# Introduction (ch. 2) part 2

Operating Systems
Based on: Three Easy Pieces by Arpaci-Dusseaux

Moshe Sulamy

Tel-Aviv Academic College

# Concurrency

- OS is working on many things at once:
  - Each program has “its own” CPU and RAM
  - Many programs run at the same time
- Modern **multi-threaded** programs exhibit the same problems
- Concurrency is everywhere

# Concurrency

threads.c:

```
volatile int counter = 0;
int loops;

void* worker(void *arg) {
    for (int i = 0; i < loops; ++i)
        ++counter;
    return NULL;
}

int main(int argc, char *argv[]) {
    loops = atoi(argv[1]);
    pthread_t p1, p2;
    printf("Initial value : %d\n", counter);
    pthread_create(&p1, NULL, worker, NULL);
    pthread_create(&p2, NULL, worker, NULL);
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf("Final value   : %d\n", counter);
}
```

# Concurrency

- The program creates two **threads**
  - Thread: a function running concurrently within the same address space
- Each thread executes `worker()`
  - Increments a global (shared) counter
- `loops`: how many times to increment the counter

# Concurrency

`loops = 1000`:

```
promp> gcc -o thread thread.c -Wall -pthread
prompt> ./thread 1000
Initial value : 0
Final value   : 2000
```

`loops = 100000000`:

```
prompt> ./thread 100000000
Initial value : 0
Final value   : 196445738 // huh??
prompt> ./thread 100000000
Initial value : 0
Final value   : 197944967 // what the??
```

# Concurrency

- Three instructions to increment a counter:
  - Load the value into a register
  - Increment the register
  - Store the register back into memory
- Does not execute **atomically** (without interference)
- A problem of **concurrency**

# Persistence

- System memory (such as DRAM) is **volatile**
  - Data is lost when power goes away or the system crashes
- We need hardware and software to store data **persistently**

# Persistence

- Hardware: **I/O device** such as a **hard drive** or **SSD**
- Software: **file system**
  - Manages the disk
  - Responsible for storing user files

- No private, virtualized disk
- Files can be viewd as virtualized disks.
  - Users want to **share** information that is in files

# Persistence

```c
int main(int argc, char *argv[]) {
    int fd = open("/tmp/file", O_WRONLY|O_CREAT,
        S_IRWXU);
    assert(fd > -1);
    int rc = write(fd, "hello world\n", 12);
    assert(rc == 12);
    close(fd);
}
```

- The program makes three calls:
  - `open()`: opens (and creates) the file
  - `write()`: writes data to the file
  - `close()`: closes the file

# Persistence

- These are **System calls**
  - Routed to part of the OS called the **file system**
  - Handles requests and returns error code
  - Like a **standard library** for OS operations
- The **file system**:
  - Figures out where on disk the new data will reside
  - Updates various structures
  - Issues I/O requests to the underlying storage device

# Kernel

- The **kernel** is a core part of the OS:
  - Always in memory
  - Executed in response to events
    - External events (**interrupts**), e.g., clock
    - Requests from running programs (**system calls**)
- What we think of as "OS" is not always part of the kernel
  - e.g., the Unix Shell is an application
- The kernel is an **event driven** program

# Kernel - Event Handler

- An **event**: mouse is moved, key is pressed, network communication, division by zero, system call, etc.
- **Interrupts** the current program, executes kernel code
- The kernel defines a **handler** for each event type

# Events

- **Interrupt** - asynchronous (external) event
  - For example: key pressed
  - Kernel stores it, can be checked later
- **Trap** - synchronous (internal) event
  - Also **exception** or **fault**
  - For example: division by zero, kernel terminates program
  - Not necessarily for errors!

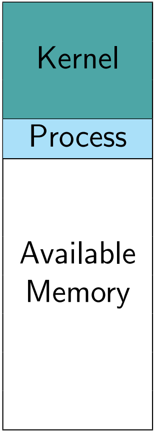
# System Calls

- User program wants to invoke OS - places a **system call**
  - For example: open a file, allocate memory, get keyboard input
- Special instruction that causes a **trap**
- Calls a procedure in the **kernel**
  - The specific **event handler**

# Physical Memory

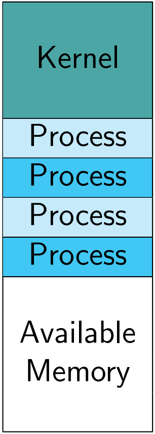

- The kernel resides in memory

# Physical Memory



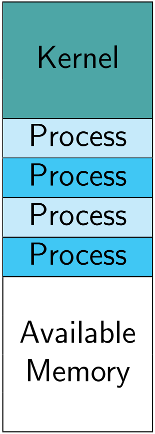

- The code & data of each running program (a **process**) is loaded into memory
  - RAM is divided: user, kernel
  - **System call**: invoke kernel code, then return to user code

# Physical Memory




- Several processes can exist in parallel
  - **Memory protection**: each process is seemingly alone

# Physical Memory




- Several processes can exist in parallel
  - **Memory protection**: each process is seemingly alone
- This is a major simplification
  - But it suffices for now

# Design Goals

- Abstraction
  - Dividing into small, understandable pieces
  - Make the system convenient and easy to use
- Performance
  - Minimize the overhead of the OS
  - Provide virtualization without excessive overheads
- Protection
  - Malicious or bad behavior of one application does not harm others or the OS
  - **Isolation** of processes
- Reliability
  - The OS must run non-stop

# Design Goals

- Other goals:
  - Energy efficiency
  - Security
  - Mobility

# Some History

- Early OS: Just libraries
  - Commonly-used functions, e.g., low-level I/O
  - No abstraction, no virtualization
  - One program at a time
  - **Batch mode**

# Some History

- Early OS: Just libraries
  - Commonly-used functions, e.g., low-level I/O
  - No abstraction, no virtualization
  - One program at a time
  - **Batch mode**
- Beyond Libraries: Protection
  - **User mode** with hardware restrictions
  - **System call**: instead of a library procedure
    - Raises privilege to **kernel mode**
    - OS has full access to hardware

# Some History

- Era of Multiprogramming
  - Make better use of machine resources
  - Load a number of jobs, switch rapidly between them
  - **Context switch**, **concurrency**
  - **Memory protection** became important

# Some History

- Era of Multiprogramming
  - Make better use of machine resources
  - Load a number of jobs, switch rapidly between them
  - **Context switch**, **concurrency**
  - **Memory protection** became important
- The Modern Era
  - The **PC**: the dominant force in computing

# Summary

- The OS: abstraction & resource management
- Multiprogramming & concurrency via context switching and memory protection
- Kernel: OS code & data that is always in memory
    - Not a running program, but pieces of code executed in response to events
- System calls: user events to trigger kernel code to act on their behalf