CPS 310

Process = Address Space + Thread(s)

Jeff Chase
Duke University

http://www.cs.duke.edu/~chase/cps310
HOW TO WRITE GOOD CODE:

START PROJECT.

DO THINGS RIGHT OR DO THEM FAST?

FAST

RIGHT

FAST

CODE WELL

DOES IT WORK YET?

ALMOST, BUT IT'S BECOME A MASS OF KLUDGES AND SPAGHETTI CODE.

NO

ARE YOU DONE YET?

NO, AND THE REQUIREMENTS HAVE CHANGED.

THROW IT ALL OUT AND START OVER.

GOOD CODE
The story so far: process and kernel

- A (classical) OS lets us run programs as processes. A **process** is a running program instance (with a **thread**).
  - Program code runs with the CPU core in untrusted **user mode**.
- Processes are protected/isolated.
  - **Virtual address space** is a “fenced pasture”
  - **Sandbox**: can’t get out. **Lockbox**: nobody else can get in.
- The OS kernel controls **everything**.
  - Kernel code runs with the core in trusted **kernel mode**.
Key Concepts for Classical OS

- **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 instruction path and its CPU register state.

- **virtual address space**
  - An execution context for thread(s) defining a name space for executing instructions to address data and code.

- **process**
  - An execution of a program, consisting of a virtual address space, one or more threads, and some OS kernel state.
Entry to the kernel

Every entry to the kernel is the result of a **trap**, **fault**, or **interrupt**. The core switches to kernel mode and transfers control to a handler routine.

The handler accesses the core register context to read the details of the exception (trap, fault, or interrupt). It may call other kernel routines.
The kernel can create and launch a process by setting up a data structure (VAS+register context) in kernel-space memory, and “pointing the machine at it”. The kernel initializes registers and VAS state for a process/thread to run untrusted code, and transfers control into it.
Processes and their threads

Each process has a virtual address space (VAS): a private name space for the virtual memory it uses.

The VAS is both a "sandbox" and a "lockbox": it limits what the process can see/do, and protects its data from others.

Each process has a **main thread** bound to the VAS, with a stack.

If we say a process does something, we really mean its thread does it.

The kernel can suspend/restart a thread wherever and whenever it wants.

On real systems, a process can have multiple threads.

We presume that they can all make system calls and **block** independently.

If nothing is wrong, each thread is running! If something is wrong...

- **STOP**
- **wait**
A process can have multiple threads

volatile int counter = 0;
int loops;

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

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "usage: threads <loops>
");  
        exit(1);
    }
    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);
    return 0;
}

Much more on this later!
Running a program is like performing a play.
The sheep analogy

Address space
“fenced pasture”

Thread

Code and data
The core-and-driver analogy

The machine has a bank of CPU cores for threads to run on.

The OS allocates cores to threads (the “drivers”).

Cores are hardware. They go where the driver tells them.

OS can force a switch of drivers any time.
Threads drive cores
What?

- A process is a running program.
- A running program (a process) has at least one thread ("main"), but it may (optionally) create other threads.
- The threads execute the program ("perform the script").
- The threads execute on the "stage" of the process virtual memory, with access to a private instance of the program’s code and data.
- A thread can access any virtual memory in its process, but is contained by the "fence" of the process virtual address space.
- Threads run on cores: a thread’s core executes instructions for it.
- Sometimes threads idle to wait for a free core, or for some event. Sometimes cores idle to wait for a ready thread to run.
- The operating system kernel shares/multiplexes the computer’s memory and cores among the virtual memories and threads.
More analogies: threads and stacks

- Threads drive their cores on paths across the stage.
- Each thread chooses its own path. (Determined by its program.)
- But they must leave some “bread crumbs” to find their way back on the return!
- Where does a thread leave its crumbs? **On the stack!**
  - Call frames with local variables
  - Return addresses

This means that **each thread must have its own stack**, so that their crumbs aren’t all mixed together.
Thread context

