LAU
الجامعة اللبنانية الأميركية
Lebanese American University

CSC 447: Parallel Programming for Multi-Core and Cluster Systems

Shared-Memory Programming: Processes, Threads, Data Races, and False Sharing

Instructor: Haidar M. Harmanani

Spring 2021

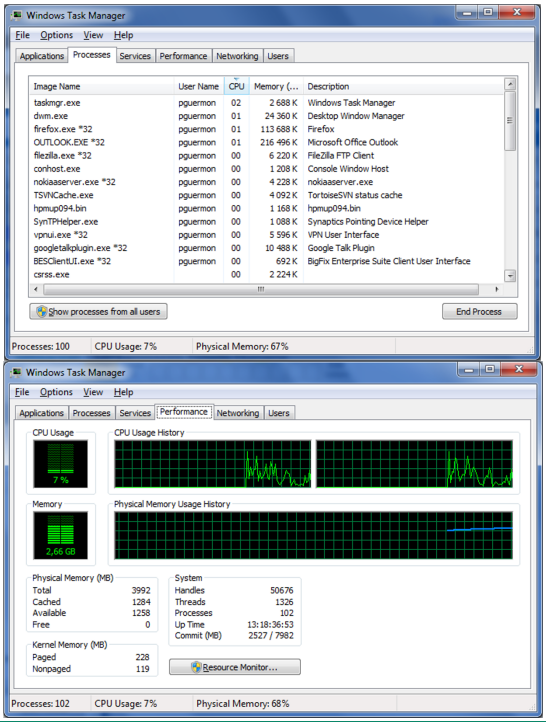
Processes

- Remember you have the choice to launch several independent software at the same time to take advantage of several cores (browser, email, music player, ...).
- It's easy, the operating system is taking care of associating in real time your software with cores and memory.
- You are using processes everyday !

intel Software

What Is a Process?

- A program in some state of execution
  - Code
  - Data
  - Logical address space
- Information about a process includes
  - Process state
  - Program counter
  - Values of core's registers
  - Memory management information



Windows

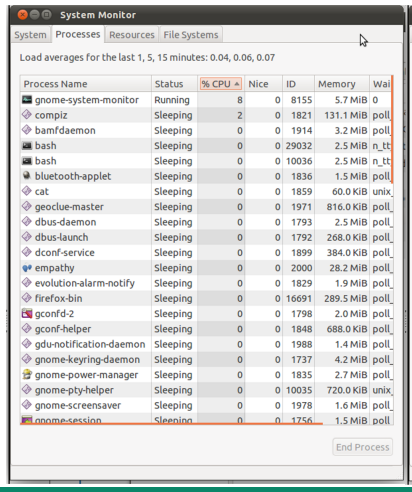
Process view

Core activity view

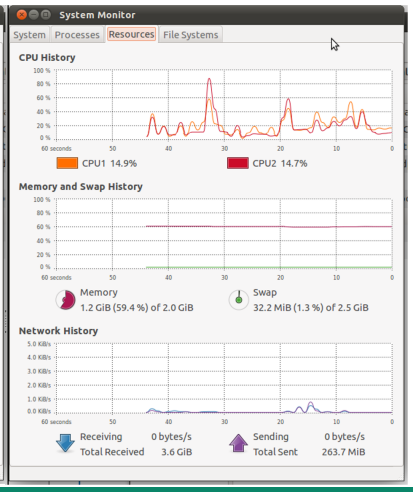
intel Software

Linux

Process view



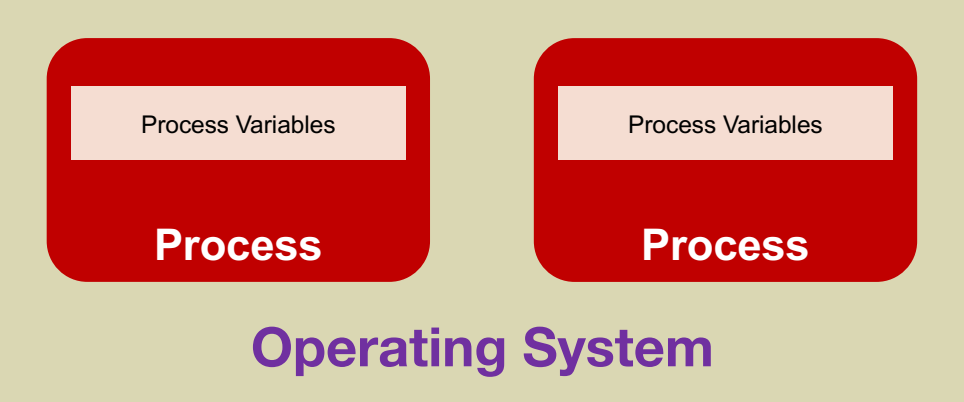
Core activity view



Processes

- Good side :
  - Each process is totally independent : *it's secure*
- Bad side :
  - But each process has a dedicated memory space so you *can't easily share* a variable in memory or code between processes.
  - Processes can communicate with other processes on the same machine or *other machines* over the network, but it's slow and generates a lot of CPU overhead.

Processes Memory Model



Example : Apache Web Server

- Apache web server is typically using processes. Dozens, hundreds of processes are launched automatically to answer client requests coming from the network.
- Good side : It's simple, secure and easy to code.
- Bad side : Communication between processes is complex, but web pages served to different clients usually do not need to share information.

LAU
الجامعة اللبنانية الأميركية
Lebanese American University

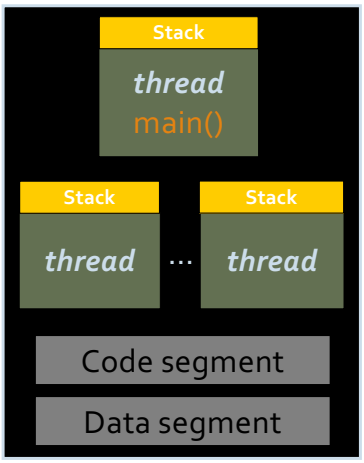
Threads

What Is a Thread?

