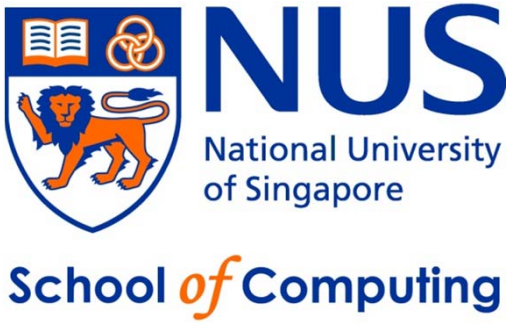
CG2271

Real-Time Operating Systems

Lecture 8

Non-RTOS Issues I

colintan@nus.edu.sg



NUS National University of Singapore
School of Computing

Learning Objectives

- **By the end of this lecture you will be able to:**
  - Understand what a general operating system is, and how they are different from RTOS.
  - Understand threads, virtual memory and file systems.

Introduction

- **RTOS are extremely specialized and highly optimized operating systems for real-time systems that have:**
  - Very tight timing requirements.
  - Very tight space requirements.
  - Small to moderate processing power.
- **For this reason RTOS lack many capabilities:**
  - Supporting multiple “sub-tasks” within a task – threads.
  - Supporting more data and programs than memory allows – virtual memory.
  - Supporting data stored on disks – file systems.

Introduction

- **However knowledge of these issues are important to you as computer engineers.**
  - Sadly you won't work solely with RTOS in your life.
  - You also have to face operating systems like Mac OS X, and very unfortunately, Microsoft Windows.
- **This lecture is designed to give you a quick overview of these issues.**

Non-RTOS Issues

THREADS

Threads

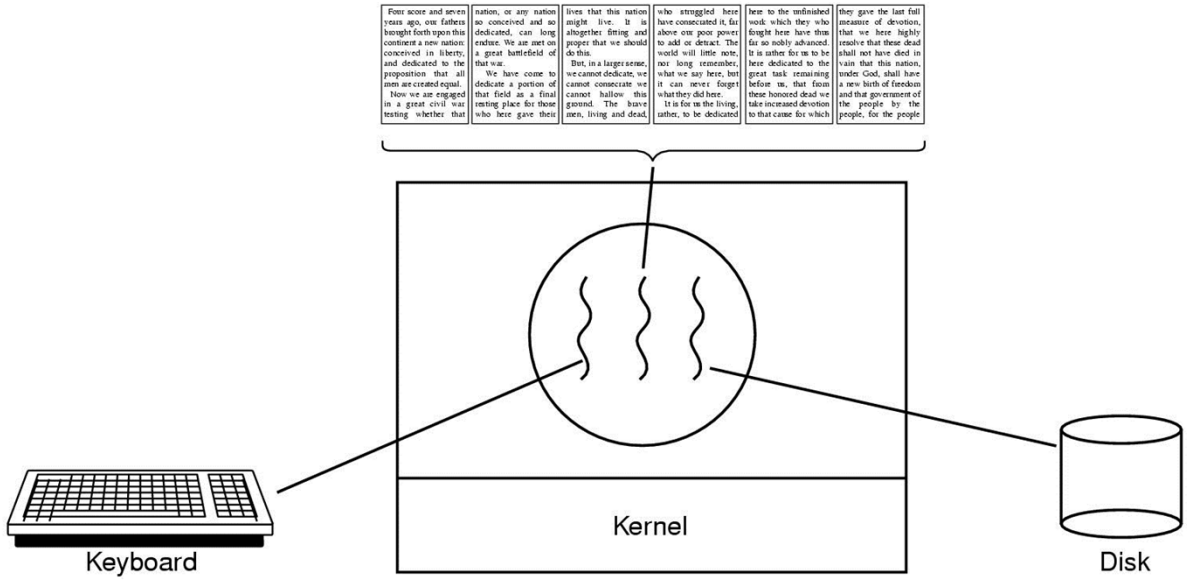
- **Threads are like processes:**
  - They're independent sequence of instructions, with their own program counter and register contents.
  - Their main purpose is to do multiple things "at the same time".
- **However threads are different from processes:**
  - Threads are "mini-processes" within a process.
  - Unlike processes, threads share the same memory space.

Threads

- **Reasons for having threads:**
  - Shared memory space means it becomes easier to share data.
  - They are less complicated (lighter) than processes, and creating/destroying a thread is 10-100x faster than for a process. This is useful in cases where an application needs to spawn/destroy independent “sub-processes” rapidly.
  - If some threads are I/O-bound, having multiple-threads effectively allows a single process to accomplish several activities at the same time.
  - Threads are useful in multiple-core or multiple-CPU systems, giving true parallelism.

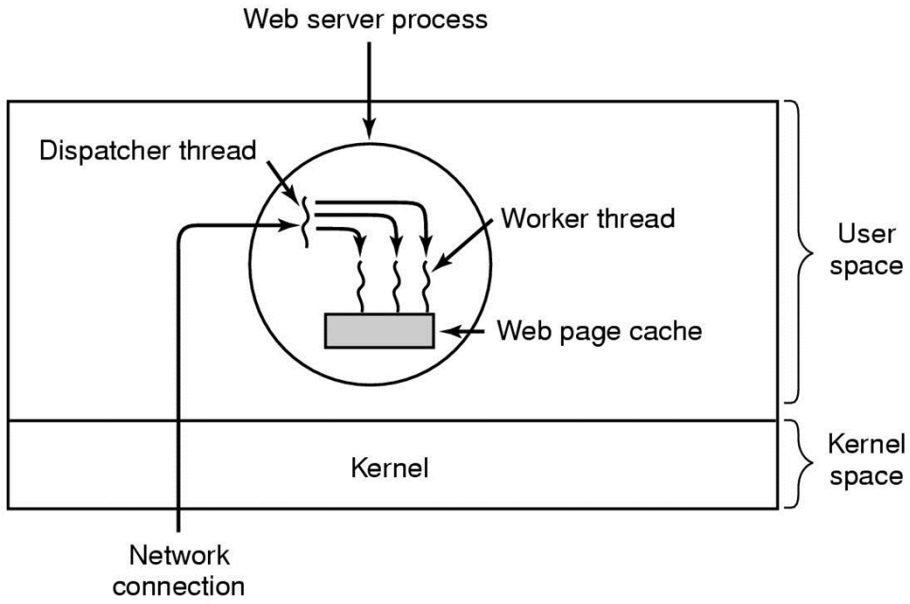
Threads

- **Example thread applications:**
  - Word processor
    - ✓**One thread to deal with disk backups.**
    - ✓**One thread to handle document reformatting.**
    - ✓**One thread to interact with the user.**



