Introduction to Operating Systems and Concurrency

The operating system (OS) is the interface between the user and the hardware.

An OS implements a sort of *virtual machine* that is easier to program than the raw hardware.
The OS and the Hardware

The OS is the “permanent” software with the power to:

- control/abstract/mediate access to the hardware
  - CPUs and memory
  - I/O devices
- so user code can be:
  - simpler
  - device-independent
  - portable
  - even “transportable”

Memory and the CPU

- CPU
  - registers
- Main Memory
  - OS code
  - OS data
  - Program A
  - Data
  - Program B
  - Data
  - code library
  - main memory
The Big Questions

The basic issues/questions in this course are how to:

• allocate memory and storage to multiple programs?
• share the CPU among concurrently executing programs?
• *suspend* and *resume* programs?
• share data safely among concurrent activities?
• protect one executing program’s storage from another?
• protect the code that implements the protection, and mediates access to resources?
• prevent programs from taking over the machine?
• allow programs to interact safely?

A First Look at Some Key Concepts

**kernel**

The software component that controls the hardware directly, and implements the core privileged OS functions.

Modern hardware has features that allow the OS kernel to protect itself from untrusted user code.

**thread**

An executing stream of instructions and its CPU register context.

**virtual address space**

An execution context for thread(s) that provides n independent name space for addressing some or all of physical memory.

**process**

An execution of a program, consisting of a virtual address space, one or more threads, and some OS state.
The Kernel

• Today, all “real” operating systems have protected kernels.
  The kernel resides in a well-known file: the “machine”
  automatically loads it into memory (boots) on power-on/reset.
  Our “kernel” is called the executive in NT.

• The kernel is (mostly) a library of service procedures shared
  by all user programs, but the kernel is protected:
  User code cannot access internal kernel data structures directly,
  and it can invoke the kernel only at well-defined entry
  points (system calls).

• Kernel code is like user code, but the kernel is privileged:
  The kernel has direct access to all hardware functions, and
  defines the machine entry points for interrupts and exceptions.

A Protected Kernel

CPU mode (a field in some status register) indicates
whether the CPU is running in a user
program or in the protected kernel.

Some instructions or
data accesses are
only legal when the
CPU is executing in
kernel mode.
Processes and the Kernel

A thread is a schedulable stream of control.
- defined by CPU register values (PC, SP)
- suspend: save register values in memory
- resume: restore registers from memory

Multiple threads can execute independently:
- They can run in parallel on multiple CPUs...
  - physical concurrency
- …or arbitrarily interleaved on a single CPU.
  - logical concurrency

Each thread must have its own stack.
Threads vs. Processes

1. The process is a kernel abstraction for an independent executing program.
   - includes at least one “thread of control”
   - also includes a private address space (VAS)
     - requires OS kernel support
       (but some use process to mean what we call thread)

2. Threads may share an address space
   - threads have “context” just like vanilla processes
     - thread context switch vs. process context switch
   - every thread must exist within some process VAS
   - processes may be “multithreaded”

Why Threads Are Important

1. There are lots of good reasons to use threads.
   - “easy” coding of multiple activities in an application
     - e.g., servers with multiple independent clients
     - parallel programming to reduce execution time

2. Threads are great for experimenting with concurrency.
   - context switches and interleaved executions
   - race conditions and synchronization
     - can be supported in a library (Nachos) without help from OS

3. We will use threads to implement processes in Nachos.
   - (Think of a thread as a process running within the kernel.)
Concurrency

Working with multiple threads (or processes) introduces *concurrency*: several things are happening “at once”.

How can I know the order in which operations will occur?

- **physical concurrency**
  
  On a *multiprocessor*, thread executions may be arbitrarily interleaved at the granularity of individual instructions.

- **logical concurrency**
  
  On a *uniprocessor*, thread executions may be interleaved as the system switches from one thread to another.

*context switch* (suspend/resume)

**Warning**: concurrency can cause your programs to behave unpredictably, e.g., crash and burn.

---

**Logical Concurrency Illustrated**

- logical concept

- reality

*context switch*
Context Switches: Voluntary and Involuntary

On a uniprocessor, the set of possible execution schedules depends on when context switches can occur.

• **Voluntary**: one thread explicitly yields the CPU to another.
  
  E.g., a Nachos thread can suspend itself with `Thread::Yield`.
  
  It may also block to wait for some event with `Thread::Sleep`.

• **Involuntary**: the system scheduler suspends an active thread, and switches control to a different thread.
  
  Thread scheduler tries to share CPU fairly by *timeslicing*.
  
  Suspend/resume at periodic intervals (e.g., `nachos -rs`)

  *Involuntary context switches can happen “any time”.

The Dark Side of Concurrency

With interleaved executions, the order in which processes execute at runtime is *nondeterministic*.

- depends on the exact order and timing of process arrivals
- depends on exact timing of asynchronous devices (disk, clock)
- depends on scheduling policies

Some schedule interleavings may lead to incorrect behavior.

- Open the bomb bay doors *before* you release the bomb.
- Two cooks can’t both stir the same pan at the same time.

The system must provide a way to coordinate concurrent activities to avoid incorrect interleavings.
Example: A Concurrent Color Stack

InitColorStack() {
    push(blue);
    push(purple);
}

PushColor() {
    if (s[top] == purple) {
        ASSERT(s[top-1] == blue);
        push(blue);
    } else {
        ASSERT(s[top] == blue);
        ASSERT(s[top-1] == purple);
        push(purple);
    }
}

Interleaving the Color Stack #1

PushColor() {
    if (s[top] == purple) {
        ASSERT(s[top-1] == blue);
        push(blue);
    } else {
        ASSERT(s[top] == blue);
        ASSERT(s[top-1] == purple);
        push(purple);
    }
}

ThreadBody() {
    while(1)
        PushColor();
}
Interleaving the Color Stack #2

```c
if (s[top] == purple) {
    ASSERT(s[top-1] == blue);
    push(blue);
} else {
    ASSERT(s[top] == blue);
    ASSERT(s[top-1] == purple);
    push(purple);
}
```

Interleaving the Color Stack #3

```c
if (s[top] == purple) {
    ASSERT(s[top-1] == blue);
    push(blue);
} else {
    ASSERT(s[top] == blue);
    ASSERT(s[top-1] == purple);
    push(purple);
}
```

Consider a yield here on blue’s first call to PushColor().
Interleaving the Color Stack #4

```c
if (s[top] == purple) {
    ASSERT(s[top-1] == blue);
    push(blue);
} else {
    ASSERT(s[top] == blue);
    ASSERT(s[top-1] == purple);
    push(purple);
}
```

Consider yield here on blue’s first call to PushColor().

Race Conditions Defined

1. Every data structure defines *invariant* conditions.
   defines the space of possible *legal* states of the structure
   defines what it means for the structure to be “well-formed”
2. Operations depend on and preserve the invariants.
   The invariant must hold when the operation begins.
   The operation may temporarily violate the invariant.
   The operation restores the invariant before it completes.
3. Arbitrarily interleaved operations violate invariants.
   Rudely interrupted operations leave a mess behind for others.
4. Therefore we must constrain the set of possible schedules.