Processes

Gary Kimura
Process management

• This module begins a series of topics on processes, threads, and synchronization
  – this is the most important part of the class, well, except for file systems and disks…

• Processes and process management
  – what are the OS units of ownership / execution?
  – how are they represented inside the OS?
  – how is the CPU scheduled across processes?
  – what are the possible execution states of a process?
    • and how does the system move between them?
  – How are they created?
  – How can this be made faster
A Digression – Mechanism and Policy

• Mechanism – how to do something (schedule a thread, fix a lightbulb)
• Policy – when to do something, who is authorized to do it (network packet arrived for thread, light is burned out by anyone but me)
• Mechanisms should NOT dictate policy.
  – Allows multiple policies for same mechanism (fix lights in batches)
  – Allows multiple mechanisms for same policy (fix lights by myself [unreliable, cheap], call electrician [reliable, expensive])
The Process

• The process is the OS’s abstraction for execution
  – the unit of ownership (root of web/tree of kernel data structures)
  – the unit of execution (sorta)
  – the unit of scheduling (kinda)
  – the dynamic (active) execution context
    • compared with program: static, just a bunch of bytes

• Process is often called a job, task, or sequential process

• The goal of the OS is to present each Process with the view that it is executing in it’s own separate, isolated computer
What is a “process”? 

• Simple, classic case (1950’s): a **sequential process**
  – An address space (abstraction of “memory”)
  – A single bit of execution: a “thread”

• A sequential process is:
  – The unit of execution
  – The unit of scheduling
  – The execution context (registers, OS state, memory, etc.)
What is a process?

- Process == fundamental abstraction for program execution
  - an address space which contains:
    - the code for the running program
    - the data for the running program
  - at least one thread with state
    - Registers, IP
    - Floating point state
    - Stack and stack pointer
  - a set of OS resources
    - open files, network connections, security caches, sound channels, …
- In other words, it’s all the stuff you need to run the program
A process’s address space (overly simplified)

32-bit address space

0x00000000

0x7FFFFFFF

stack
(dynamic allocated mem)

heap
(dynamic allocated mem)

static data
(data segment)

code
(text segment)

IP

SP
The OS’s process namespace

• (Like most things, the particulars depend on the specific OS, but the principles are general)
• The name for a process is called a process ID (PID)
  – An integer (how many bits?), possibly a string(!)
• The PID namespace is global to the system
  – Only one process at a time has a particular PID: uniqueness
• Operations that create processes return a PID
  – E.g., NtCreateProcess, ShellExecute
• Operations on processes take PIDs as an argument
  – E.g., NtOpenProcess
The Process Object

• There’s a data structure called the process object (_KPROCESS in base\ntos\inc\ke.h) that holds all this stuff
  – Processes are identified from user space by a process ID, returned by NtCreateProcess.

• OS keeps all of a process’s hardware execution state in the _KTHREAD (same file) when the process isn’t running
  – IP, SP, registers, etc.
  – when a process is unscheduled (i.e., processor is taken away from the process), the state is transferred out of the hardware into the _KTHREAD

• Note: It is natural to think that there must be some esoteric techniques being used
  – fancy data structures that you’d never think of yourself

Wrong! It’s pretty much just what you’d think of!

Except for some clever assembly code in one place…
_KTHREADs and hardware state

- When a process is running, its hardware state is inside the CPU
  - IP, SP, registers
  - CPU contains current values
- When a process is transitioned to the waiting state, the OS saves its CPU state in the _KTHREAD (actually, _PRCB, but that’s not important 😊)
  - when the OS returns the process to the running state, it loads the hardware registers with values from that process’s _KTHREAD
- The act of switching the CPU from one process to another is called a context switch
  - systems may do 100s or 1000s of switches/sec.
  - takes a few microseconds on today’s hardware
- Choosing which process to run next is called scheduling, more when we talk about threads
Process creation

• New processes are created by existing processes
  – creator is called the parent
  – created process is called the child
  – what creates the first process, and when?

• In some systems, parent defines or donates resources and privileges for its children
  – LINUX/UNIX: child inherits parent’s security context, environment, open file list, etc.
  – NT: all the above are optional (remember, mechanism vs policy), the Windows subsystem provides policy.

• When child is created, parent may either wait for it to finish, or may continue in parallel, or both!
Process Creation 2

• In LINUX, fork/exec pairs.
  – fork() *clones* the current process, duplicates all memory, “inherit” open files
  – exec() throws away all memory and loads new program into memory. Keeps all open files!
  – Very useful, but… wasteful. >99% of all fork() calls followed by exec(). Copy-on-write memory helps but still a big overhead (have to “duplicate” all data structures)

• Windows has parent process doing the work
  – Create process
  – Fill in memory
  – Pass handles
  – Create thread with stack and IP
  – Many system calls (compared with LINUX) but all policy is in user code. More flexible.
Process Destruction

- Privileged operation!
  - Process can always kill itself
  - Killing another process requires permission
- Terminates all threads
- Releases owned resources to known state
  - Files
  - Events
  - Memory
- Notification sent to interested parties
- KPROCESS is freed
So you want to run a process..