Threads

- Web Server. When a new connection comes in:
  - ✓**A dispatcher thread wakes up an idle "worker thread".**
  - ✓**Worker thread checks to see if the page is in cache. If yes it returns the page, if not, it initiates disk I/O and blocks. The dispatcher then chooses another thread to deal with more work.**



Threads

- **Rough outline of the dispatcher thread (left) and worker thread (right).**

```
while (TRUE) {
    get_next_request(&buf);
    handoff_work(&buf);
}
```

(a)

```
while (TRUE) {
    wait_for_work(&buf)
    look_for_page_in_cache(&buf, &page);
    if (page_not_in_cache(&page))
        read_page_from_disk(&buf, &page);
    return_page(&page);
}
```

(b)

Threads

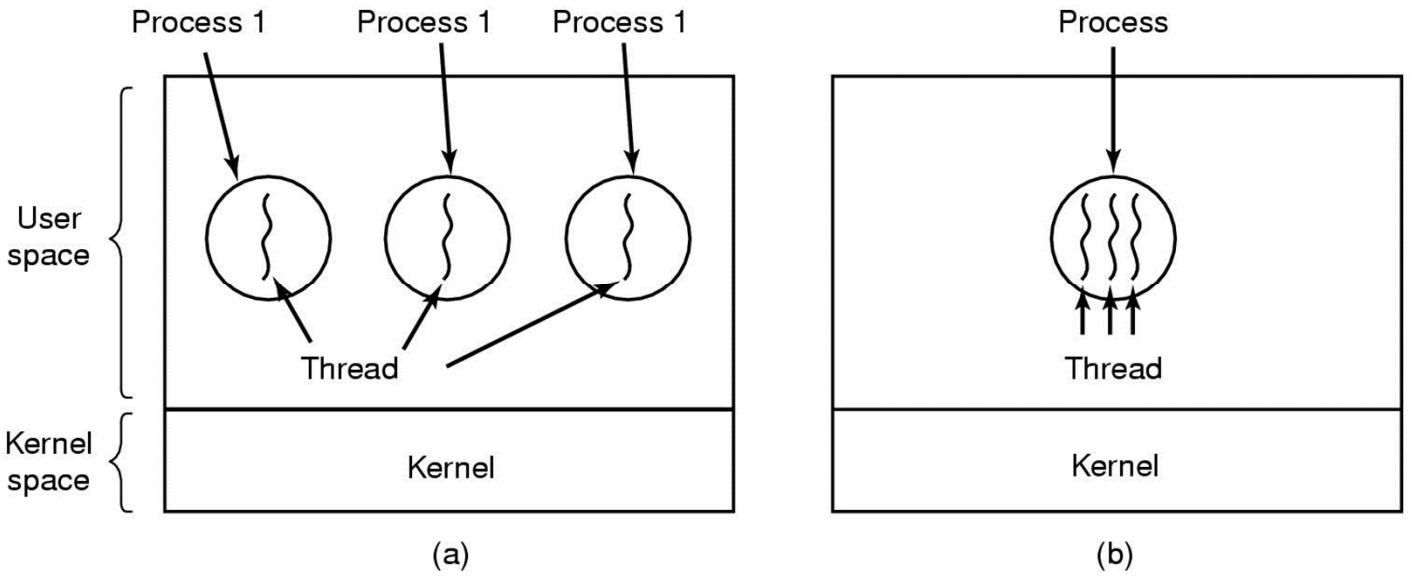
- **Reasons for NOT having threads:**
  - The OS must now manage multiple PCs and register sets for each process, instead of just one set. More overheads!!
  - One advantage of threads is the ability to quickly create and destroy them when needed.
    - ✓**This however can make task execution times unpredictable.**
- **Real-time tasks are very similar to threads, except that:**
  - You cannot rapidly create and destroy tasks arbitrarily.
  - Tasks may not necessarily share memory.
  - You do not manage multiple sets of context data for a single task.

The Classical Thread Model

- **One way of looking at processes:**
  - A grouping of related resources.
    - ✓**Address space containing program instructions and data.**
    - ✓**Open files**
    - ✓**Child processes**
    - ✓**Accounting information, etc.**
- **Another way of looking at processes**
  - Consists of a single thread of execution.
    - ✓**With it's own program counter, and registers for storing data to be operated on, and results of operations.**
    - ✓**It's own stack which contains execution history.**

The Classical Thread Model

- **What the thread model introduces is:**
  - The ability to have multiple threads of execution running within a shared resources space of the process. So each thread has:
    - ✓**It's own program counter, registers and stack.**
    - ✓**But shares open files and data with other threads in the same process.**



The Classical Thread Model

- **On a single CPU, a scheduler picks threads to run according to a scheduling algorithm.**
  - Each thread takes turns, to give the illusion of parallel execution.
- **Threads share address space. This means that all threads can access global variables within the process.**
- **There is no memory protection because:**
  - There's no hardware provisions in the CPU to do this.
  - There's no need to.
    - ✓**Processes may come from different users and you need to protect one user from another possibly hostile user.**
    - ✓**Threads come from the same user, who presumably would not harm his own threads.**