- Each thread has a **context** (exactly one).
  - Context == values in the thread’s registers
  - Including a (protected) identifier naming its VAS.
  - And a pointer to thread’s **stack** in VAS/memory.
- Each core has a context (at least one).
  - Context == a register set that can hold values.
  - The register set is baked into the hardware.
- A core can change “drivers”: **context switch**.
  - Save running thread’s register values into memory.
  - Load new thread’s register values from memory.
  - Enables **time slicing** or **time sharing** of machine.
Two threads

"on deck" and ready to run

running thread

CPU (core)

registers

Virtual memory

program

code library

data

stack

Register values saved in memory
Running code can suspend the current thread just by saving its register values in memory. Load them back to resume it at any time.
More analogies: context/switching

1. Page links and back button navigate a “stack” of pages in each tab.
2. Each tab has its own stack. One tab is active at any given time. You create/destroy tabs as needed. You switch between tabs at your whim.
3. Similarly, each thread has a separate stack. The OS switches between threads at its whim. One thread is active per CPU core at any given time.
What causes a context switch?

There are three possible causes:

1. Preempt (yield). The thread has had full use of the core for long enough. It has more to do, but it’s time to let some other thread “drive the core”.
   - E.g., timer interrupt, quantum expired → OS forces yield
   - Thread enters Ready state, goes into pool of runnable threads.

2. Exit. Thread is finished: “park the core” and die.

3. Block/sleep/wait. The thread cannot make forward progress until some specific occurrence takes place.
   - Thread enters Blocked state, and just lies there until the event occurs. (Think “stop sign” or “red light”.)
Switching out

- What causes a core to switch out of the current thread?
  - Fault+sleep or fault+kill
  - Trap+sleep or trap+exit
  - Timer interrupt: quantum expired
  - Higher-priority thread becomes ready
  - ...?

**Note:** the thread switch-out cases are sleep, forced-yield, and exit, all of which occur in kernel mode following a trap, fault, or interrupt. But a trap, fault, or interrupt does not necessarily cause a thread switch!
What cores do

scheduler
getNextToRun()

nothing?

pause

idle

got thread

get thread

ready queue (runqueue)

timer quantum expired?

sleep? exit?

run thread

switch in

switch out

put thread
Wait state

Dude, here's the file you wanted. Now go get ready!

OS

(file)

Times up!

This walk out

OS

(core)

(core)

(TCB)

Ready state

SO EXCITED...

I'm so ready!

(os)

PC
What is a Virtual Address Space?

- **Protection domain**
  - A “sandbox” for threads that limits what memory they can access for read/write/execute.
  - A “lockbox” that limits which threads can access any given segment of virtual memory.

- **Uniform name space**
  - Threads access their code and data items without caring where they are in machine memory, or even if they are resident in memory at all.

- **A set of V→P translations**
  - A level of indirection mapping virtual pages to page frames.
  - The OS kernel controls the translations in effect at any time.
Memory Allocation

How should an OS allocate its memory resources among contending demands?

- Virtual address spaces: fork, exec, sbrk, page fault.
- The kernel controls how many machine memory frames back the pages of each virtual address space.
- The kernel can take memory away from a VAS at any time.
- The kernel always gets control if a VAS (or rather a thread running within a VAS) asks for more.
- The kernel controls how much machine memory to use as a cache for data blocks whose home is on slow storage.
- Policy choices: which pages or blocks to keep in memory? And which ones to evict from memory to make room for others?
Virtual Address Translation

Example only: a typical 32-bit architecture with 4KB pages.

Virtual address translation maps a virtual page number (VPN) to a page frame number (PFN) in machine memory: the rest is easy.

Deliver fault to OS if translation is not valid and accessible in requested mode.
Virtual memory faults

- Machine memory is “just a cache” over files and segments: a page fault is “just a cache miss”.
  - Machine passes faulting address to kernel (e.g., x86 control register CR2) with fault type and faulting PC.
  - Kernel knows which virtual space is active on the core (e.g., x86 control register CR3).
  - Kernel consults other data structures related to virtual memory to figure out how to resolve the fault.
  - If the fault indicates an error, then signal/kill the process.
  - Else construct (or obtain) a frame containing the missing page, install the missing translation in the page table, and resume the user code, restarting the faulting instruction.