• How was it created?
  – Someone wrote some C/C#/C++/etc
  – Compile/fix-errors/compile again
  – Get object files

• What’s in the .o or .obj files?
  – Code and data and fixups
  – Code and data are easy
  – Fixups describe relationships
    • Targets of jumps/calls
    • Data references
  – What do you do about references to other (extern) code/data?
    • Fixups too!
More of what’s in .o / .obj files

• Old style format (reflecting stream view of files)
  – Stream of records <tag><data> where <tag> was
    • DATA: <data> was constant data
    • BSS: <data> was just the size of the BSS reserved
    • CODE: just like DATA
    • FIXUP: applies to previous CODE/DATA record, may list a
      name (external) or an offset into some other prior record and
      describe a width (8, 16, 32, 64) and an operation (ADD, IMM,
      SELF-REL)

• Modern format (take advantage of memory mapping)
  – Header on file describes sections suitable for mmap()
What’s a section?

• Section is a piece of contiguous memory
  – Named
  – Protected: read only, read/write, execute, read/execute, etc.
  – Location in file
  – Location in memory

• Some names are important
  – DATA
  – CODE
  – BSS
  – DEBUG
  – FIXUP
Putting .o/.obj files together

- The “linker”
  - take a collection of object files and produce an executable image
  - Gathers and appends like named/protected sections
  - Evaluates fixups and establishes addressing (linkages) between sections
  - Emits special sections
    - DEBUG
    - IMPORT
    - EXPORT
  - All into a file with the same general format as .o/.obj files
    - A few new sections
    - But it’s header says it’s executable
    - Called the image file
Executing the image file

• What does exec() or CreateProcess() do?
  – Easy stuff:
    • Allocate KPROCESS
    • Create address space
  – Harder stuff
    • Create first thread
    • Copy handle environment from parent
  – The meat:
    • Opens image file
    • Memory maps header (section table of contents)
    • For each section:
      – Memory map the appropriate portion of the file
      – Into the correct address space location
      – With correct memory protection
Is that all?

• Once upon a time, yes
  – All code was in one file
  – Included all special stuff for calling the OS

• Not nearly useful enough
  – What if system call #’s changed?
  – What about sharing common code between apps?
  – What about 3\textsuperscript{rd} party code?
  – What about extensibility?
Dynamic (aka Shared) Libraries

• **Goal**: break down single images into multiple pieces
  – Independently distributable
  – Breakdown based on functionality / extensibility

• **Implications on image format**
  – Need a way to reference between image files
  – Add IMPORT and EXPORT sections
  – IMPORT lists all functions required by the image file (executable or library)
  – EXPORT lists all functions offered by the image file

• **Big implications on process creation**
Process Creation with libraries

• Easy/Harder stuff still the same
• Hardest stuff:
  – No longer loading just a single file, loading multiple modules
  – Walking each IMPORT table, finding references to modules not yet loaded and loading them
  – Big graph traversal (remember “transitive closure”?)
  – How are linkages established between modules?
Module Linkage

• Naïve approach is to use something similar to fixups
  – Modify the sections to establish linkage
  – Modifies the memory mapped pages
    • Don’t want to modify the original file
    • Copy-on-write
    • Bigger page file
    • More dirty pages in memory

• Work with compiler
  – Observe that inter-module references are always direct
    (never self-relative). Call or pointer reference
  – Keywords in language (or header files) that change direct
    calls into indirect calls and direct addressing into indirect
    addressing.
Efficient Linkage

• Foo( args ) turns into (*import_Foo)( args )
• Gather all import_X addresses into a single section
  – Called IAT (import address table)
  – Usually only a single page in size, not inefficient to dirty
  – Still have to do some big work
• Can we do better?
Binding

• Floating modules
  – No known address
  – IAT required to handle differing locations based on other modules’ locations

• Bind modules to specific locations
  – Section table describes location, mapping is trivial
  – IAT can be pre-built with locations already in mind
  – Zero program-startup fixups

• What’s the issue?
Binding

• What address do you assign?
  – 32 bit address space *seems* large enough
  – XP has >1200 modules.

• What if there’s a *collision*?
  – New release of module grows in size (bug fixes, functionality)
  – Modules produced by two independent companies
  – Loader needs to be robust in the face of this
  – Choose another location
  – Fix up IAT (small number of pages)
A few cheats

• Compiler needs to generate self-relative instructions
  – Otherwise relocation of module would require fixups
  – Works well on x86…
  – Most of XP’s DLL’s can be broken into disjoint groups and addresses assigned to each
Vista cool feature

• “dynamic rebasing”
  – At install time, all modules are rebased to random addresses
  – Just edit the IATs, still have speedy program start
  – What problem would this solve?

• Buffer overflow attacks
  – Operate by overflowing a stack buffer and overwriting a
    return address
  – Knowing where special code might be would allow attacker
    to hijack return to code in a module not directly referenced
  – Not if the module moves…
Windows CreateProcess

• Different from fork/exec.
  – Fork/exec are in kernel mode and embody the entire process creation experience
  – Windows Kernel has
    • NtCreateProcess – creates a new process address space. BUT NO THREAD
    • NtCreateThread – creates a new thread in a given process
    • NtSetThreadInformation – sets execution context for thread (notably stack and PC)
Windows CreateProcess

- CreateProcess is user code in kernel32 module
  - Creates process (NtCreateProcess)
  - Maps in kernel call DLL (ntdll)
  - Maps in image (but no libraries)
  - Creates initial thread
  - Sets thread to initialization routine in ntdll (LdrpInitializeProcess)
    - Go!

- LdrpInitializeProcess does all the memory mapping work
  - Executing in the new image’s context
  - Walking module lists is just memory access
  - Makes NtCreateSection calls
Why not do what Unix did?

• Extensibility
  – Differing loader policies (OS/2, DOS)
  – New loader implementations
  – Smaller kernel

• Simpler kernel loader code