Class 3 Overview

- kernels
- processes: HWP, LWP, threads
- parallel processing overview
The OS kernel

- OS
  - provider of services on
  - resources created by OS (with hardware)

- software of OS must be in memory to run
The OS kernel

- OS
  - provider of services on
  - resources created by OS (with hardware)
- software of OS must be in memory to run
- the OS code that is always in memory (resident) we call the kernel
  - but how big does the kernel have to be?

In the beginning…

- lots of programmers wrote lots of code
  - written in assembler
  - any routine able to call any other
  - little structure
  - one, big blob: monolithic kernels
Layers to the Rescue

- introduce structure by establishing hierarchy of responsibility: **layered kernel**
  - once each layer working, can treat as ‘black-box’ and build new layer on top
  - each layer relatively self-contained
  - most (if not all) layers execute in kernel mode

- but:
  - changes in one layer can affect many others (up and down)
  - hard to customize kernel for particular needs
  - security: lots of layer boundaries to cross
Layered Kernels

hardware
primitive process management
memory management (virt mem)
I/O, device management
IPC
file system
user processes

Less is More

- clean slate: throw everything out of kernel
- reintroduce the absolute barest essentials
- everything else runs as a user process
Less is More

- clean slate: throw everything out of kernel
- reintroduce the absolute barest essentials
- everything else runs as a user process
  - e.g., file system
  - e.g., security system
  - e.g., device drivers
- new, smaller (one hopes!) kernel is called a microkernel

Microkernels

- basic idea:
  - very small set of essential routines next to the hardware
  - everything else interacts via messages: so have client–server model
View of a Microkernel

- client processes
- device drivers
- file system server
- process server
- memory management (virt mem)

file system server sends message via microkernel to device driver to request particular disk operation
Advantages of Microkernels

1. uniform interfaces:
   - all interaction is via standardized messages from ‘client’ to ‘server’ passed through microkernel

2. extensibility:
   - easy to add new features or entire new services
   - don’t have to rebuild kernel to support new things
Advantages of Microkernels

1. uniform interfaces
2. extensibility
3. flexibility:
   - easy to adjust services (add or remove) hence easy to customize

4. portability:
   - processor-specific confined to microkernel
   - support for new architectures easy to (re)write
Advantages of Microkernels

1. uniform interfaces
2. extensibility
3. flexibility
4. portability
5. reliability:
   - small amount of kernel code can be thoroughly tested
   - smaller set of APIs means programmers can learn more thoroughly, use better

6. distributed OS support:
   - ‘parties’ communicating through microkernel need not be on same box
Advantages of Microkernels

1. uniform interfaces
2. extensibility
3. flexibility
4. portability
5. reliability
6. distributed OS support
7. object-oriented OS design:
   - easier to introduce OO design practices with microkernel msg-passing model
   - lead to use of “components”
Microkernel Performance

- which is faster?
  - system call, directly to kernel mode code
  or
  - build message
  - send message to dest (server) via microkernel
  - microkernel handles message passing
  - dest receives, does its thing, produces response msg
  - microkernel handles message passing
  - originator gets answer

Microkernel Performance

- first generation microkernels typical size 300 Kb, 140 system calls
- early microkernels suffered “substantial performance penalty”
  - caused by taking functions out of kernel
  - so now need to do msg passing to use those services
  - how to fix?
Microkernel Performance

- put stuff back into kernel
  - took too much out
  - reintroduce some servers and drivers to the kernel
  - e.g., Mach
    - developed at CMU
    - first widely deployed on NeXT
    - basis of current Mac OS X

- be more radical...

Microkernel Performance

- make microkernel even smaller
  - make microkernel simpler and smaller
  - then microkernel can be faster

- e.g., “L4”
  - typical “2nd generation” microkernel
  - 12 Kbytes total size
  - 7 system calls
  - these run about as fast (sometimes faster) than standard layered UNIX systems
How do microkernels do that?

