s) run</td>
<td>other thread(s) run</td>
</tr>
<tr>
<td></td>
<td></td>
<td>thread is resumed</td>
<td>thread is resumed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$y = y + x$</td>
<td>$y = y + x$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$z = x + 5y$</td>
<td>$z = x + 5y$</td>
</tr>
</tbody>
</table>
Correctness with Concurrent Threads

• Goal: Correctness by Design
  • What makes this a challenging goal?

• Non-determinism:
  • Scheduler can run threads in any (non-deterministic) order
    • Why?
  • Scheduler can switch threads at any time
    • Why?

• Independent Threads
  • No state shared with other threads
  • Deterministic, reproducible conditions

• Cooperating Threads
  • Shared state between multiple threads
Race Conditions

• Initially x == 0 and y == 0

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1;</td>
<td>y = 2;</td>
</tr>
</tbody>
</table>

• What are the possible values of x below after all threads finish?

• Must be 1. Thread B does not interfere with Thread A.
Race Conditions

- Initially $x == 0$ and $y == 0$

**Thread A**

$x = y + 1;$

**Thread B**

$y = 2;$

$y = y \times 2;$

- What are the possible values of $x$ below?

- 1 or 3 or 5 (non-deterministic)

- Race Condition: Thread A “races” against Thread B!
Abstraction II: Processes
Recall: Process

• Definition: execution environment with restricted rights
  • One or more threads
  • Execution state: everything that can affect, or be affected by, a thread
    • Code, data, registers, call stack, files, sockets, etc.
  • Part of the process state is “owned” by individual threads
  • Part is shared among all threads in the process
Process control block (PCB)

- Each process has a “state”
  - Execution state for each thread
  - Scheduling information
  - Information about memory used by the process
  - Information about files, sockets, etc.
  - ..
Processes

• How to manage process state?
  • How to create a process?
  • How to manage process state?
  • How to exit from a process?

• Remember: Everything outside of the kernel is running in a process!

• Processes are created and managed... by processes!
Processes

• Processes are created and managed by ....
  • processes!
  • Hhhmm. How does the first process start?
    • By the kernel
      • Often configured as an argument to the kernel
        • Before the kernel boots
        • Often called the “init” process
  • After this, all processes are created by other processes
Process Management

• **exit** — terminate a process

• **fork** — copy the current process

• **exec** — change the *program* being run by the current process

• **wait** — wait for a process to finish

• **kill** — send a *signal* (interrupt-like notification) to another process

• **sigaction** — set handlers for signals
Process Management

- **exit** — terminate a process
- **fork** — copy the current process
- **exec** — change the *program* being run by the current process
- **wait** — wait for a process to finish
- **kill** — send a *signal* (interrupt-like notification) to another process
- **sigaction** — set handlers for signals
exit ()

• Called after process terminates
  • Deallocates memory
  • Destructs most OS data structures
  • Closes open files
```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
    /* get current processes PID */
    pid_t pid = getpid();
    printf("My pid: %d\n", pid);

    exit(0);
}
```

Q: What if we let main return without ever calling exit?

- The OS Library calls exit() for us!
- The entry point of the executable is in the OS library
- OS library calls main
- If main returns, OS library calls exit
Process Management

- **exit** — terminate a process
- **fork** — copy the current process
- **exec** — change the *program* being run by the current process
- **wait** — wait for a process to finish
- **kill** — send a *signal* (interrupt-like notification) to another process
- **sigaction** — set handlers for signals
fork ()

- **Used to create processes**—copy the current process
- New “child” process has a different process ID (pid) AND a single thread
- New “child” process is a clone:
  - State of original process **duplicated** in both parent and child process
- Returns twice (!), to both the parent and the child process
  - Sets pid to different values (return value from fork(): pid)
    - When > 0
      - Running in original **(parent)** process
      - Return value is child’s process pid
    - When = 0
      - Running in new **child** process
    - When < 0
      - Error (must handle somehow)
      - Running in parent process
fork() example

