Operating Systems

05. Threads

Paul Krzyzanowski

Rutgers University

Spring 2015

© 2014-2015 Paul Krzyzanowski

Thread of execution

Single sequence of instructions

- Pointed to by the program counter (PC)
- Executed by the processor

Conventional programming model & OS structure:

- Single threaded
- One process = one thread

© 2014-2015 Paul Krzyzanowski

Multi-threaded model

A thread is a subset of a process:

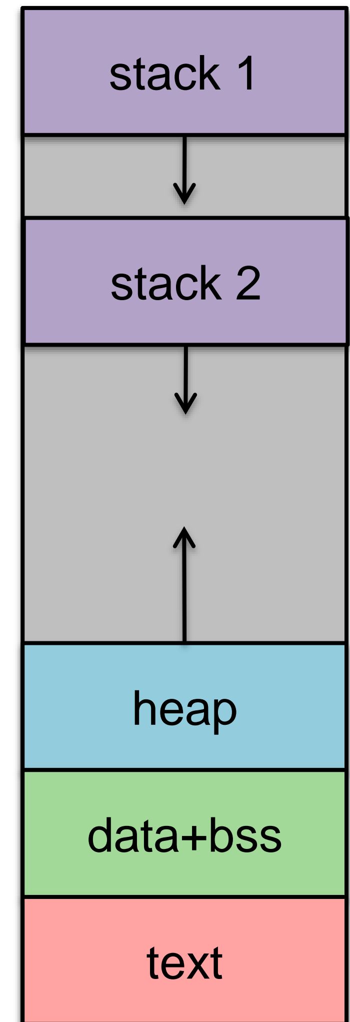
– A process contains one or more kernel threads

Share memory and open files

– BUT:
  separate program counter, registers, and stack
– Shared memory includes the heap and global/static data
– No memory protection among the threads

Preemptive multitasking:

– Operating system preempts & schedules threads



 © 2014-2015 Paul Krzyzanowski

Sharing

Threads share:

- Text segment (instructions)
- Data segment (static and global data)
- BSS segment (uninitialized data)
- Open file descriptors
- Signals
- Current working directory
- User and group IDs

Threads do not share:

- Thread ID
- Saved registers, stack pointer, instruction pointer
- Stack (local variables, temporary variables, return addresses)
- Signal mask
- Priority (scheduling information)

© 2014-2015 Paul Krzyzanowski

Why is this good?

Threads are more efficient

– Much less overhead to create: no need to create new copy of memory space, file descriptors, etc.

Sharing memory is easy (automatic)