- generic microkernel provides these functions:
  1. low-level memory management
  2. IPC
  3. I/O and interrupt management

Microkernel Memory Management

- microkernel handles all mapping of real memory to processes
- once mapping done, user processes can manage ‘their’ memory directly
  - shared memory for IPC
- primitives for memory mapping:
  1. **grant**: proc gives memory to another proc
  2. **map**: proc shares memory with one or more procs
  3. **flush**: proc reclaims memory given up previously
Microkernel IPC

- done via messages
- each msg has sender/dest identification address
- each service at an address has a port
  - queue where arriving requests accumulate
- each port has set of capabilities
  - define who can contact this service (process), what they can expect to do with the service
- microkernel maintains port identities, capabilities

- must use some shmem mechanism for msgs
  - otherwise need mem to mem copy: too slow
Microkernels and Interrupts

- treat an interrupt like a msg
- microkernel
  - recognizes interrupt
  - does not handle the interrupt
- microkernel generates msg as result of interrupt
  - sends msg to whatever user-level process handles those messages
- \( \Rightarrow \) handles like any other msg-passing request
  - except request originates within microkernel as result of hardware event

Microkernels: Where are they today?

- in your Macintosh
  - OS X is Mach derivative
- in your PC
  - Win2K, XP are microkernel based
Class 3 Overview

- kernels
- processes: HWP, LWP, threads
- parallel processing overview

Another view of a Process

- a running process **has**, among other things:
  - some stack space,
  - other memory it uses,
  - values in CPU registers
    - GPRs
    - specials like PC, PSW
  - a region of memory all its own in which it runs
  - resources given to it by OS, e.g., open files, signals
- a process **runs** some sequence of instructions
Another view of a Process

- distinct:
  - process has
  - process does
- the term “task” often used to refer to what the process has
- look at the “does” part...

Another view of a Process

- suppose we can watch the process as it wanders through memory:
  - imagine we can leave a dot at each location the process visits for instructions:
Another view of a Process

- and if we connect the dots, we get a stringy line tracing the execution of the process: a thread

---

Threads, Processes

- every process we’ve seen so far has exactly 1 thread of execution
- can we have > 1?
Threads, Processes

- can we have > 1?
- YES
  - but what do we need to make that work?

- each thread needs to have:
  - a PC value
  - a PSW value
  - a GPR set
  - tiny amount of stack space
  - a TID: thread ID

Multi-threaded Processes

- now can have multiple threads of execution in the same memory space
Multi-threaded Processes

- now can have multiple threads of execution in the same memory space

- so all variables are shared across all threads in the process:
  - there is only one copy of the data
Multi-threaded Processes

- now can have multiple threads of execution in the same memory space
- so all variables are shared across all threads in the process:
  - there is only one copy of the data
- great for sharing data between different threads
  - different threads within a process may do different things but need to exchange info
- fraught with peril
  - what 1 thread does to data affects every thread using that data
Multi-threaded Processes

- example of multithreading: an interactive text formatter
  - thread to read keyboard characters
  - thread to apply formatting criteria to characters
  - thread to keep screen display updated with formatting
- not all threads run all the time
  - some have to sleep while waiting for work to do
- but all threads work cooperatively
  - sharing data between subsets of threads

Threads

- so a thread is a unit of execution within a process
- where do threads come from?
  - created by parent threads
Threads

- so a thread is a **unit of execution** within a process
- where do threads come from?
  - created by parent threads
- how do threads synchronize?
  - can wait for events
  - can be controlled by a synchronizing thread

- synchronizing/scheduling of threads can be completely user controlled
Thread Control: a User Issue

- synchronizing/scheduling of threads can be completely user controlled:
  - user process contains ≥ 1 thread(s)
  - switching from 1 thread to another done by user's control thread: kernel not involved
- user-level control over what thread(s) run when
- how expensive is a thread-switch compared to a process-switch?

Threads & Multiprocessing

- when a HWP blocks for file I/O, what happens to execution in the process?
Threads & Multiprocessing

- when a HWP blocks for file I/O, what happens to execution in the process?
- blocked waiting for event to occur, so entire HWP ‘stops’

- how about when a thread blocks for file I/O?
  - suppose we have many threads, and only one is blocking on I/O

Blocked Threads

- where is the thread that blocked?
Blocked Threads

- where is the thread that blocked?
  - one of many in a process

- does the kernel see the thread? or only the process?