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main(int argc, char *argv[]) {
    pid_t cpid, mypid;
    pid_t pid = getpid();  // get current processes PID
    printf("Parent pid: %d\n", pid);
    cpid = fork();
    if (cpid > 0) {          /* Parent Process */
        mypid = getpid();
        printf("[%d] parent of [%d]\n", mypid, cpid);
    } else if (cpid == 0) {  /* Child Process */
        mypid = getpid();
        printf("[%d] child\n", mypid);
    } else {
        perror("Fork failed");
    }
}
```
Process Management

• **exit** — terminate a process

• **fork** — copy the current process

• **exec** — change the *program* being run by the current process

• **wait** — wait for a process to finish

• **kill** — send a *signal* (interrupt-like notification) to another process

• **sigaction** — set handlers for signals
exec (program, arguments)

• Used to run program in the current process with specified arguments
  • Load program into address space
  • Copy arguments into address space’s memory
  • Start execution at `start’`
Process Management

• **exit** — terminate a process

• **fork** — copy the current process

• **exec** — change the *program* being run by the current process

• **wait** — wait for a process to finish

• **kill** — send a *signal* (interrupt-like notification) to another process

• **sigaction** — set handlers for signals
wait ()

• Causes the parent process to wait until the child process terminates
  • Parent gets return value from child
  • If no children alive, wait() returns immediately

• Different from exit()
  • exit() called after process terminates
Abstraction III: I/O
Everything is a “File”

• A radical idea
  • Proposed by Dennis Ritchie and Ken Thompson in 1974
  • In their seminal paper on UNIX called “The UNIX Time-Sharing System”

• Core idea: we should have identical interfaces for:
  • Files on disk
  • Networking (sockets)
  • Devices (terminals, printers, etc.)
  • Local interprocess communication (pipes, sockets)

• Based on the system calls **open()**, **read()**, **write()**, and **close()**
**Key Design Ideas**

- **Uniformity:** everything is a file

- **open() before use:** Provides opportunity for access control and arbitration

- **Byte-oriented:** Least common denominator
  - OS hides underlying details:
    - Block-based data transfers? Sure.
    - Stream data transfers? Sure.

- **Kernel buffered read() and write()**
  - Helpful to make everything byte-oriented
  - Process is *blocked* while waiting for return
  - Complete in background
    - Writes return immediately
  - Enables a “global” buffer management (eg., taking caches into account)

- **Explicit close()**
Interprocess Communication

• What if two processes wish to communicate with one another?
  • What are the possible options?

• One option: shared memory address space
  • But the OS enforces protection...
  • Possible, but can be catastrophic

• Another option: use a file
  • Producer (writer) writes to a file; consumer (reader) reads.
  • Better; OS even provides a way:
    • file descriptors are shared between parent & child processes
  • Problem?
    • High overheads

• Other options: IPC and RPC
Interprocess Communication

- A crazy idea: Create an *in-memory* queue
  - Data written by producer process is written to the queue
  - Consumer processes can read from the queue
  - Use a file interface to enable reads and writes
    - Recall: file descriptors are shared between parent & child processes
  - Done!?!?

- Allowing the processes to access the queue as and when they want leads to...
  - Potential protection violation (it is shared memory after all)
  - What could we do?
    - Suppose we ask the Kernel to help...
    - Use syscalls! Allow accessing the queue via system calls

- Challenge?
  - What if A generates data faster than B can consume it?
  - What if B consumes data faster than A generates it?
“Pipe” for Interprocess Communication

• A crazy idea: Create an *in-memory* queue
  • Data written by producer process is written to the queue
  • Consumer processes can read from the queue
  • Use a file interface to enable reads and writes
    • Recall: file descriptors are shared between parent & child processes
    • Enable accessing the queue via syscalls!

• Challenge?
  • What if A generates data faster than B can consume it?
  • What if B consumes data faster than A generates it?

• Solution: blocked reads and writes!

• This queue is called a “pipe”
  • Has two file descriptors, one for executing each of read and write
“Sockets” for Remote Interprocess Communication

• What if the two processes are running on two different physical servers?
  • With a network sitting in the middle?
  • What could we do?

• Sockets!
  • Create an in-memory queue at each process
  • Exactly the same semantics as a file
  • Ensure the correct “semantics” between the two queues
    • Data read at the consumer has exactly the same ordering as the data written by the producer

• The correctness is enabled by the OS
  • Using a reliable, in-order, delivery protocol for data transfer over the network