– No need to figure out inter-process communication mechanisms

Take advantage of multiple CPUs – just like processes

– Program scales with increasing # of CPUs

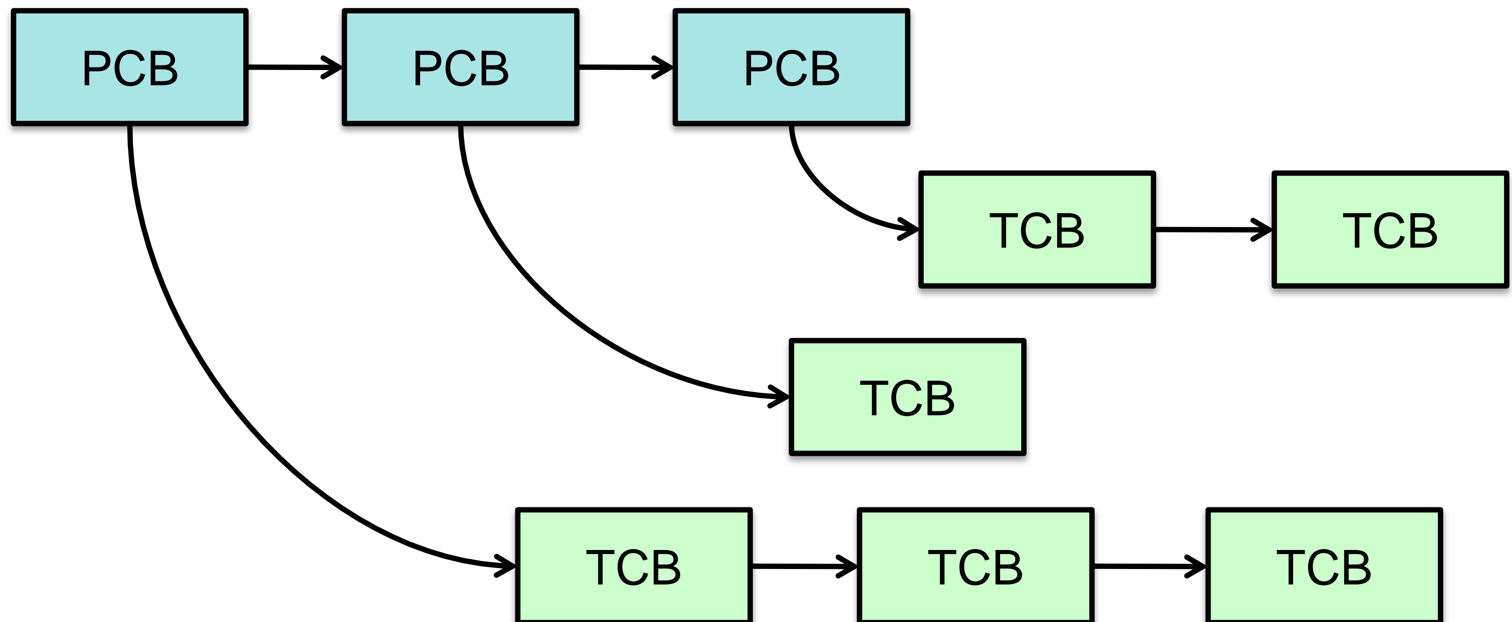
– Take advantage of multiple cores

© 2014-2015 Paul Krzyzanowski

Implementation

Process info (Process Control Block) contains one or more Thread Control Blocks (TCB):

- Thread ID
- Saved registers
- Other per-thread info (signal mask, scheduling parameters)



© 2014-2015 Paul Krzyzanowski

Scheduling

A thread-aware operating system scheduler schedules *threads*, not *processes*

- A process is just a container for one or more threads

© 2014-2015 Paul Krzyzanowski

Scheduling Challenges

Scheduler has to realize:

- Context switch among threads of different processes is more expensive
  - Flush cache memory (or have memory with process tags)
  - Flush virtual memory TLB (or have tagged TLB)
  - Replace page table pointer in memory management unit
- CPU Affinity
  - Rescheduling threads onto a different CPU is more expensive
  - The CPU's cache may have memory used by the thread cached
  - *Try to reschedule the thread onto the same processor on which it last ran*

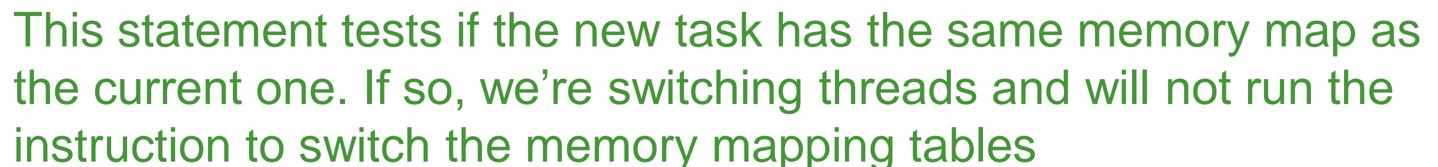
© 2014-2015 Paul Krzyzanowski

Process vs. Thread context switch

A thread switch within the same process is not a full context switch – the address space (memory map) does not get switched

```
linux/arch/i386/kernel/process.c:

/* Re-load page tables for a new address space */
{
    unsigned long new_cr3 = next->tss.cr3;
    if (new_cr3 != prev->tss.cr3)
        asm volatile("movl %0,%%cr3": :"r" (new_cr3));
}
```



© 2014-2015 Paul Krzyzanowski

Multi-threaded programming patterns

Single task thread

- Do a specific job and then release the thread

Worker threads

- Specific task for each worker thread
- Dispatch task to the thread that handles it

Thread pools

- Create a pool of threads *a priori*
- Use an existing thread to perform a task; wait if no threads available
- Common model for servers

© 2014-2015 Paul Krzyzanowski

Kernel-level threads vs. User-level threads

Kernel-level

- Threads supported by operating system
- OS handles scheduling, creation, synchronization

User-level

- Library with code for creation, termination, scheduling
- Kernel sees one execution context: one process
- May or may not be preemptive

© 2014-2015 Paul Krzyzanowski

User-level threads

Advantages

- Low-cost: user level operations that do not require switching to the kernel
- Scheduling algorithms can be replaced easily & custom to app
- Greater portability

Disadvantages

- If a thread is blocked, all threads for the process are blocked
  - Every system call needs an asynchronous counterpart
- Cannot take advantage of multiprocessing

© 2014-2015 Paul Krzyzanowski

You can have both

User-level thread library on top of multiple kernel threads

1:1 – kernel threads only
(1 user thread = 1 kernel thread

N:1 – user threads only
(*N* user threads on *1* kernel thread/process)

N:M – hybrid threading
(*N* user threads on *M* kernel threads)

© 2014-2015 Paul Krzyzanowski

pthreads: POSIX Threads

- POSIX.1c, Threads extensions (IEEE Std 1003.1c-1995)
  - Defines API for managing threads
- Linux: native POSIX Thread Library
- Also on Solaris, Mac OS X, NetBSD, FreeBSD
- API library on top of Win32

© 2014-2015 Paul Krzyzanowski

Using POSIX Threads

Create a thread

```
pthread_t t;
pthread_create(&t, NULL, func, arg)
```

- Create new thread *t*
- Start executing function *func(arg)*

Join two threads (wait for thread *t* to terminate)

```
void *ret_val;
pthread_join(t, &ret_val);
```

- Thread *t* exits via *return* or *pthread_exit*

No parent/child relationship among threads!