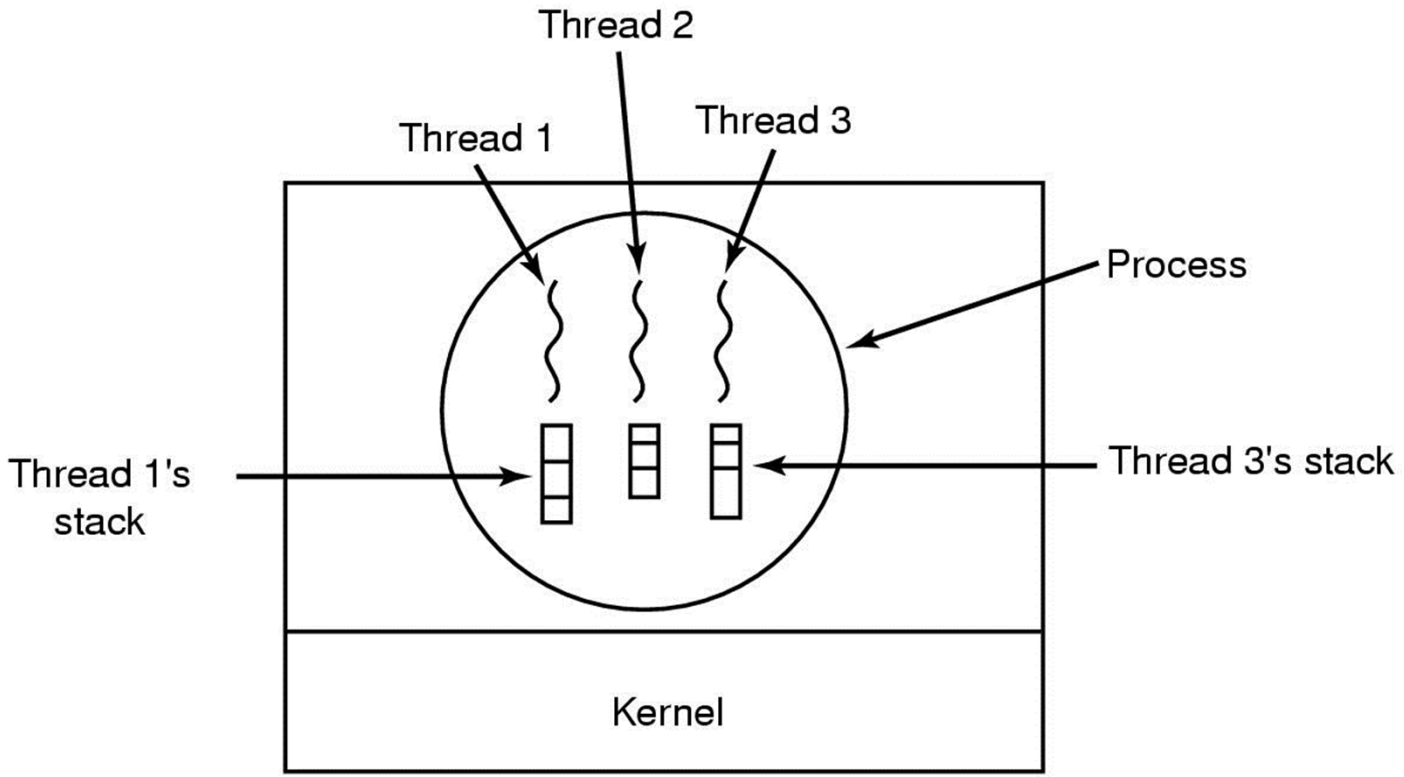
The Classical Thread Model

- **The column on the left shows resources that are shared across threads, and the right shows resources private to each thread:**

Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

The Classical Thread Model

- **Each thread is treated like a separate process and can be either in a RUNNING, READY or BLOCKED state.**
- **Furthermore each thread has its own context and stack.**



Thread Example

- **Example using POSIX threads.**

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUMBER_OF_THREADS 10

void *print_hello_world(void *tid)
{
     /* This function prints the thread's identifier then exits. */
     printf("Hello world. Greetings from thread %d", tid);
     pthread_exit(NULL);
}
int main(int argc, char **argv)
{
    /* Create 10 threads and then exits */
    pthread_t threads[NUMBER_OF_THREADS];
    int status, i;

    for(i=0; i<NUMBER_OF_THREADS; i++)
    {
            printf("Main here. Creating thread %d\n", i);
            status=pthread_create(&threads[i], NULL, print_hello_word, (void *) i);

            if(status)
            {
                    printf("Oops! thread_create returned error code %d\n", status)
                    return -1;
            }