"A unit of control within a process" — Carver and Tai

- Main thread executes program's "main" function
- Main thread may create other threads to execute other functions
- Threads have own program counter, copy of core registers, and stack of activation records
- Threads share process's data, code, address space, and other resources
- Threads have lower overhead than processes

Processes and Threads



- Modern operating systems load programs as processes
  - Resource holder
  - Execution
- A process starts executing at its entry point as a thread
- Threads can create other threads within the process
  - Each thread gets its own stack
- All threads within a process share code & data segments

Utility of Threads

- Threads are flexible enough to implement
  - Domain decomposition
  - Task decomposition
  - Pipelining

Example : MySQL Database

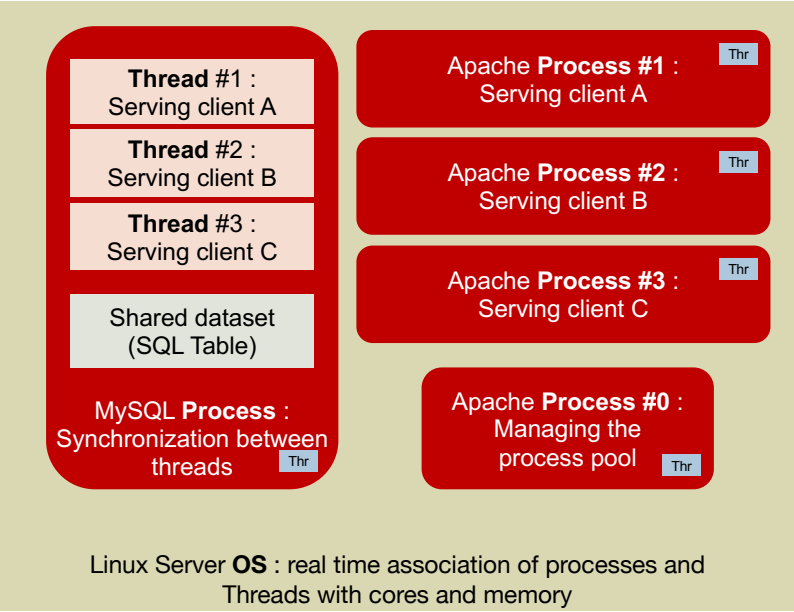
- MySQL Database server is launching threads to serve clients requests.
- Clients typically request information from the same dataset (SQL Table) : communication between threads is needed. That's why MySQL could not have easily used processes.
- MySQL developers had to take care of parallel programming risks to protect shared information. “Synchronization”.

intel Software

LAU
الجامعة اللبنانية الأميركية
Lebanese American University

Threads + Processes

Example : LAMP



Web Client A

Web Client B

Web Client C

Example : LAMP

- Each PHP generated page is processed from a different process
  - 3rd party developers coding in PHP or coding PHP itself don't need to worry about synchronization. Nothing is shared.
- MySQL is using threads to process various SQL requests. MySQL developers had to code synchronization to protect tables from concurrent access.
  - SQL users can also lock tables using SQL transactions.

LAU
الجامعة اللبنانية الأميركية
Lebanese American University

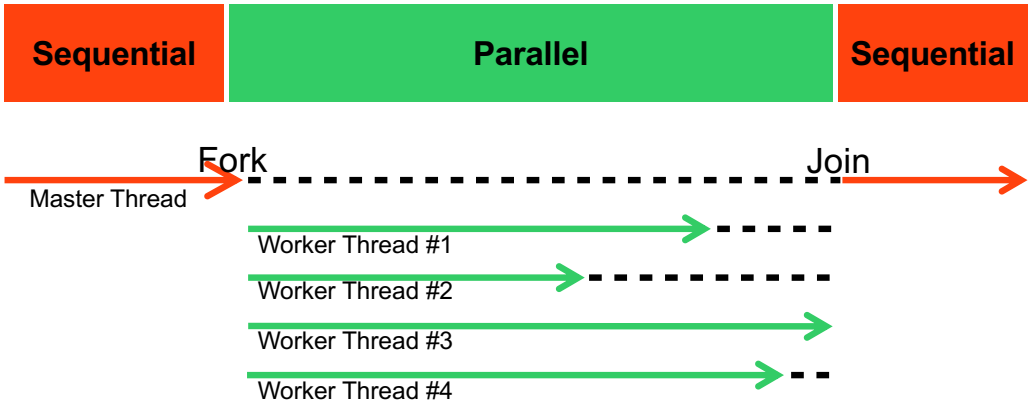
Fork/Join Model

Fork/Join Model

- When your main() function begins, that's the beginning of the first thread of the software.
  - You are in serial mode!
- When you enter the parallel part of your mutithreaded software, threads are launched.
  - Master thread forks worker threads.
- When all the worker threads have finished, they join.
- You are in serial mode again.
- End of main() function, end of process.

intel Software

Fork/Join Model



intel Software

Review of Unix Processes

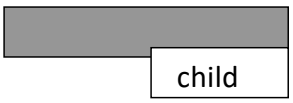
Read On Your Own

Unix Processes

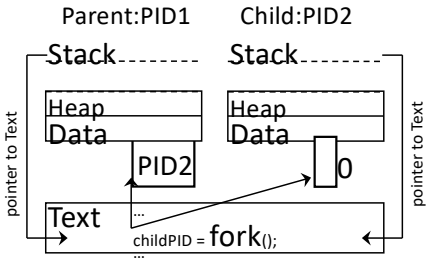
- A UNIX process is created by the operating system, and requires a fair amount of "overhead" including information about:
  - Process ID
  - Environment
  - Working directory.
  - Program instructions
  - Registers
  - Stack
  - Heap
  - File descriptors
  - Signal actions
  - Shared libraries
  - Inter-process communication tools (such as message queues, pipes, semaphores, or shared memory).

Fork System Call

```
#include <unistd.h>
pid_t  fork ();
```



- The *fork()* system call creates the copy of process that is executing.
- It is an only way in which a new process is created in UNIX.
- Returns: child PID >0 - for Parent process,
  0 - for Child process,
  -1 - in case of error (errno indicates the error).
- Fork reasons:
  -Make a copy of itself to do another task simultaneously
  -Execute another program (See *exec* system call)



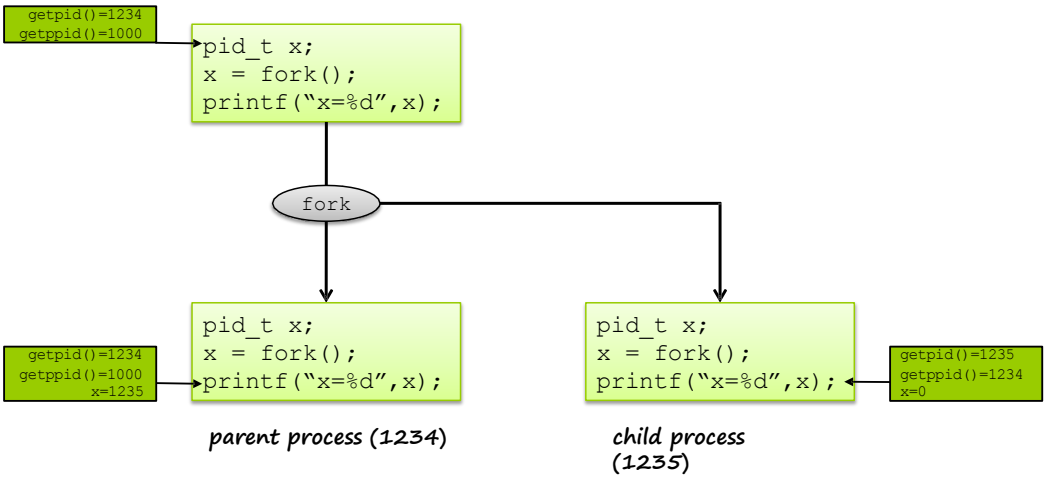

```
pid_t childPID;              /* typedef int pid_t */
...
childPID = fork();
if (childPID < 0)
{
   perror("fork failed");
   exit(1);
}
else if (childPID > 0)          /* This is Parent */
{
    /* Parent processing */
    ...
    exit(0);
}
else   /* childPID == 0 */      /* This is Child */
{
    /* Child processing  */
    ...
    exit(0);
}
```

Child Process has Shared Text and Copied Data.

