Processes

OS Lecture 2

UdS/TUKL WS 2015

Who am I?

- Björn Brandenburg
    - bbb@mpi-sws.org
    - http://www.mpi-sws.org/~bbb
- Head of the Real-Time Systems Group @ MPI-SWS (since 2011)
- I work on real-time operating systems
    - LITMUS^{RT}: http://www.litmus-rt.org

Announcements

1. Make sure you are subscribed to the **mailing list** (see homepage).
   » http://courses.mpi-sws.org/os-ws15/
2. Need to form a **two-person team** for assignments
   » First assignment out next Monday
3. Send email to course mailing list if you are looking for a partner
4. Reminder: take your own notes and **ask questions**.

Every modern general-purpose OS has a notion of a *process*.

- What is a process?
- Why have them?

Many roles of processes

- A **computation in progress**
- A **sphere of isolation**: process = program + “everything that it can affect or be affected by”
- security & protection, scheduling, resource accounting, …

A basic unit for **system organization/decomposition**:

- *complex*, *concurrent* activities = many *simple*, *sequential* processes that are being interleaved

Idealized **abstraction**:

- programmer is not aware of actual processor complexities, and no need to worry about other processes

What is a process?

1. Historical perspective:
   - **virtualize the processor**
   - Evolution: single jobs ➞ batch job processing ➞ multiprogramming ➞ time-share systems
2. Modern perspective:
   - Key abstraction for **decomposition**
   - Sequential **computation in progress**

Historical Perspective

Original motivation: sharing expensive computers

- First computers: (manually) load program; run; output results; repeat
  → No abstraction at all!
- Idea: *virtualize* processor & memory
  → give each running program a *virtual processor*

process = program running on a (virtual) processor process ≠ program

process ≠ program

Each program may be executed multiple times.
→ 1 program – n processes

More than a program:

- computation *in progress*
- program + resources + state “how far we’ve gotten and how to continue”

Less than a program:

- What looks like a single “program” to the user can consist of many processes (e.g., `gcc`).

What’s in a process?

What does the OS have to keep track of?

What’s in a process?

Two key aspects:

» Computation in **progress**
→ “how far we’ve gotten and how to continue”

» Sphere of **isolation**
→ “things that it can affect or be affected by”

Computation in progress:

- » **program counter**, indicating next instruction
- » register file: set of **CPU registers** + current values
- » the **stack**: state of incomplete function calls

Sphere of isolation:

- » the **text segment**: code for the running program
- » the **heap**: data of the running program
- » set of **OS resources** (files, network connections, credentials, …)

Modern Perspective (1/3)

*Can you have more than one **computation in progress** in the same **sphere of isolation**?*

Yes.

- **threads** of execution or
- **lightweight processes** (LWP)

Now ubiquitous. Historically, only a single thread per process.

Modern Perspective (2/3)

Two **completely orthogonal** concepts:

1. *protection domains* (= spheres of isolation)
   - often (incorrectly) called "address spaces"
2. *threads* (= sequential computations in progress)
   - each thread executes in some protection domain
   - We will discuss threads in more detail later.

Modern Perspective (3/3)

Almost any combination possible:

- 1 protection domain, 1 thread (classic process)
- 1 protection domain, many threads
  - multithreaded process
  - DOS, Classic Mac OS, many embedded systems

But also:

- 1 thread, many protection domains (thread migration)
- 1 protection domain, 0 threads (why?)

How is the processor virtualized?

One physical CPU, one set of registers
→ many “running” processes?

Processes are sequential

- Only one computation step at a time on a (virtual) processor
- Concurrency: the OS *interleaves* execution of processes on physical processor
- **Context switch**: *preempt* current process and *dispatch* another
- Typically, a process is the basic unit of *scheduling*
- scheduling vs. dispatching

Process state

OS maintains a state machine for each process:

» READY: can be dispatched by scheduler
» RUNNING: currently executing on a processor
» WAITING: cannot proceed in execution until some event occurs (e.g., waiting for I/O to complete)

There is always *some* process running, perhaps the *idle process*.

On each processor, only one process can run at a time.

Process Control Block (PCB)

OS stores all relevant information about a process in the **PCB**.
(It's just a struct with a special name.)

- process ID
- process state
- copies of register values (for context switch)
- memory state (which memory may be accessed)
- scheduling information
- accounting information
- user information
- …

Process Management (1/2)

- OS maintains several queues, depending on process state
  - ready queue(s) managed by scheduler
  - queues of waiting processes
- each PCB is queued on some queue
- allocate & initialize PCB when process is created, deallocate when process terminates

Process Management (2/2)

- How to initialize?
  → `fork()` vs. `CreateProcess()`
- How to allocate?
  - General Purpose OS (GPOS)
    → dynamic allocation (kernel heap)
    → as many processes as needed (memory limit)
  - Real-Time OS (RTOS) / embedded OS
    → statically allocated array of PCBs
    → max. number processes known at design time

Multiprogramming vs. Time-sharing

Multiprogramming:

- More than one process can exist at a time
- Context switches at coarse granularity
- Some processes *swapped out* altogether

Time-sharing:

- multiple ready processes supported
- frequent context switches so that processes appear to "run at the same time" to human observer

How does a context switch work?

How does a context switch work?

Switching from *prev* to *next*.