    }
    return 0;
}
```

Thread Implementations

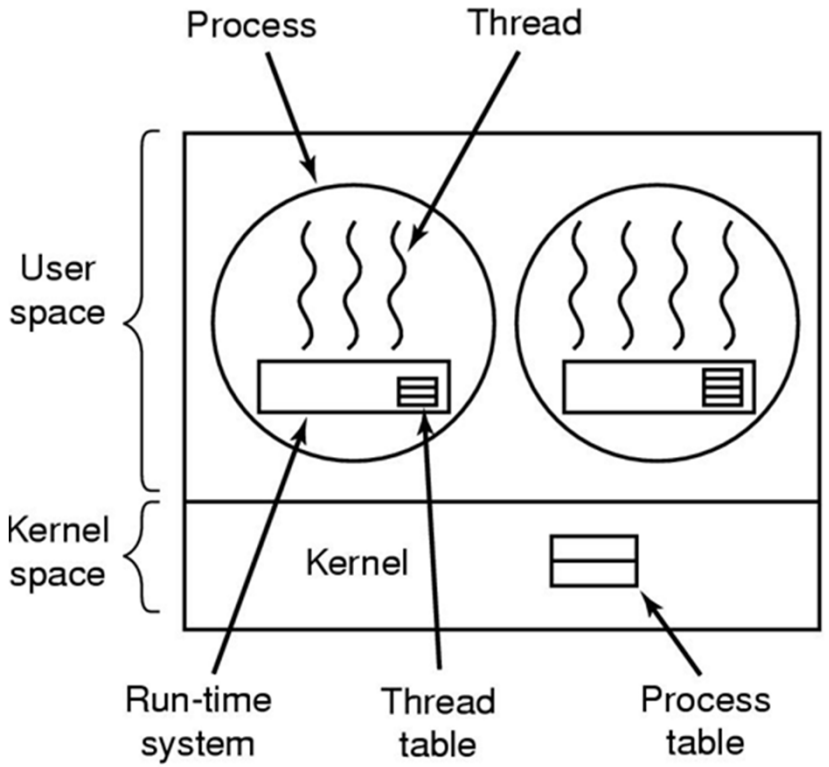
- **Schedulable Entities:**
  - These are “objects” that the OS scheduler is aware of.
- **We now look at several ways to implement threads:**
  - User Level (N:1) Threads
    - ✓**N threads per schedulable entity**
    - ✓**Each schedulable entity is a process.**
  - Kernel Level (1:1) Threads
    - ✓**1 thread per schedulable entity.**
    - ✓**Each schedulable entity is a thread.**
  - Hybrid (M:N) Threads
    - ✓**M threads per N schedulable entities.**
    - ✓**Each schedulable entity is a thread.**

Thread Implementations
User-Level

- **There are two major ways of implementing threads:**
  - User level threads.
    - ✓**These are threads that are implemented completely in the user space, with the kernel not knowing anything about the threads.**
    - ✓**This is done through a user thread library.**
    - ✓**Very useful for operating systems that don't support threads.**
  - Kernel level threads.
    - ✓**These are threads that are implemented in the kernel space.**
    - ✓**The operating system provides a thread management interface and manages all thread accounting.**

Thread Implementations User-Level

- **User Level Thread**
  - Each process maintains its own thread table.
  - Similar to a process table but only contains thread data like PC, stack pointer, registers, etc.
- **Thread tables are managed by a thread runtime.**
  - The runtime also manages thread states like BLOCKED, RUNNING and READY.



Thread Implementations
User-Level

- **Advantages of User Level Threads**
  - Ability to implement threading even in OS that has no thread support.
  - Thread management is done within the process, and there is minimal need to involve OS calls.
    - ✓**If the thread context can be managed with just a few instructions to copy all the registers, etc, this is potentially very fast.**
  - More scalable as each process' copy of the thread runtime manages the threads in the process.
    - ✓**Contrast this with a kernel approach where the kernel has to manage threads across many tasks.**

Thread Implementations
User-Level

- **Disadvantages of User Level Threads**
  - Implementation of blocking calls is tricky.
    - ✓**E.g. if a thread makes a call to read the keyboard before a key is hit, the call might "block".**
    - ✓**When this happens, this blocked thread will prevent all other threads from running since it can't yield execution.**
    - ✓**Solutions:**

      i. Rewrite calls in the OS to be non-blocking. - Not ideal, often not possible.

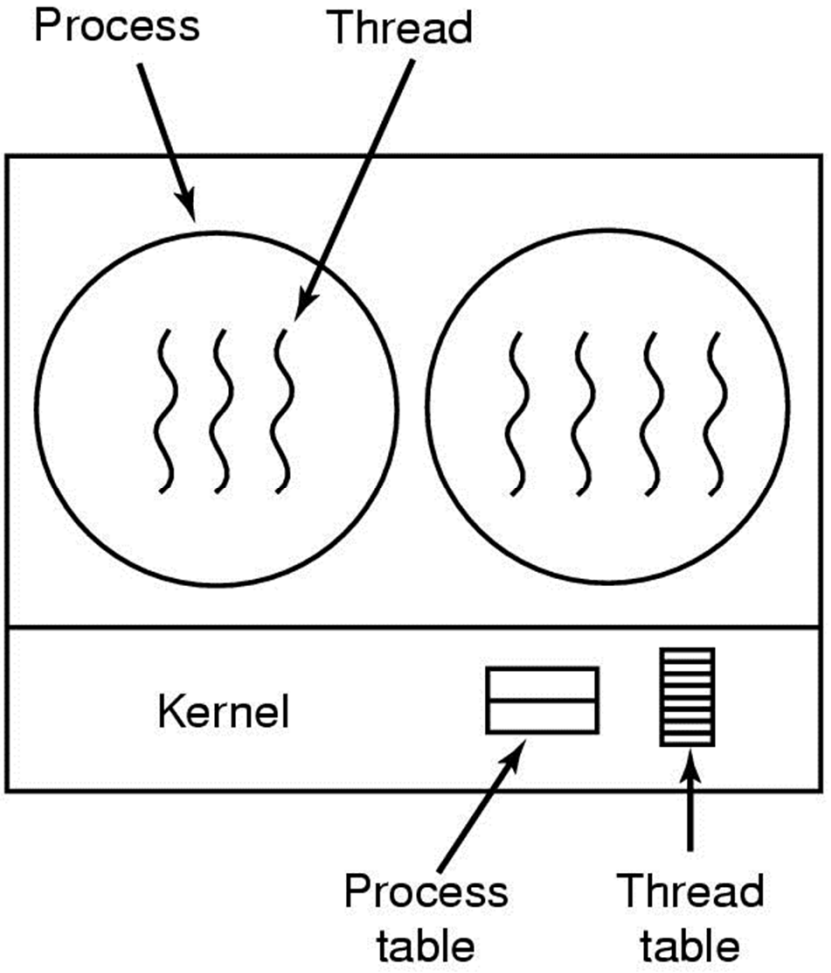
      ii. Test whether a call will block. If so, select another thread for running. When the thread runtime gets control again, repeat the test and restart the thread if the call will not block. – Inelegant.

Thread Implementations
User-Level

- **Disadvantages of User Level Threads**