It differs from Parent Process by following attributes:

- PID (new)
- Parent PID (= PID of Parent Process)
- Own copies of Parent file descriptors
- Time left until alarm signal reset to 0

The fork() System Call [2/3]



The Fork of Death!

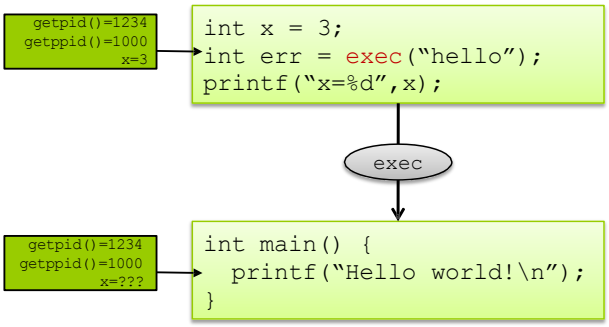
- You must be careful when using fork()!!
  - Very easy to create very harmful programs

```
while (1) {
  fork();
}
```

- Finite number of processes allowed
  - PIDs are 16-bit integers, maximum 65536 processes!
- Administrators typically take precautions
  - Limited quota on the number of processes per user
  - Try this at home ☺

The exec() System Call [1/4]

- exec() allows a process to switch from one program to another
  - Code/Data for that process are destroyed
  - Environment variables are kept the same
  - File descriptors are kept the same
  - New program's code is loaded, then started (from the beginning)
  - There is no way to return to the previously executed program!



The exec() System Call [2/4]

- fork() and exec() could be used separately
  - Imagine examples when this could happen?
- But most commonly they are used together:

```
if (fork()==0)     /* child */
{
  exec("hello");   /* load & execute new program */
  perror("Error calling exec()!\n");
  exit(1);
}
else  /* parent */
{
  ...
}
```

Shared Memory

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget(key_t key, int size, int shmflg);
```

- Shared memory segments must be created, then attached to a process to be usable

```
#include <sys/types.h>
#include <sys/shm.h>
void *shmat(int shmid, const void *shmaddr,
int shmflg);
```

- Must detach and destroy them once done

```
int shmdt(const void *shmaddr);
```

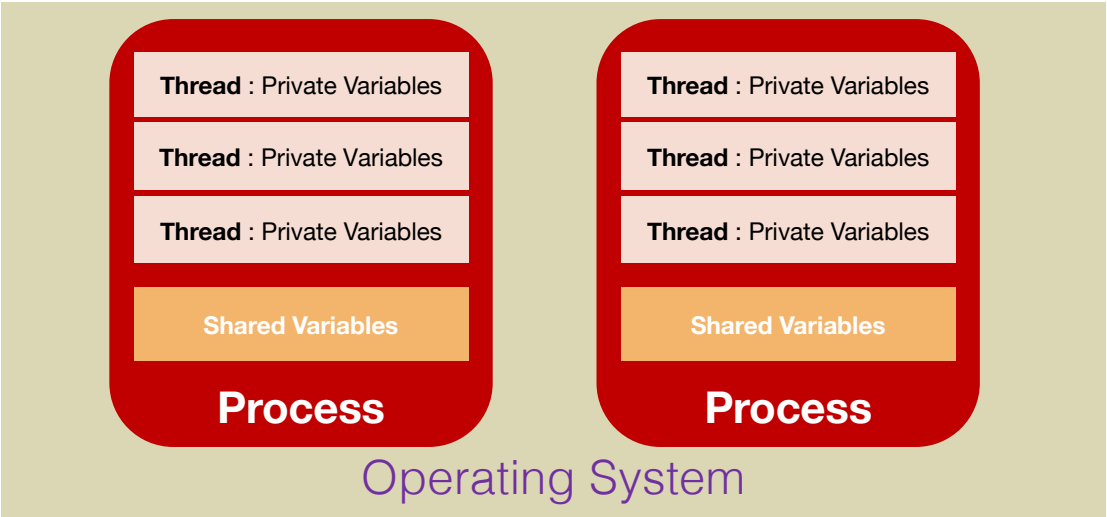

Example: Shell Skeleton

```
while (1) // Infinite
   print_prompt();
   read_command(command, parameters);
   if (fork()) {
      //Parent
      wait(&status);
   }
   else {
         execve(command, parameters, NULL);
         }
}
```

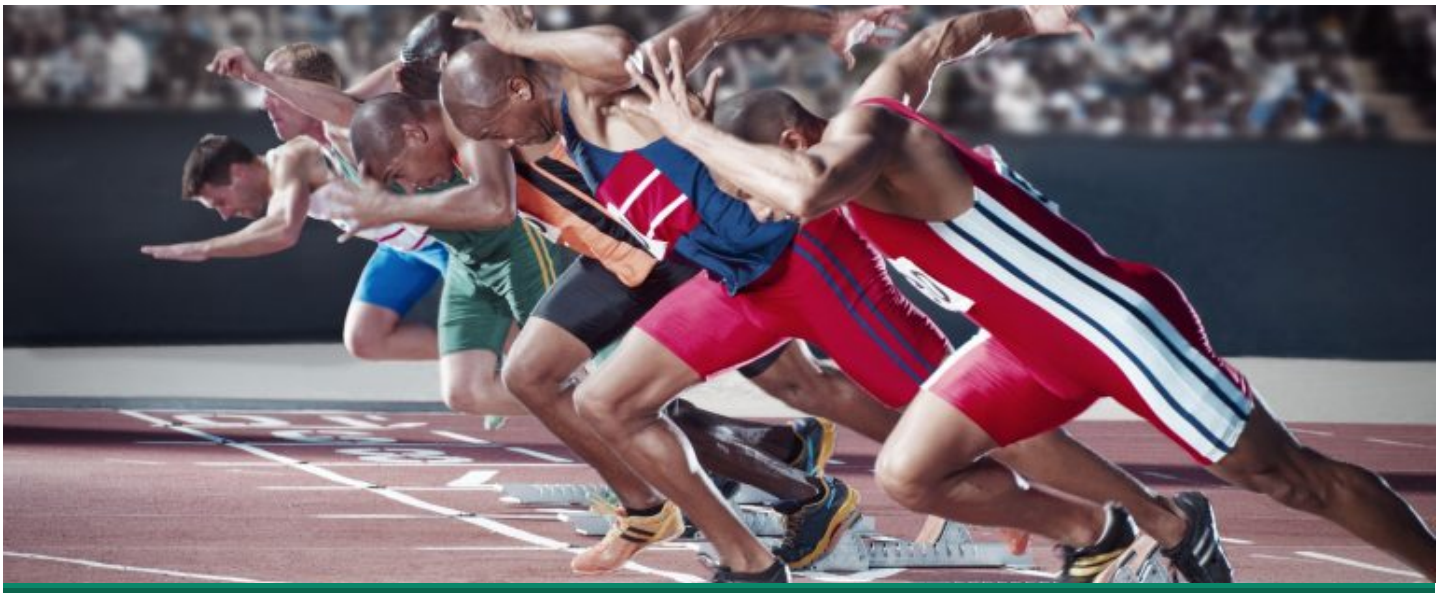
LAU
الجامعة اللبنانية الأميركية
Lebanese American University

Threads Memory Model

Threads Memory Model



intel Software



Shared-Memory Programming: Data Races and False Sharing

Consider a Simple Problem

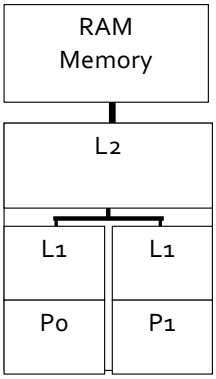
- Count the 3s in `array[]` of `length` values
- Definitional solution …
  - Sequential program