- Any one thread may wait (join) on another thread

A different approach to threads

- Threads force a highly-shared model
  - Memory map, signals, open files, current directory, etc. all shared
- Processes force a non-shared model
  - Separate memory, open files, etc.
- What if we allow the user to specify what is shared when a new process is created?
  - Then we don't need threads since processes can share all memory if they want to … and open files … and anything else

© 2014-2015 Paul Krzyzanowski

Linux threads

- Linux has no concept of a thread
- All threads implemented as standard processes
  - No special scheduling semantics
  - All processes defined in the kernel by `task_struct`
- Support thread-like behavior via *clone* system call
  - Designed to implement threads
  - A process can control what gets shared with a new process
  - Based on Plan 9's *rfork* system call

© 2014-2015 Paul Krzyzanowski

Linux *clone()* system call

- Clone a process, like *fork,* but:
  - Specify function that the child will run (with argument)
    - Child terminates when the function returns
  - Specify location of the stack for the child
  - Specify what's shared:
    - Share memory (otherwise memory writes use new memory)
    - Share open file descriptor table
    - Share the same parent
    - Share root directory, current directory, and permissions mask
    - Share namespace (mount points creating a directory hierarchy)
    - Share signals
    - *And more…*
- Used by pthreads

© 2014-2015 Paul Krzyzanowski

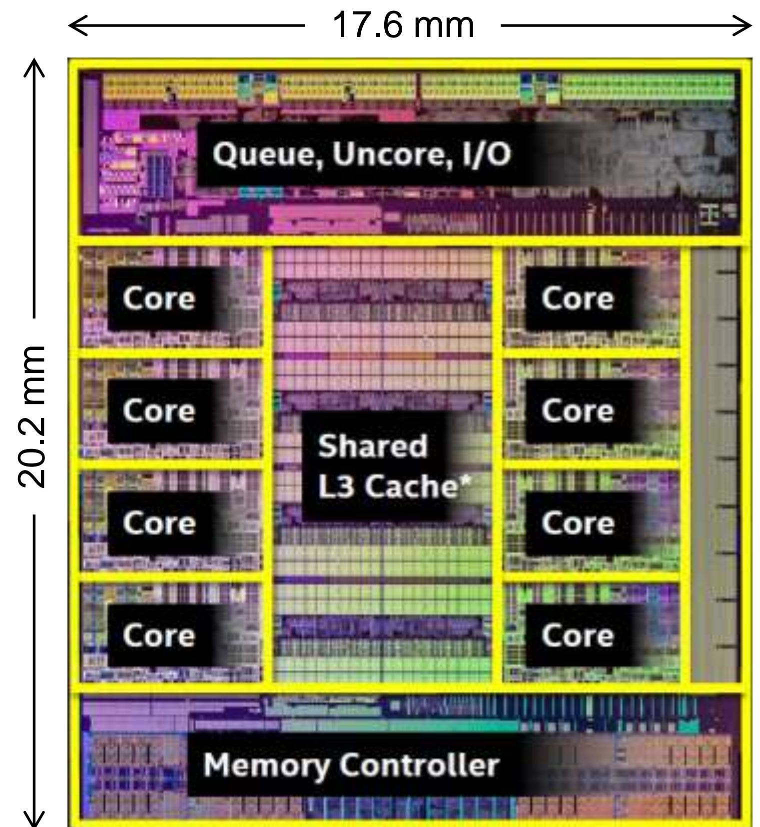
Threading in hardware

- Hyper-Threading (HT) vs. Multi-core vs. Multi-processor
- One core = One CPU
- Hyper-Threading
  - One physical core *appears* to have multiple processors
    - Looks like multiple CPUs to the OS
    - Separate registers & execution state
  - Multiple threads run but compete for execution unit
  - Events in the pipeline switch between the streams
  - Threads do not have to belong to the same process
    - But the processors share the same cache
    - Performance can degrade if two threads compete for the cache
  - Works well with instruction streams that have large memory latencies