  - A thread must voluntarily yield CPU control, since there are no “clock ticks” or other means within a process to know force the thread to yield control.
- **Threads are often used in applications that make frequent system calls.**
  - E.g. in a web server, threads will access the disk often.
  - This negates the primary advantage of user-level threads: that they don’t involve the OS and are potentially very fast.

Thread Implementations Kernel-Level

- **Kernel-Level Threads**
  - Thread management is handled by the kernel.
    - ✓**A single thread table is held within the kernel instead of in the memory of each process.**
      - *This is maintained as part of the information of a process.*
    - ✓**No need for thread runtime running in user-space to manage the threads.**



Thread Implementations
Kernel-Level

✓If a thread makes a blocking call, easy for OS to find another thread to run.

- Disadvantages:

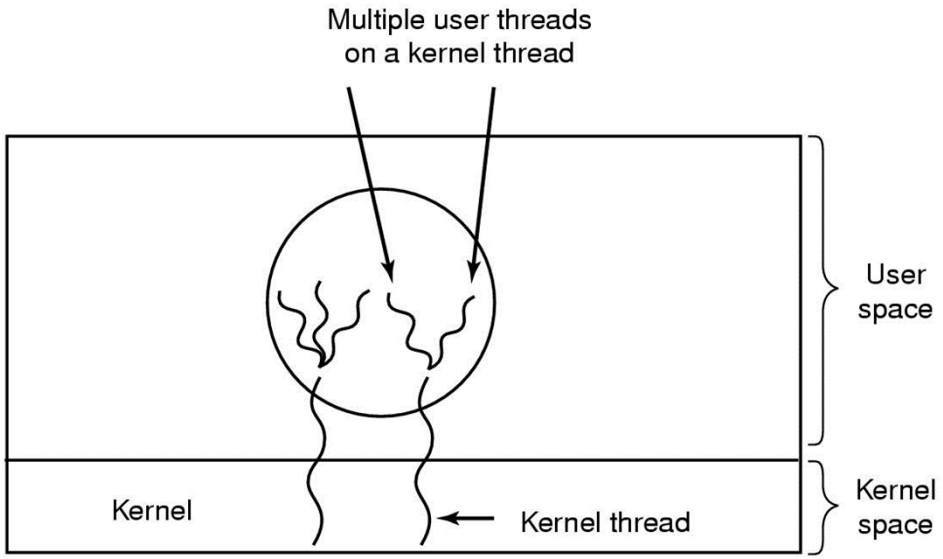
✓System calls to create and schedule threads require switch to kernel mode. This is very slow.

✓Typically requires 1,500 cycles to do so.

✓On a modern microprocessor that can execute 4 to 5 instructions per cycle, this is equivalent to the execution time of 6,000 instructions.

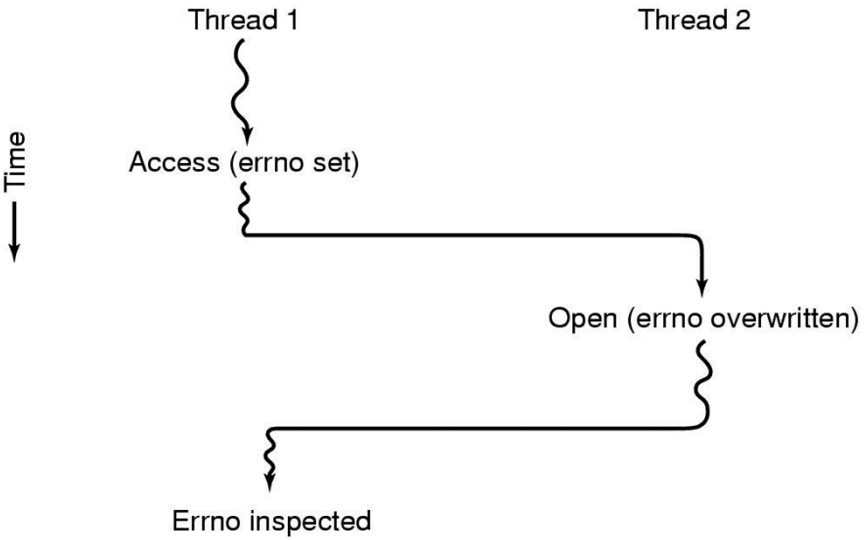
Thread Implementations Hybrid Threads

- **In the hybrid approach, a small number of kernel threads are created.**
  - Several user-level threads then run on top of each kernel thread.
    - ✓**A user-level runtime then shares the kernel thread with each user thread.**
    - ✓**User-level threads can be created , scheduled and destroyed quickly.**
  - Kernel is aware only of the kernel threads and schedules these.



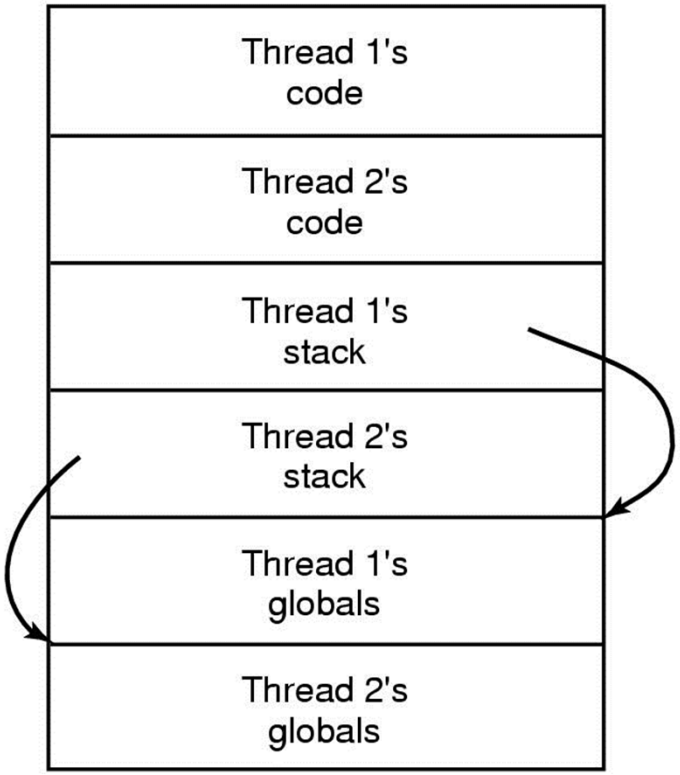
Multithreading Single Thread Codes

- **Global variables used in single-threaded codes can become an issue when multithreading:**
  - "errno" is a system variable to indicate success/failure in a system call.
    - ✓**Thread 1 makes a system call that fails. Failure code is written to errno.**
    - ✓**Thread 1 is pre-empted and thread 2 runs, and it makes a successful system call, causing errno to be "cleared".**
    - ✓**Thread 1 resumes, reads errno and assumes his own system call succeeded.**



Multithreading Single Thread Codes

- **Possible solution:**
  - Let each thread have its own copy of global variables.
    - ✓**If a thread has multiple sub-functions, each sub-function can access these globals.**
    - ✓**However other threads in the process cannot.**
  - Can be tricky to implement since most programming languages either have:
    - ✓**Globals that are accessible across the entire process.**
    - ✓**Locals that are accessible only within a function.**



Multithreading Single Thread Codes

- **A second issue is "library re-entrancy".**
  - Similar in nature to our "errno" problem but more insidious because it involves library codes written by others.
  - Consider a library function "add1" that adds 1 to a global variable x. In assembly this would look like:

```
PUSH R0              ; Save R0 and R1 onto the stack
PUSH R1              ;
LI R0, address(x)    ; Put address of variable X into R0
LW R1, (R0)          ; Load the contents of the address in R0 into R1
ADDI R1, R1, 1       ; Add 1 to R1
SW R1, (R0)          ; And write the result back.
POP R1               ; Restore R1
POP R0               ; Restore R0
```

Multithreading Single Thread Codes

- **Single-threaded version. Initially R0=R1=0, x=2. Stack grows from right to left.**

Thread 1	Thread 1 Registers	Thread 2	Thread 2 Registers	Stack	x
push r0	r0=0, r1=0			0	2
push r1	r0=0, r1=0			0,0	2
li r0, address(x)	r0=&x,r1=0			0,0	2
lw r1, (r0)	r0=&x, r1=2			0,0	2
addi r1, r1, 1	r0=&x, r1=3			0,0	2
sw r1, (r0)	r0=&x, r1=3			0,0	3
pop r1	r0=&x, r1=0			0	3
pop r0	r0=0, r1=0				3
		push r0	r0=0, r1=0	0	3
		push r1	r0=0, r1=0	0,0	3
		li r0, address(x)	r0=&x, r1=0	0,0	3
		lw r1, (r0)	r0=&x,r1=3	0,0	3
		addi, r1, r1, 1	r0=&x,r1=4	0,0	3
		sw r1, (r0)	r0=&x,r1=4	0,0	4
		pop r1	r0=&x,r1=0	0	4
		pop r0	r0=0,r1=0		4

Multithreading Single Thread Codes

- **Multi-threaded version:**
  - Initially thread 1 calls add1
  - After the LW instruction, thread 1 gets pre-empted, and thread 2 is started.
  - Thread 2 calls add1.
- **Outcome is shown on the next slide.**

Multithreading Single Thread Codes

Thread 1	Thread 1 Registers	Thread 2	Thread 2 Registers	Stack	x
push r0	r0=0, r1=0			0	2
push r1	r0=0, r1=0			0,0	2
li r0, address(x)	r0=&x,r1=0			0,0	2
lw r1, (r0)	r0=&x, r1=2			0,0	
		push r0	r0=&x, r1=2	&x,0,0	2
		push r1	r0=&x, r1=2	2, &x, 0, 0	2
		li r0, address(x)	r0=&x, r1=2	2, &x, 0, 0	2
		lw r1, (r0)	r0=&x, r1=2	2, &x, 0, 0	2
		addi, r1, r1, 1	r0=&x, r1=3	2, &x, 0, 0	2
		sw r1, (r0)	r0=&x, r1=3	2, &x, 0, 0	3
		pop r1	r0=&x, r1=2	&x, 0, 0	3
		pop r0	r0=&x, r1=2	0, 0	3
addi r1, r1, 1	r0=&x, r1=3			0, 0	3
sw r1, (r0)	r0=&x, r1=3			0, 0	3
pop r1	r0=&x, r1=0			0	3
pop r0	r0=0, r1=0				3

Multithreading Single Thread Codes

- **We can see that:**
  - In the single-threaded case, thread 1 updates x to 3, then thread 2 updates x to 4.
  - In the multi-threaded case, thread 1 reads x to be 2, gets pre-empted by thread 2.
  - Thread 2 reads x to be 2, updates it to 3.
  - Thread 1 is restarted with the old value of 2, updates it to 3, and writes the (incorrect) value to x.
    - ✓**Correct value is 4!**
- **Since thread pre-emption is unpredictable, such bugs are hard to reproduce.**

Multithreading Single Thread Codes

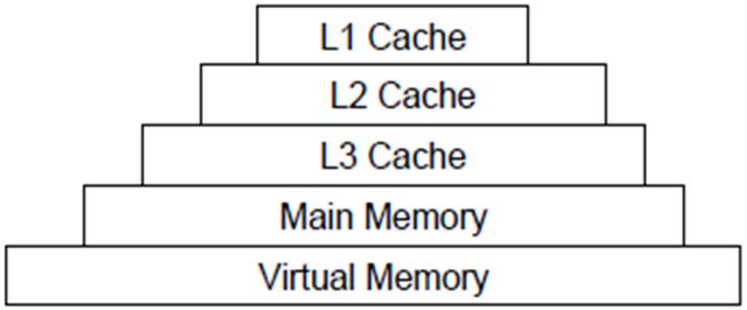
- **We say that add1 is "non re-entrant" because, in the situation that:**
  - ✓**One thread is in the middle of running add1 and gets pre-empted,**
  - ✓**And the second thread calls add1,**
  - Execution of add1 becomes incorrect!!
- **To make add1 re-entrant:**
  - We add a "wrapper" around add1 so that:
    - ✓**When thread1 calls add1 through the wrapper, the wrapper sets a "busy bit".**
    - ✓**When thread2 calls add1 again through the wrapper, the wrapper sees that the bit is busy thread2 suspends till thread1 is done.**

Non-RTOS Issues

VIRTUAL MEMORY

Memory Hierarchy

- **"Memory" in a desktop computer today is rarely a single monolithic structure.**
  - Consists of layers, called a "hierarchy".
    - ✓**Topmost layer is the fastest, but most expensive and therefore the smallest.**
    - ✓**Bottom-most layer is the cheapest and biggest, but the slowest.**
  - Each layer above contains a small portion of the layer below:
    - ✓**Use "replacement policies" to decide which portions to copy.**
- **When you buy a computer with "16GB of memory", it refers to size of the "main memory" layer.**



Memory Hierarchy