```
count = 0;
for (i=0; i<length; i++)
  {
    if (array[i] == 3)
       count += 1;
  }
```

Write A Parallel Program

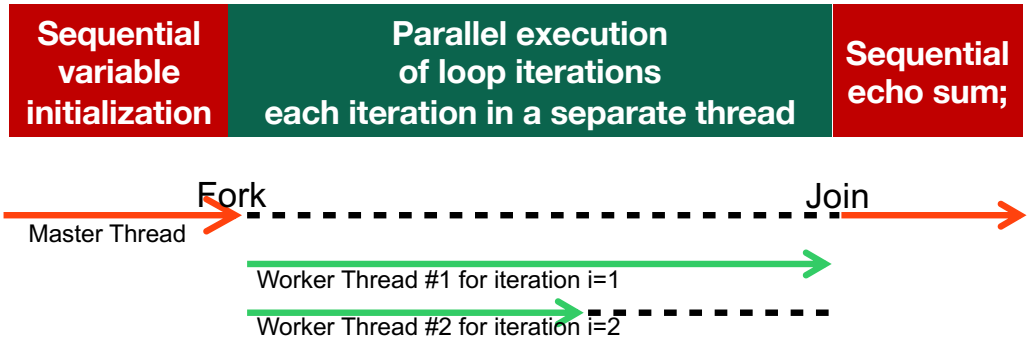
- Need to know something about machine … use multicore architecture

How would you solve it in parallel?



Sharing variables – Parallel run

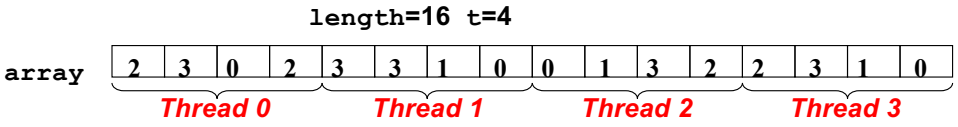
- Now, let's say you find a way (let's see later how) to compute iteration i=1 and i=2 in two separate threads to save execution time.



intel Software

Divide Into Separate Parts

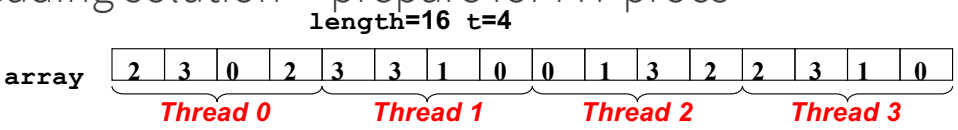
- Threading solution -- prepare for MT procs




```
int length_per_thread = length/t;
int start = id * length_per_thread;
for (i=start; i<start+length_per_thread; i++)
    {
       if (array[i] == 3)
         count += 1;
    }
```


Divide Into Separate Parts

- Threading solution -- prepare for MT procs

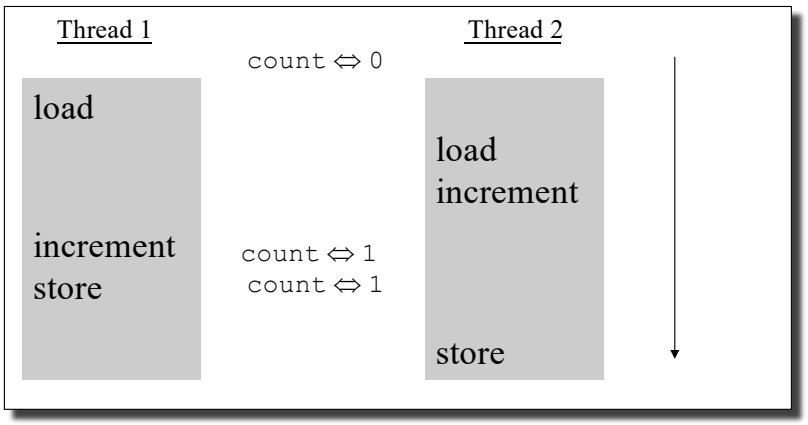



```
int length_per_thread = length/t;
int start = id * length_per_thread;
for (i=start; i<start+length_per_thread; i++)
    {
       if (array[i] == 3)
         count += 1;
    }
```

Doesn't actually get the right answer

Races

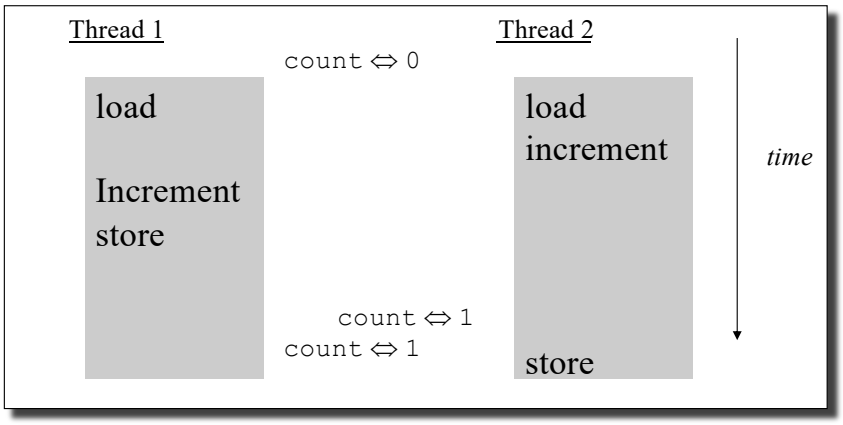
- Two processes interfere on memory writes



Races

- Two processes interfere on memory writes

Try 1



Race condition

- Cause : `count` is shared between threads.
- Solution : Need to explain to the compiler that sum has to be protected from random parallel access
  - How?

Serial bugs and parallel bugs are different.
If you are aware of the problem, solutions are easy to implement.

Protect Memory References

- Protect Memory References