The x86 details are examples: not important.
Virtual Addressing: Under the Hood

How to monitor page reference events/frequency along the fast path?
“Limited direct execution”

User code runs on a CPU core in user mode in a user space. If it tries to do anything weird, the core transitions to the kernel, which takes over.

The kernel executes a special instruction to transition to user mode (labeled as “u-return”), with selected values in CPU registers.
An analogy

• Each thread/context transfers control from user process/mode to kernel and back again.
• User can juggle ball (execute) before choosing to hit it back.
• But kernel can force user to return the ball at any time.
• Kernel can juggle or hide the ball (switch thread out) before hitting it back to user.
• Kernel can drop ball at any time.
• Kernel is a multi-armed robot who plays many users at once.
• At most one ball in play for each core/slot at any given time.
The kernel must be bulletproof

Secure kernels handle system calls verrry carefully.

Syccalls indirect through syscall dispatch table by syscall number. No direct calls to kernel routines from user space!

What about references to kernel data objects passed as syscall arguments (e.g., file to read or write)?

Use an integer index into a kernel table that points at the data object. The value is called a handle or descriptor. No direct pointers to kernel data from user space!

Kernel copies all arguments into kernel space and validates them.

Kernel interprets pointer arguments in context of the user VAS, and copies the data in/out of kernel space (e.g., for read and write syscalls).
Kernel Stacks and Trap/Fault Handling

Threads execute user code on a **user stack** in user space (the process virtual address space).

Each thread has a second **kernel stack** in **kernel space** (VM accessible only in kernel mode).

System calls and faults run in kernel mode on a kernel stack for the current thread.

Kernel code running in P’s process context has access to P’s virtual memory.

The syscall (trap) handler makes an indirect call through the system call dispatch table to the handler registered for the specific system call.
Process management

- OS offers system call APIs for managing processes.
  - Create processes (children)
  - Control processes
  - Monitor process execution
  - “Join”: wait for a process to exit and return a result
  - “Kill”: send a signal to a process
  - Establish interprocess communication (IPC: later)
  - Launch a program within a process

- We study the Unix process abstraction as an example.
  - Illustrative and widely used for 40+ years!
  - Use it to build your own shell.
The essence of Unix process “fork”

Oh Ghost of Walt, please don’t sue me.
Unix fork/exit syscalls

```plaintext
int pid = fork();
Create a new process that is a clone of its parent. Return child process ID (pid) to parent, return 0 to child.

exit(status);
Exit with status, destroying the process. Status is returned to the parent.
Note: this is not the only way for a process to exit!
```
int pid;
int status = 0;

if (pid = fork()) {
    /* parent */
    ....
} else {
    /* child */
    ....
    exit(status);
}
int pid;
int status = 0;

if (pid = fork()) {
   /* parent */
   ..... 
   pid = wait(&status);
} else {
   /* child */
   ..... 
   exit(status);
}
A simple program: parallel

```c
... int main(...arg N...) {
    for 1 to N
dofork();
    for 1 to N
    wait(...);
}

void child() {
    BUSYWORK {x = v;}
    exit(0);
}
...
```

*Parallel* creates N child processes and waits for them all to complete.

Each child performs a computation that takes, oh, 10-15 seconds, storing values repeatedly to a global variable, then it exits.

How does N affect completion time?

```bash
chase$ cc –o parallel parallel.c
chase$ ./parallel
???
chase$
```
A simple program: parallel

Completion time (ms)

Completion time (ms)

Three different machines

N (# of children)
Parallel: some questions

• Which machine is fastest?
• How does the total work grow as a function of N?
• Does completion time scale with the total work? Why?
• Why are the lines flatter for low values of N?
• How many cores do these machines have?
• Why is the timing roughly linear, even for “odd” N?
• Why do the lines have different slopes?
• Why would the completion time ever drop with higher N?
• Why is one of the lines smoother than the other two?
• Can we filter out the noise?