- **Due to the "principles of locality", over 98% of data and instructions needed by a program are in the topmost (and fastest) layer!**
  - Spatial Locality: If you've accessed a memory location, there's a very high chance that the next location you access is right next to it.
    - ✓**E.g. executing instructions sequentially, accessing elements of an array.**
  - Temporal Locality: If you've accessed a memory location, there's a very high chance that you will access it again.
    - ✓**E.g. Loops.**

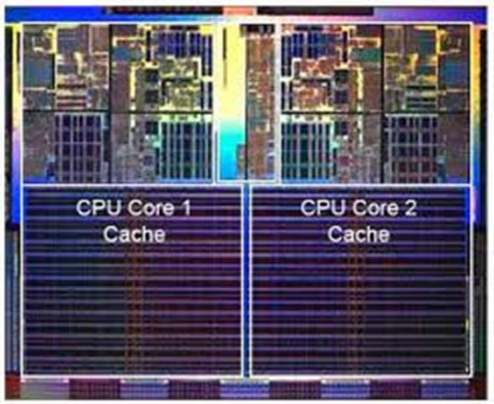
Memory Hierarchy

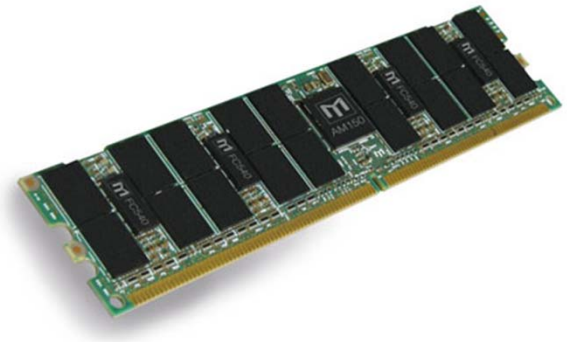
- **This is significant:**
  - The highest layers – cache – work in the 6-20 ns range.
  - The main memory layer works in the 80-100 ns range
  - The lowest layer works in the 50-60 ms range. Almost 1,000x slower than the highest layer!
- **Because of locality, memory hierarchy allows you to have:**
  - Very fast memory, since most come from the fast top layer.
  - Very large memory, since we can continue to keep what we don't (yet) need in the lowest layer.

NUS National University of Singapore
School of Computing

Memory Hierarchy

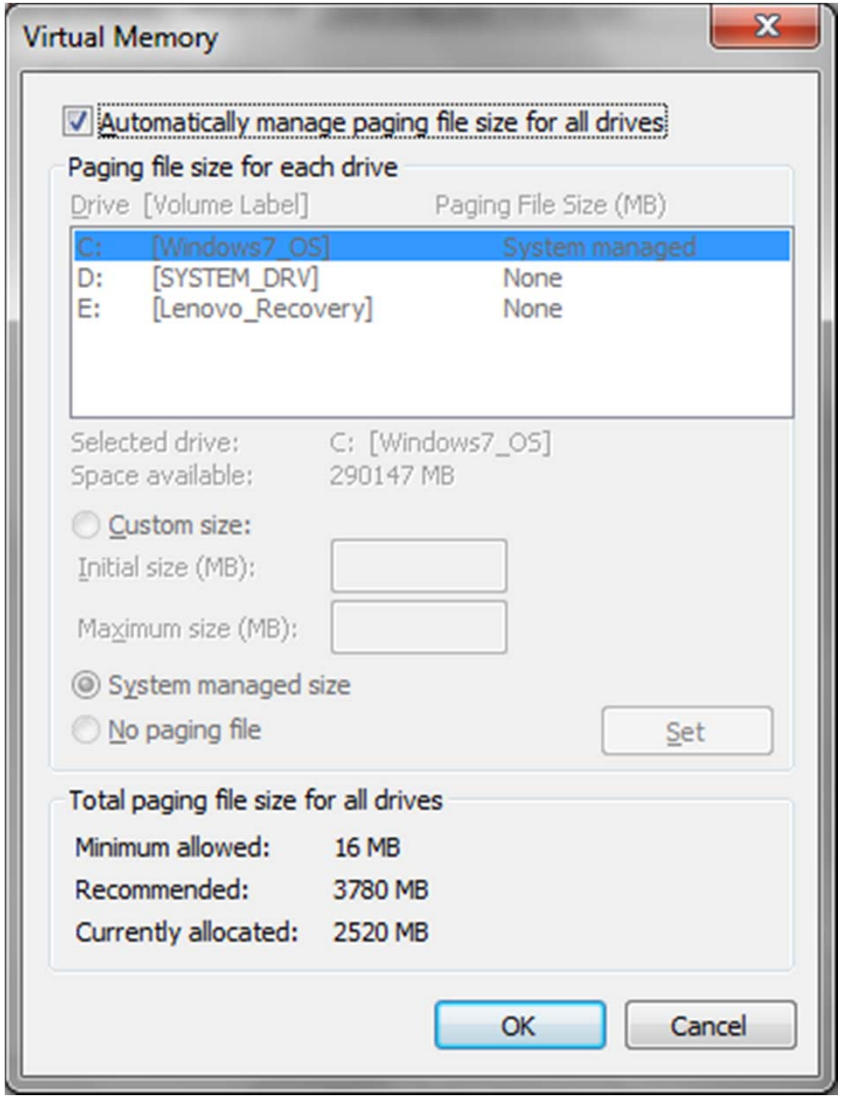
- **L1 cache – normally on the microprocessor chip die.**
- **L2 cache – Off the chip die, but within the chip casing.**
- **L3 cache – On the motherboard (rare).**
- **Main memory – That stuff you got snookered into buying a lot of.**
- **Virtual memory – The star of the show.**





Virtual Memory

- **Virtual memory is:**
  - The lowest layer, slowest layer of the memory hierarchy.
    - ✓**Actually, unlike the rest of memory, VM is not even implemented in silicon!**
    - ✓**It is implemented on your hard disk!**
  - The layer above (your "main memory") maintains a copy of a small portion of the VM.



Virtual Memory

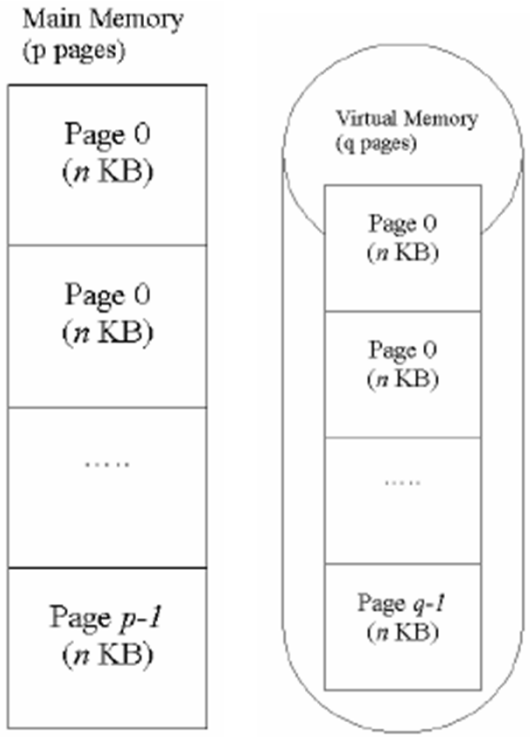
- **Primary motivation:**
  - Hard disk space is CHEAP!
  - Make a portion of the hard disk *look like* memory to the CPU, so that we can fool the CPU into thinking there's more memory that we actually paid for!
- **We can do this by having instructions and data that the CPU is currently interested in, in main memory. Everything else stays on the VM.**
- **In this way we can squeeze MUCH MORE instructions and data than otherwise possible!**

Virtual Memory

- **Problem:**
  - The CPU can only execute instructions and access data if it's in the main memory (actually, only if it's in the L1 cache, but we'll overlook that detail).
  - But main memory contains only a portion of the VM contents!
- **Questions:**
  1. How do we map the much larger addressing space of the VM to the smaller space of the main memory?
  2. What if the CPU tries to read instructions/data that are in VM but not in the main memory?

Virtual Memory

- **Paging.**
  - Both main memory and VM are divided into fixed size blocks called “pages”.
    - ✓**A “Virtual page” in the VM can be loaded to any “physical page” in main memory.**
- **To answer the first question:**
  - The CPU always generates “virtual addresses”.
    - ✓**I.e. any CPU address always points to a page in virtual memory.**



Virtual Memory

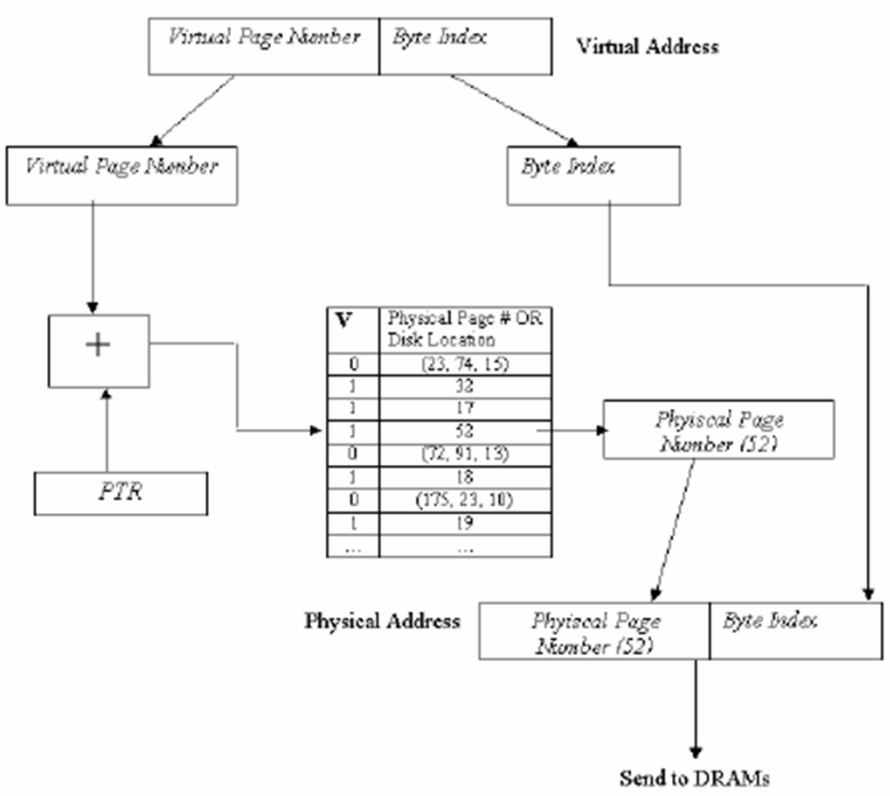
- Specialized hardware on the CPU, together with virtual memory services in the OS, work together to translate "virtual addresses" into "physical addresses" that correspond to locations in main memory.
  - ✓**This is achieved through a "page table".**
  - ✓**This is a data structure in main memory created and maintained by the OS.**
- V=1 means the information requested is in main memory.

V	Physical Page # OR Disk Location
0	(23, 74, 15)
1	32
1	17
1	52
0	(72, 91, 13)