```
mutex m;
for (i=start; i<start+length_per_thread; i++)
     {
     if (array[i] == 3)
        {
           mutex_lock(m);
             count += 1;
           mutex_unlock(m);
        }
     }
```

Protect Memory References

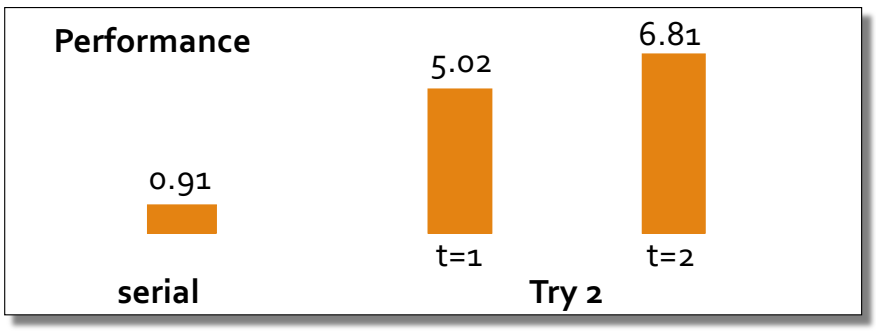
- Protect Memory References

```
mutex m;
for (i=start; i<start+length_per_thread; i++)
     {
     if (array[i] == 3)
        {
           mutex_lock(m);
             count += 1;
           mutex_unlock(m);
        }
     }
```

Try 2

Correct Program Runs Slow

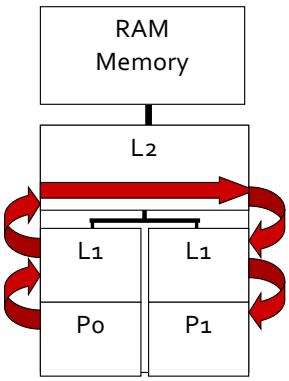
- Serializing at the mutex



  - The processors wait on each other

Closer Look: Motion of count, m

- Lock Reference and Contention




```
mutex m;
for (i=start; i<start+length_per_thread; i++)
      {
      if (array[i] == 3)
         {
            mutex_lock(m);
              count += 1;
            mutex_unlock(m);
         }
      }
```


Accumulate Into Private Count

- Each processor adds into its own memory; combine at the end

```
for (i=start; i<start+length_per_thread; i++)
     {
        if (array[i] == 3)
           {
              private_count[t] += 1;
           }
      }
   mutex_lock(m);
      count += private_count[t];
   mutex_unlock(m);
```

Accumulate Into Private Count

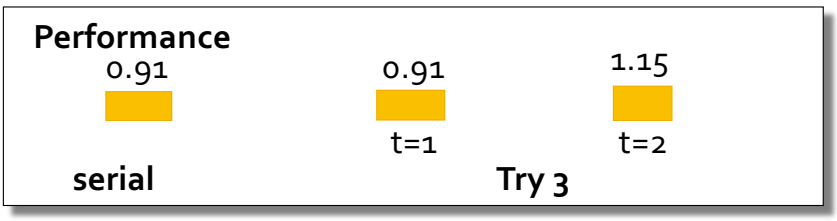
- Each processor adds into its own memory; combine at the end

Try 3

```
for (i=start; i<start+length_per_thread; i++)
     {
        if (array[i] == 3)
           {
              private_count[t] += 1;
            }
       }
mutex_lock(m);
  count += private_count[t];
mutex_unlock(m);
```


Keeping Up, But Not Gaining

- Sequential and 1 processor match, but it's a loss with 2 processors



False Sharing

False Sharing

- Do these loops have the same run time? If not, which one is faster?

```
int[] arr = new int[64 * 1024 * 1024];

// Loop 1 for (int i = 0; i < arr.Length; i++) arr[i] *= 3;

// Loop 2 for (int i = 0; i < arr.Length; i += 16) arr[i] *= 3;
```

- Although not expected, the have the same run time!

See: http://igoro.com/archive/gallery-of-processor-cache-effects/

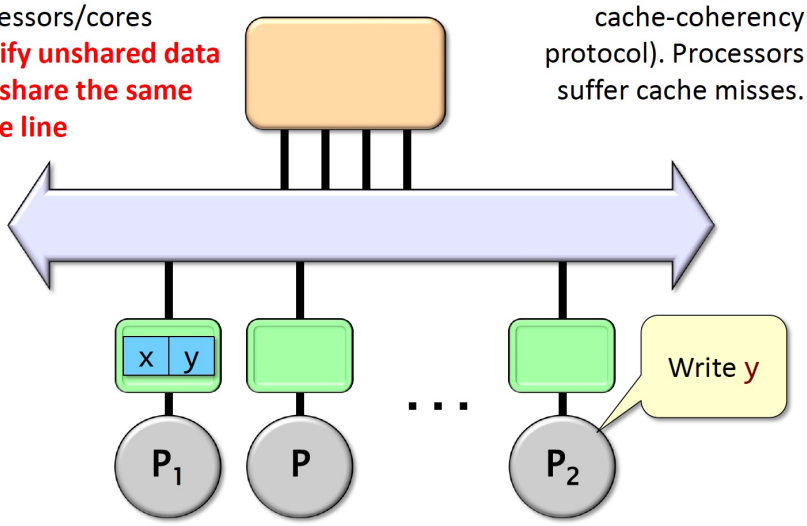
False Sharing

- Why?
  - The reason is that CPUs do not access memory byte by byte.
    - They fetch memory in chunks of (typically) 64 bytes, called cache lines.
  - When a particular memory location is read, the entire cache line is fetched from the main memory into the cache.
  - Accessing values in different cache lines is expensive!
- In the loop example, 16 ints take up 64 bytes (one cache line), for-loops with a step between 1 and 16 have to touch the same number of cache lines: all of the cache lines in the array. But once the step is 32, we'll only touch roughly every other cache line, and once it is 64, only every fourth.
- Alignment of data determines whether an operation touches one or two cache lines.

Another Problem…

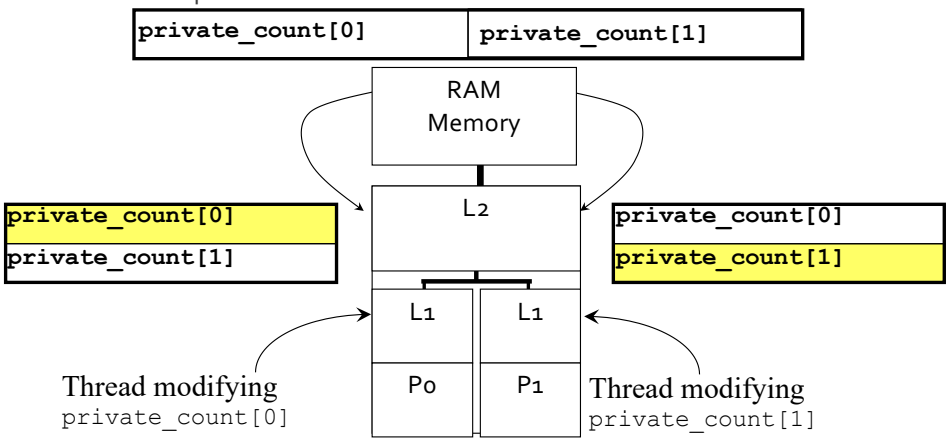
False sharing: threads running on different processors/cores **modify unshared data** that **share the same cache line**

Ping-pong effect on cache-line (due to cache-coherency protocol). Processors suffer cache misses.



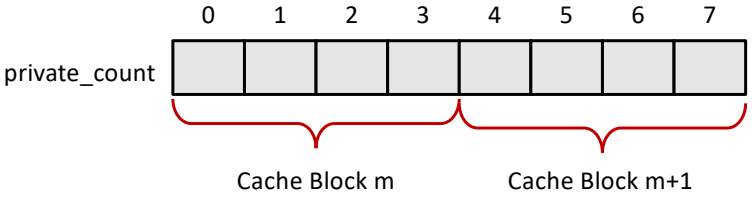
Back to our example…

- Private var ≠ private cache-line



False Sharing

- Coherency maintained on cache blocks
- To update `private_count[i]`, thread `i` must have exclusive access
  - Threads sharing common cache block will keep fighting each other for access to block



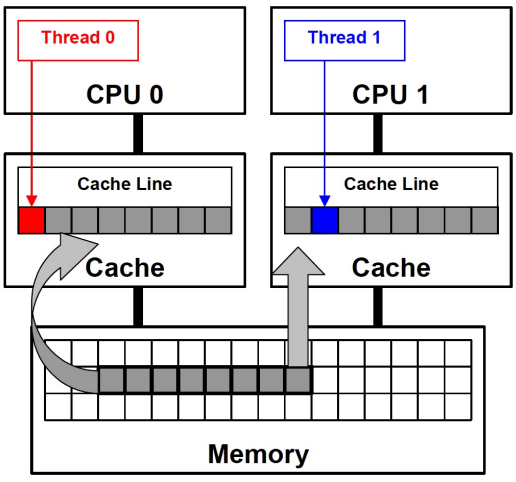
Force Into Different Lines

- Padding the private variables forces them into separate cache lines and removes false sharing