1. Store all register contents, processor flags, etc. in PCB of *prev*.
   - alternatively, push all registers on stack
2. Overwrite CPU's stack register (`SP`) with *next*'s stack pointer (stored in PCB).
3. Restore all register contents, processor flags, etc. from copy in *next*'s PCB
   - alternatively, pop all registers from stack
4. Return from function call (to return address on *next*'s stack!!!)

```
switch_to(next):
    push R1 // <--- save all registers on prev's stack
    push R2
        ...
    push Rn

    mov <next.stack_ptr>, SP // <--- the actual context switch,
                                     now next is running

    pop Rn // <--- restore all registers from next's stack
        ...
    pop R2
    pop R1
    ret <--- return to whatever next was doing before preemption
```

```
prev:                                   next:
push R1
push R2
    ...
push Rn
mov <next.stack_ptr>, SP // <--- the actual context switch

                                        pop Rn
                                            ...
                                        pop R2
                                        pop R1
                                        ret
                                        [....some computation....]
                                        [calls switch_to(prev)]
                                        push R1
                                        push R2
                                            ...
                                        push Rn
                                        mov <prev.stack_ptr>, SP
pop Rn // <--- restore all registers from prev's stack
    ...
pop R2
pop R1
ret <--- return to whatever prev was doing before preemption
```

How to make sure a process does not destroy OS data structures?

e.g., accounting or scheduling information

Kernel mode vs. user mode

- modern processors have (at least) two modes
- kernel mode: unrestricted access to hardware and privileged instructions & registers
- user mode: certain registers and privileged instructions off limits
  - enforced by hardware
  - ensures process executing in user mode cannot access memory belonging to kernel
- dispatcher *switches mode* from kernel mode to user mode before continuing next process

How to regain control?

How to transfer control back to the OS kernel / dispatcher when a user process runs?

Return control to kernel

Problem: At some point, we must stop execution of a user-mode process and return to kernel mode.

- Process may be stuck in `while (true);` loop
- Process may do something invalid, e.g., divide by zero

Solution: hardware ensures that certain **well-defined events** automatically transfer execution to kernel mode *at a known location*.

- Override program counter, enable kernel mode, place **status code** in register or on stack.

Types of events

Traps or **exceptions**: synchronous (= internal) events

- system call
- error (illegal instruction, bad address, divide by zero, …)
- page fault (related to virtual memory)

Interrupts: asynchronous (= external) events

- character typed on terminal
- network packet arrived
- disk operation completed
- **timer**: set up by OS to regain control after allowed *timeslice*

How do interrupts work?

Interrupt and Exception Management

- Table of addresses of *interrupt service routines* (ISR) (or *exception handlers*) at location known to processor (e.g., address stored in register)
  - populated by OS during bootup
- on interrupt / trap, the processor
  1. switches to kernel mode
  2. pushes status information & (certain) registers on stack
  3. looks up the appropriate handler corresponding to the interrupt / trap ID and branches to ISR
- Interrupts can be temporarily *disabled* or *masked*; traps typically cannot be suppressed.

Completely Isolated Processes

Benefits and properties?

Original Goal: Isolation

- complete isolation = processes are *independent*
- sequential & independent = deterministic
  - output determined solely by input
  - reproducible
  - can pause and restart without ill effects
- Can load systems with arbitrary processes and arbitrary number of processes without changing results of computations (modulo memory limits and differences in response times)

Terminology

uniprogramming: one process at a time, run to completion

multiprogramming: multiple processes, one processor, interleaved

multiprocessing: multiple processes, multiple processors

- each process on at most one processor at a time
- processes may *migrate*: run on different processors at different times
- easy to do with independent processes

Cooperating Processes

What are reasons to give up on independence? Effects?

Cooperating Processes: Why?

- Computers reflect social structure — humans interact (email, shared files, etc.)
- Decomposition — solve a large, complex problem with a collection of simple, sequential, cooperating processes
- Performance — want to efficiently utilize hardware (overlay I/O with useful computation)
  - Example: load next video frame while decoding current

Cooperating Processes: Effects

- shared (system) state: **order of accesses** by different processes is relevant
- output may depend on interleaving of processes
  - system behavior may be *nondeterministic*
  - behavior may be *irreproducible*

Example: Process 1 writes “ABC” to the terminal. Process 2 writes “CBA”. What can happen?

When is it safe to interleave processes?

Not all operations are sensitive

`A = 1; B = 2` has same outcome as `B = 2; A = 1`

» can safely interleave `A = 1 || B = 2`

But `A = B + 1; B = 2 * B` cannot be reordered.

What happens for `A = 1 || A = 2`?

» Can we get 3?

What happens for `A = 0x1 || A = 0x10000`?

Race Condition

Processes “racing” to carry out their conflicting operation.

» outcome of computation depends on order of interleaving and relative speed of processes

» don’t know what exactly will happen

» difficult to reason about

» common source of bugs

Atomic Operations

Cannot be interrupted "in the middle" of execution.

Example: suppose writes to 16-bit *aligned* words in memory are atomic.

- If `A` is a **16-bit** variable stored in an aligned word, then `A = 1 || A = 2` never yields `A == 3`
- If `A` is a **32-bit** variable stored in an aligned word, then `A = 0x1 || A = 0x10000` *can* yield `A == 0x10001`!

Where do atomic operations come from?

Where do atomic operations come from?

Fixed set of atomic ops *provided by hardware*.

Common examples:

» word-aligned load/store are typically atomic

» fetch-and-increment

» test-and-set

» compare-and-exchange (or compare-and-swap, `CAS`)

On a *uniprocessor*, anything between two interrupts is atomic: → interrupts *masked* / *disabled* = atomic.

The OS Approach

The set of available hardware primitives:

- is fairly limited (e.g., often no `CAS2`)
- differs from machine to machine
- is difficult to use correctly

Solution:

- Provide a *higher-level abstraction* at the OS level
- Nice, portable semantics for user processes
- Realized in the kernel with available hardware primitives