1	18
0	(175, 23, 10)
1	19
…	…

NUS National University of Singapore
School of Computing

Virtual Memory Page Table

- **The virtual address forms an index into the page table.**
  - If the "virtual page" is in memory, a "memory translation" process takes place that locates which physical page this virtual page has been loaded into.
- **This information is used to generate the "physical address" to access main memory.**



Virtual Memory
Page Faults

- **This scheme works if the instruction/data requested is actually in main memory.**
- **If it's still on the disk:**
  - The V flag will be "0". The hardware sees this and generates a "page fault" interrupt.
  - This is vectored to a "page fault" ISR within the OS.

Virtual Memory
Page Faults

- **The page fault ISR does the following:**
  - Locates where the missing page is on the disk.
    - ✓**Easy: When V=0, the page table contains the location on disk where the page is (in cylinder/side/block format).**
  - Finds a free "physical page" to load the VM page into.
  - Updates the page table.
    - ✓**Set V=1, and changes the entry to show which physical page the virtual page has been loaded into.**
- **The VM then goes through the rest of the address translation process to allow the CPU to access the missed data/instruction.**

Virtual Memory
Page Faults

- **What if there are no more free physical pages?**
  - We must select data/instructions in one of the physical pages that is in used to be copied back to VM.
    - ✓**"Swapping" back to VM.**
  - The new virtual page is then loaded into the freshly vacated physical page.
- **How do we choose which page to "swap out"?**
  - Least Recently Used.
  - Least Frequently Used.

Virtual Memory Thrashing

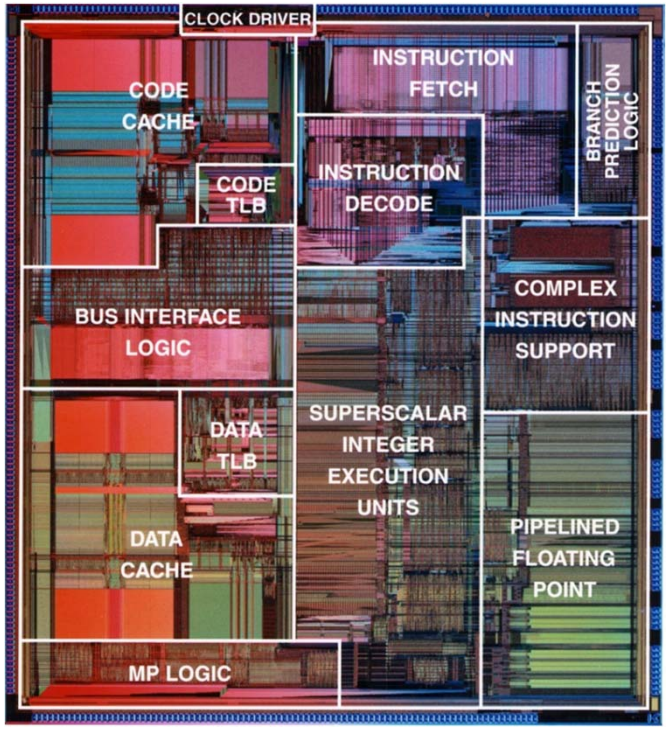
- **When the amount of data/instructions in VM is much, much greater than available physical memory:**
  - Accessing instructions/data might frequently be in pages that aren't yet in physical memory.
  - Shortage of physical memory causes pages to be frequently swapped out to disk.
  - The swapped out pages are accessed again, causing other pages to be swapped out so that these can be swapped back in.
- **This is called "thrashing" and is a performance disaster.**
  - Your disk light keeps flashing and your computer grinds down almost to a halt.

Virtual Memory Translation Lookaside

- **The page table is in main memory.**
  - This means TWO accesses to main memory for each CPU memory access:
    - ✓**One access to consult the page table.**
    - ✓**One access to actually read/write the physical memory.**
  - Remember that main memory is SLOW!
    - ✓**Typically 80ns vs 10ns of the L3 cache. L1 and L2 are even faster.**
  - Solution:
    - ✓**Use a special high speed memory to store parts of the page table!**

Virtual Memory Translation Lookaside

- **This special cache is called the "Translation Lookaside Buffer" or TLB.**
  - This is located on the CPU die itself.



Virtual Memory Translation Lookaside

- **To translate a virtual address:**
  - Check the TLB first to see if the required page table entry is there.
  - If yes, translate from entry in TLB. FAST!
  - If no, load entry from page table in main memory into TLB and perform translation.

Summary

- **RTOS are very different from general operating systems like Windows, MacOS or Linux:**
  - Focus is on compactness, performance guarantees and reliability.
  - Lacking threads, virtual memory, memory management and file systems.
- **To round off your knowledge of OS, we've looked at:**
  - Threads
  - Virtual Memory.
- **We will also be looking at file systems and memory management.**