```
struct padded_int
  {   int value;
      char padding[128];
  }   private_count[MaxThreads];
```


Conclusion

- False sharing is a well-known performance issue on SMP systems where each processor has a local cache
- False sharing occurs when threads on different processors modify variables that reside on the same cache line
  - It is called false sharing because each thread is not actually sharing access to the same variable



And now the solution

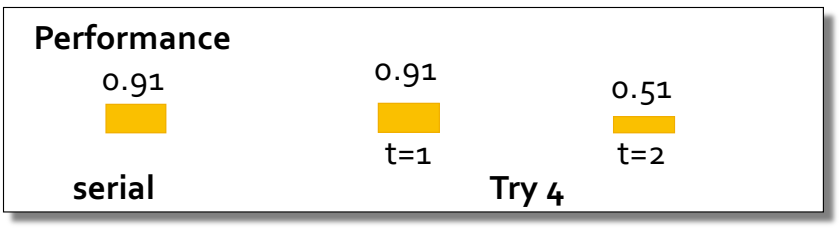
- Padding the private variables forces them into separate cache lines and removes false sharing

```
struct padded_int
  {  int value;
     char padding[128];
  }  private_count[MaxThreads];
```

Try 4

Success!!

- Two processors are almost twice as fast



- Is this the best solution???

Other solutions

- Use compiler directives to force individual variable alignment
  - `__declspec (align(64)) int thread1_global_variable` (Intel)

False Sharing: Solution #2

- Pad the data using a data structure

```
struct ThreadParams
{
 // For the following 4 variables: 4*4 = 16 bytes
 unsigned long thread_id;
 unsigned long v; // Frequent read/write access variable
 unsigned long start;
 unsigned long end;

 // expand to 64 bytes to avoid false-sharing
 // (4 unsigned long variables + 12 padding)*4 = 64
 int padding[12];
};

__declspec (align(64)) struct ThreadParams Array[10];
```

False Sharing: Solution #3

- Reduce the frequency of false sharing by using thread-local copies of data

```
struct ThreadParams
{
 // For the following 4 variables: 4*4 = 16 bytes
 unsigned long thread_id;
 unsigned long v; //Frequent read/write access variable
 unsigned long starat;
 unsigned long end;
};

void threadFunc(void *parameter)
{
 ThreadParams *p = (ThreadParams*) parameter;
 // local copy for read/write access variable
 unsigned long local_v = p->v;

 for(local_v = p->start; local_v < p->end; local_v++)
 {
 // Functional computation
 }