Blocked Threads

- does the kernel see the thread? or only the process?

- if the kernel only sees processes, it sees a process blocked on I/O
  - i.e., this kernel doesn’t know about threads within the process

- so entire process blocks
  - including, thus, all threads within the process
Where the Threads are...

- kernels need not know anything about threads
  - all the thread support exists in a set of library functions, e.g., pthreads
- these threads often called user-level threads

Disadvantage:
- when thread blocks, all threads block
- can’t really benefit from multiple CPUs if avbl

Advantages?

User Level Threads

- advantages:
  1. thread switching is very fast, happens entirely within user proc, no mode changes
  2. thread scheduling handled by user so can implement scheduling model best suited to app
  3. library supported, so highly portable
User Level Threads

- jacketing: a way to ‘cheat’ on blocking
  - normally, app makes system call resulting in block
  - app calls user–level function to do system call action
  - this function checks if the system call would block if invoked now; if so, arrange for some other thread to be run
    - when invoking thread again runs, it can check again
  - if wouldn’t block, then proceed with system call

Where Else the Threads are

- move all thread support into kernel: kernel level threads
  - no thread management code in application
  - app uses API to kernel thread functions
    - e.g., linux, Win2K
  - kernel schedules individual threads
    - thread–level, not process–level, scheduling granularity
Kernel level threads

- advantages:
  1. kernel can schedule multiple threads concurrently on multiple CPUs if avbl
  2. blocking one thread need not impact processing of other threads in the process
- disadvantages?
  - thread switching involves kernel intervention, mode changes

Thread Performance

- how do user-level/kernel-level threads compare
  - with each other?
  - with use of separate processes?
- measurements reported by Anderson et. al. (1992)
  - on a VAX where a procedure invocation takes 7μs
  - kernel trap takes 17 μs
  - tested:
    - null-fork: time to create child thread/process that, once created does nothing except exit,
    - signal-wait: time for a thread/process to signal a waiting thread/process and wait for an event
Thread Performance

- results: (times in microseconds)

<table>
<thead>
<tr>
<th>TEST</th>
<th>USER-LEVEL THREADS</th>
<th>KERNEL-LEVEL THREADS</th>
<th>PROCESSES</th>
</tr>
</thead>
<tbody>
<tr>
<td>null-fork</td>
<td>34</td>
<td>948</td>
<td>11300</td>
</tr>
<tr>
<td>signal-wait</td>
<td>37</td>
<td>441</td>
<td>1840</td>
</tr>
</tbody>
</table>

Combined Approach: Solaris

- can user-level and kernel-level threads be combined to provide advantages of each without disadvantages?
- sort of:
  - let the kernel be thread-aware and contain KLTs
  - let each user process be associated with $\geq 1$ KLT
  - each user process may have $\geq 1$ ULT
  - the connection between KLT and ULT is handled by a ‘broker’ that Sun calls a LWP
    - different from what we’ve called a LWP
  - each LWP connected to exactly 1 KLT
Solaris Threads

- User task, 3 (user) threads
- User thread
- LWP

If one ULT blocks, its corresponding LWP blocks.
Solaris Threads

“traditional” HWP

pinned to a CPU

Threads & Processes: Models

- what is relationship between threads and processes?
- NOTE! This is different from what SG&G discuss in section 5.4
  - they talk about thread:kernel_thread relationship
- 1:1
  - each thread of execution has its own unique task space
  - e.g., traditional UNIX model
Threads & Processes: Models

- 1 : 1
- many : 1
  - many threads present in 1 task space
  - threads can be created & managed within the task space
  - e.g., WinNT, linux, Solaris, OS/390, Mach

- 1 : many
  - 1 thread may migrate across/through several different processes, possibly across systems
  - e.g., Ra (Clouds), Emerald
Threads & Processes: Models

- 1 : 1
- many : 1
- 1 : many
  - many : many
    - combines features of previous 2 models
    - e.g., TRIX