© 2014-2015 Paul Krzyzanowski

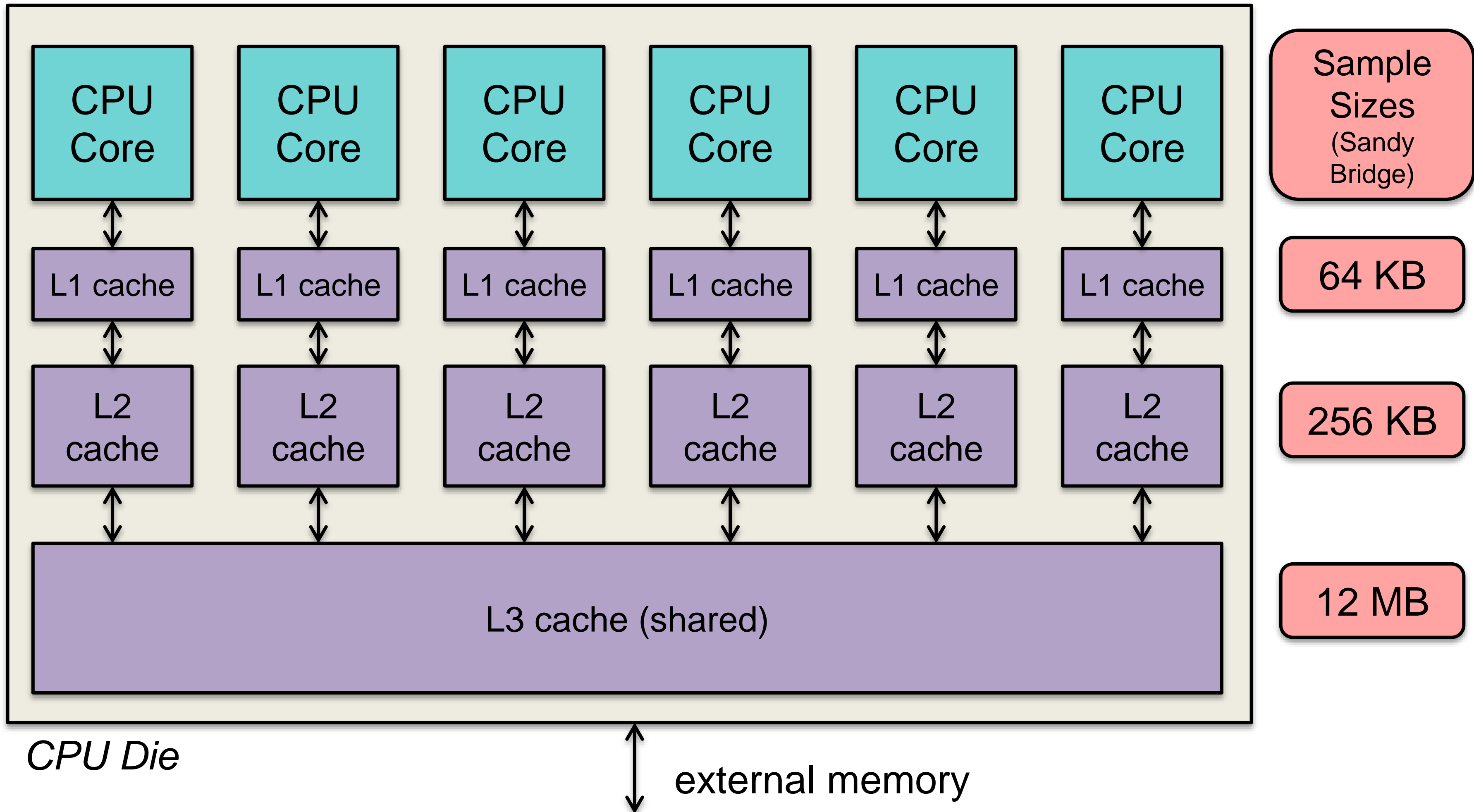
Example CPU

- Intel® Core™ i7-5960X Extreme Edition (Haswell-E)
- 3.0 GHz up to 3.5 GHz (Turbo)
- 2.6B 22 nm Tri-Gate 3-D transistors
- 8 cores; 16 threads
- Per-Core caches:
  - 512 KB L1 cache (128 KB data; 128 KB instruction)
  - 2 MB L2 cache
- 20 MB shared L3 cache



© 2014-2015 Paul Krzyzanowski

Multi-core architecture



© 2014-2015 Paul Krzyzanowski

Stepping on each other

- Threads share the same data
- Mutual exclusion is critical
- Allow a thread be the only one to grab a critical section
  - Others who want it go to sleep

```
pthread_mutex_t = m = PTHREAD_MUTEX_INITIALIZER;
    ...
pthread_mutex_lock(&m);
/* modify shared data */
pthread_mutex_unlock(&m);
```


© 2014-2015 Paul Krzyzanowski

The End

© 2014-2015 Paul Krzyzanowski