 p->v = local_v; // Update shared data structure only once
}
```


LAU
الجامعة اللبنانية الأميركية
Lebanese American University

Final Comments on Multicore and Manycore computing

Multicore vs. Manycore

- A multicore processor is a single computing component with two or more “independent” processors (called "cores")
  - Intel Core i9 with 10 cores with Hyperthreading
- A many-core processor is a multi-core processor (probably heterogeneous) in which the number of cores is large enough that traditional multi-processor techniques are no longer efficient.
  - DGX A100 from NVIDIA with 6912 CUDA cores and 3456 FP cores!

10TH GEN INTEL® CORE™ DESKTOP PROCESSORS

PROCESSOR NUMBER	BASE CLOCK SPEED (GHZ)	INTEL® TURBO BOOST TECHNOLOGY 2.0 MAXIMUM SINGLE CORE TURBO FREQUENCY (GHZ)	INTEL® TURBO BOOST MAX TECHNOLOGY 3.0 FREQUENCY (GHZ)	INTEL® THERMAL VELOCITY BOOST TECHNOLOGY SINGLE / ALL CORE TURBO FREQUENCY (GHZ)[4]	INTEL® ALL CORE TURBO FREQUENCY (GHZ)	CORES/ THREADS	THERMAL DESIGN POWER	UNLOCKED[2]	PLATFORM PCIE 3.0 LANES	MEMORY SUPPORT[1]	PROCESSOR GRAPHICS	INTEL® OPTANE™ MEMORY[3]	RCP PRICING (USD 1K)
i9-10900K	Up to 3.7	Up to 5.1	Up to 5.2	Up to 5.3 / 4.9	Up to 4.8	10/20	125	✔	Up to 40	Two Channels DDR4-2933	Intel® UHD Graphics 630	✔	$488
i9-10900KF	Up to 3.7	Up to 5.1	Up to 5.2	Up to 5.3 / 4.9	Up to 4.8	10/20	125	✔	Up to 40	Two Channels DDR4-2933		✔	$472
i9-10900	Up to 2.8	Up to 5.0	Up to 5.1	Up to 5.2 / 4.6	Up to 4.5	10/20	65		Up to 40	Two Channels DDR4-2933	Intel® UHD Graphics 630	✔	$439
i9-10900F	Up to 2.8	Up to 5.0	Up to 5.1	Up to 5.2 / 4.6	Up to 4.5	10/20	65		Up to 40	Two Channels DDR4-2933		✔	$422
i7-10700K	Up to 3.8	Up to 5.0	Up to 5.1	NA	Up to 4.7	8/16	125	✔	Up to 40	Two Channels DDR4-2933	Intel® UHD Graphics 630	✔	$374
i7-10700KF	Up to 3.8	Up to 5.0	Up to 5.1	NA	Up to 4.7	8/16	125	✔	Up to 40	Two Channels DDR4-2933		✔	$349
i7-10700	Up to 2.9	Up to 4.7	Up to 4.8	NA	Up to 4.6	8/16	65		Up to 40	Two Channels DDR4-2933	Intel® UHD Graphics 630	✔	$323
i7-10700F	Up to 2.9	Up to 4.7	Up to 4.8	NA	Up to 4.6	8/16	65		Up to 40	Two Channels DDR4-2933		✔	$298

Intel® processor numbers are not a measure of performance. Processor numbers differentiate features within each processor family, not across different processor families. All processors are lead-free (per EU RoHS directive July 2006) and halogen free (residual amounts of halogens are below November 2007 proposed IPC/JEDEC J-STD-709 standards). All processors support Intel® Virtualization Technology (Intel® VT-x).

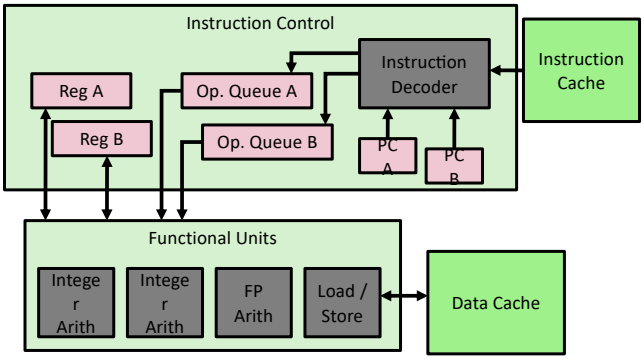
- [1] DDR4 maximum speed support is 1 and 2 DPC for UDIMMs but only 1 DPC for SODIMMs. DDR4 2DPC UDIMM 2933 or 2666 is capable when same UDIMM part number are populated with in each channel.
- [3] Intel® Optane™ memory requires specific hardware and software configuration. Visit www.intel.com/Optanememory for configuration requirements.
- [4] Intel® Thermal Velocity Boost feature is opportunistic at a temperature of 70ºC or lower and when turbo power budget is available. The frequency gain and duration is dependent on the workload (best for bursty workloads), capabilities of the individual processor, and the processor cooling solution. Frequencies may reduce over time and longer workloads may start at the max frequency but drop as processor temperature increases.

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration. No product or component can be absolutely secure. Check with your system manufacturer or retailer or learn more at intel.com.

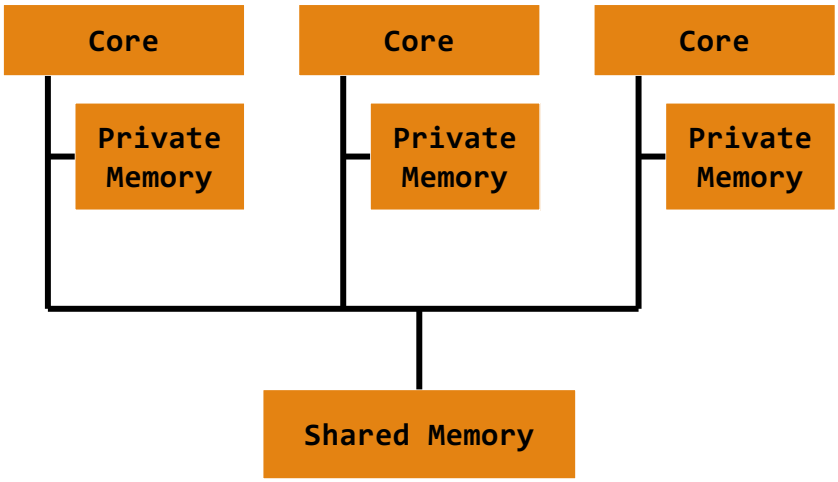
Embargoed until April 30, 2020 at 6am Pacific Time

Hyperthreading

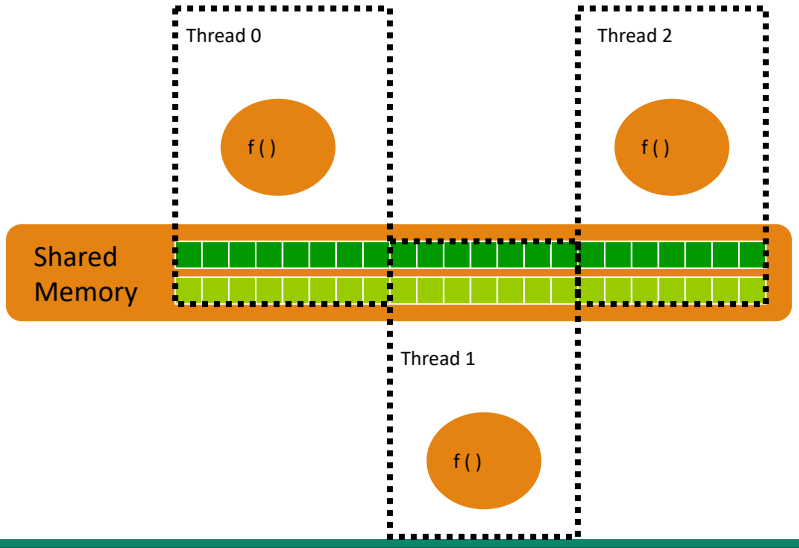
- Replicate enough instruction control to process K instruction streams
  - K copies of all registers
  - Share functional units
- Hyper-threading (HT - Intel) is the ability, of the hardware and the system, to schedule and run two threads or processes simultaneously on the same processor core



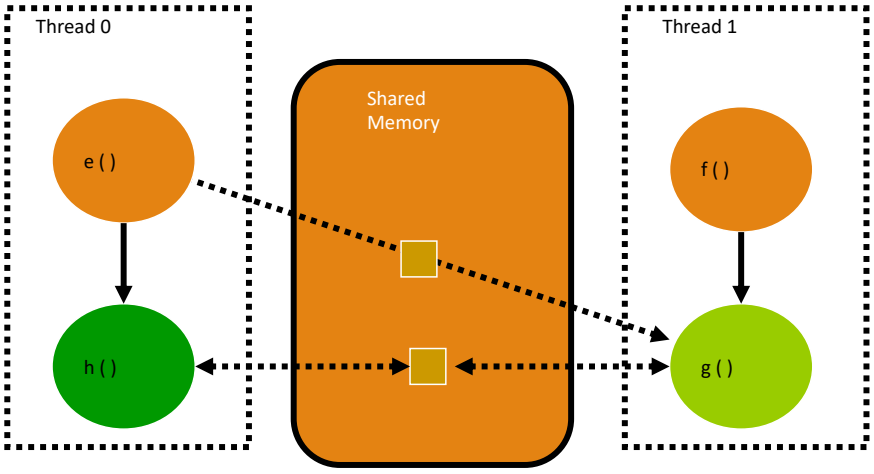
Recall: The Shared-Memory Model



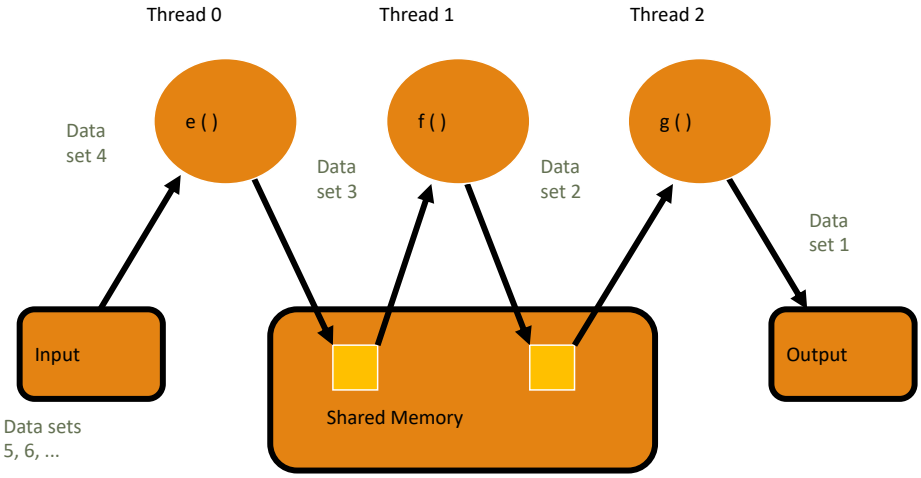
Domain Decomposition Using Threads



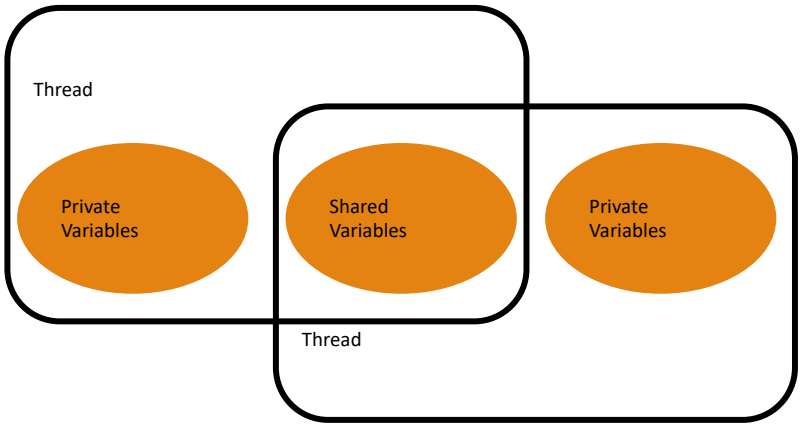
Task Decomposition Using Threads



Pipelining Using Threads



Shared versus Private Variables



Domain Decomposition

- Sequential Code:

```
int a[1000], i;

for (i = 0; i < 1000; i++)
    a[i] = foo(i);
```


Domain Decomposition

- Sequential Code:

```
int a[1000], i;
for (i = 0; i < 1000; i++) a[i] = foo(i);

Thread 0:
for (i = 0; i < 500; i++) a[i] = foo(i);

Thread 1:
for (i = 500; i < 1000; i++) a[i] = foo(i);
```

Domain Decomposition

- Sequential Code:
- `int a[1000], i;`
- `for (i = 0; i < 1000; i++) a[i] = foo(i);`
- Thread 0:
- `for (i = 0; i < 500; i++) a[i] = foo(i);`
- Thread 1:
- `for (i = 500; i < 1000; i++) a[i] = foo(i);`

Private

Shared

Task Decomposition

```
int e;

main () {
   int x[10], j, k, m;

   j = f(x, k);
   m = g(x, k);
...
}

int f(int *x, int k)
{
   int a;

   a = e * x[k] * x[k];
   return a;
}
```

```
int g(int *x, int k)
{
   int a;

   k = k-1;
   a = e / x[k];
   return a;
}
```

Task Decomposition

```
int e;

main () {
   int x[10], j, k, m;   j = f(x, k);  m = g(x, k);
}
```

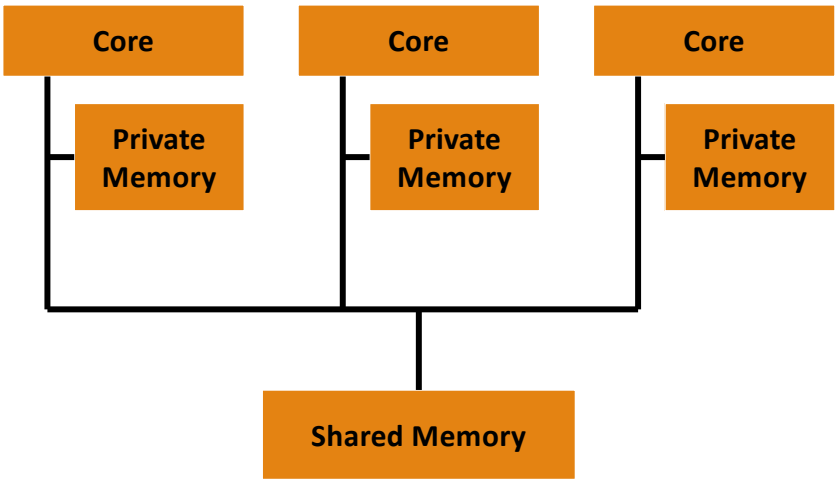
Thread 0

```
int f(int *x, int k)
{
   int a;   a = e * x[k] * x[k];   return a;
}
```

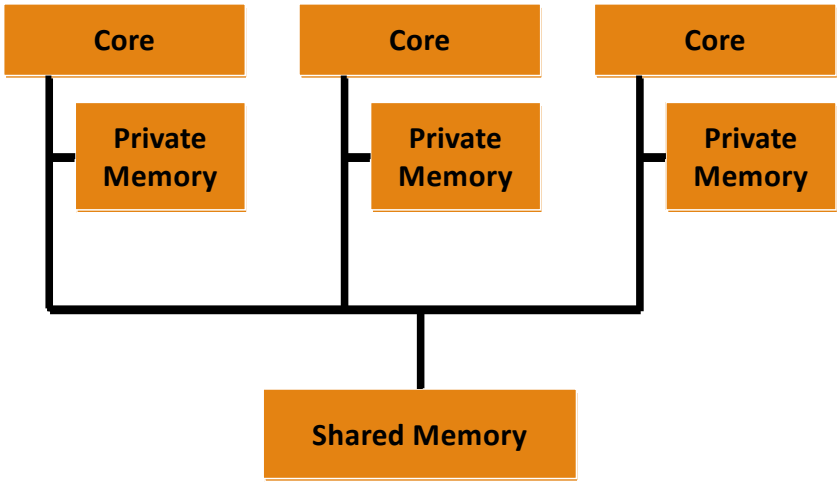
Thread 1

```
int g(int *x, int k)
{
   int a;   k = k-1;  a = e / x[k];   return a;
}
```


The Shared-Memory Model (Reprise)



The Threads Model



Final Comments

- All threads have equal priority read/write access to the same global (shared) memory.
- Threads also have their own private data.
  - Local variables for instance.
- Programmers are responsible for synchronizing any write access (protection) to globally shared data.