Java: new twists on threads

- Java provides thread support in language
  - 'not' in library
  - not in 'kernel'
- has class Thread
  - with methods for synchronization
    - e.g., sleep(), suspend(), resume(), stop()
  - for creating new threads
    - e.g., start()
- JVM supports threads at run time
  - but what goes on underneath?
JVM and host OS Threads

- Java threads generally treated as user level threads
  - so JVM benefits from speed advantages of managing ULTs
- JVM spec doesn’t dictate mapping of Java threads to underlying OS kernel threads
  - WinNT supports as 1 Java thread : 1 kernel thread
  - Solaris started with many : 1, went to many : many

Class 3 Overview

- kernels
- processes: HWP, LWP, threads
- parallel processing overview
One CPU Is Not Enough

- the power of parallelism is undeniable and alluring
- many different approaches for introducing parallelism into computing:

Different Ways to Parallel

- SISD: single instruction, single data stream
  - 1 CPU executes 1 instruction stream on 1 data stream in 1 memory
Different Ways to Parallel

- SISD: single instruction, single data stream
- SIMD: single instruction, multiple data stream
  - 1 machine instruction controls execution on multiple CPUs in lock-step
  - each CPU has associated data memory, so the instruction is executed on different data
  - e.g., vector and array processors

Different Ways to Parallel

- SISD: single instruction, single data stream
- SIMD: single instruction, multiple data stream
- MISD: multiple instruction, single data stream
  - 1 data stream sent to multiple CPUs, each executing a (possibly) different instruction on that datum
  - not implemented (as of 2001)
Different Ways to Parallel

- SISD: single instruction, single data stream
- SIMD: single instruction, multiple data stream
- MISD: multiple instruction, single data stream
- MIMD: multiple instruction, multiple data stream
  - multiple CPUs simultaneously executing (possibly) different instruction sequences on different (possibly) data sets
  - if each CPU has its own dedicated memory ⇒ each processing element is separate computer ⇒ cluster
  - if CPUs share memory ⇒ shared memory multiprocessor

The Parallel Computing Family Tree

parallel computing

SIMD
  - shared memory (tightly coupled)
    - master/slave
    - symmetric multiprocessors

MIMD
  - distributed memory (loosely coupled)
    - clusters
Shared Memory Multiprocessors

- 2 kinds, depending on duties CPUs can take on
- master/slave or asymmetric:
  - all CPUs can run user mode instructions
  - OS kernel runs only on 1 of the CPUs
  - conflict resolution? only 1 master
  - simplified design
    - easy to extend from single CPU OS
- disadvantages?

- failure of master CPU wipes out whole system
- master becomes performance bottleneck
Symmetric Multiprocessors

- 2 kinds, depending on duties CPUs can take on
  - master/slave or asymmetric
- symmetric:
  - any CPU can perform user or kernel mode instructions, i.e., kernel can run on any CPU
  - CPUs do self-scheduling from pool of avbl. procs
  - kernel may consist of multiple threads so that parts can run concurrently
  - OS much more complex: synchronizing access to shared resources difficult
SMP OS issues

- OS should hide complexity of multiple CPUs, so users can take advantage of multi CPUs without needing to be aware of 1 or > 1
- major issues:
  - concurrency:
    - multiple CPUs may be simultaneously executing the same or different parts of the kernel ⇒ access to kernel data structures must be carefully synchronized
- major issues:
  - concurrency
  - scheduling:
    - any CPU can schedule processes or threads ⇒ ensure no conflicts
    - if kernel-level multithreading used, then may schedule multiple threads of task on different CPUs concurrently
SMP OS issues

- major issues:
  - concurrency
  - scheduling
  - synchronization:
    - to control access to shared resources (devices, memory)
    - to enforce one-at-a-time access where needed

- memory management:
  - handling multiported memory to provide efficient but safe access
  - special care to shared memory chunks (segments, pages)
SMP OS issues

- major issues:
  - concurrency
  - scheduling
  - synchronization
  - memory management
  - reliability, fault tolerance:
    - individual CPU failures should have little impact
    - kernel structures have to re-organize ‘around’ points of failure

For assignment submissions

- use UNIX tar command to produce single file for upload
- see info on course website, Library section, on using tar