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

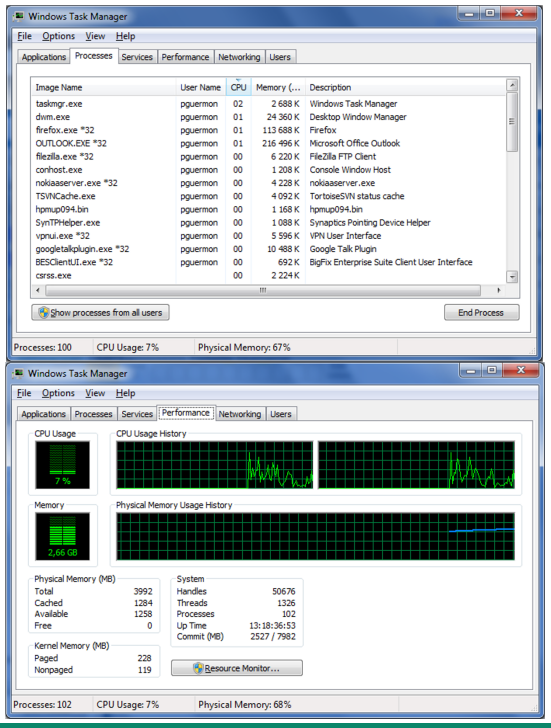
Instructor: Haidar M. Harmanani

Spring 2019

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



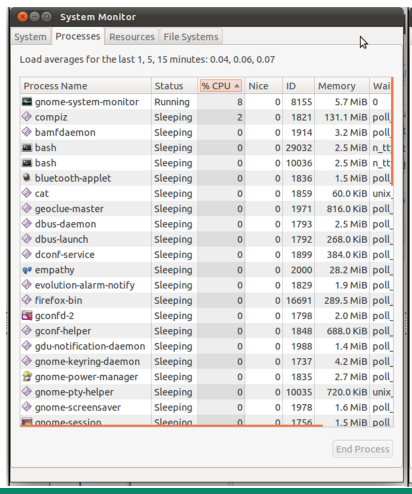
Windows

Process view

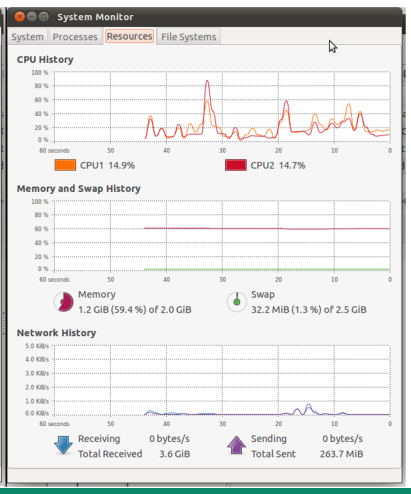
Core activity view

Linux

Process view



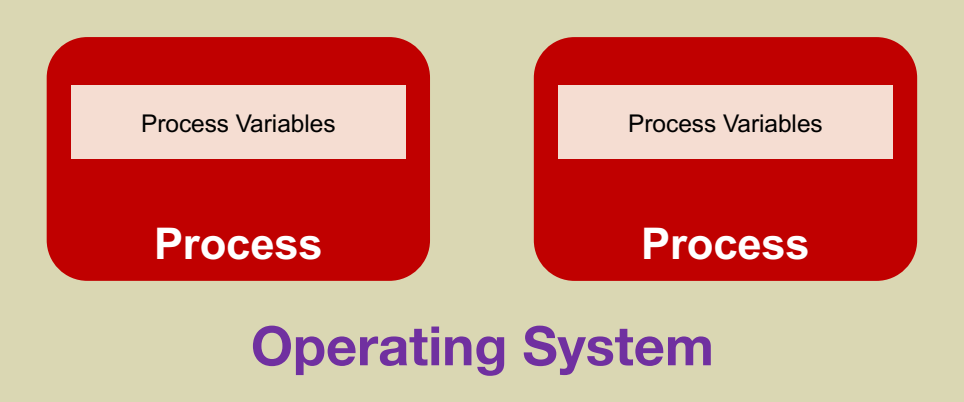
Core activity view



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

Threads

- Inside processes, you have thread(s).
- When your `main()` function begins, that's the beginning of the first thread of your software.
- If you create more threads, the operating system will allocate them transparently with cores and memory in real time, just like processes.
- Danger
  - Unlike processes, threads from the same process share memory (data and code).
  - They can communicate easily, but it's dangerous if you don't protect your variables correctly.

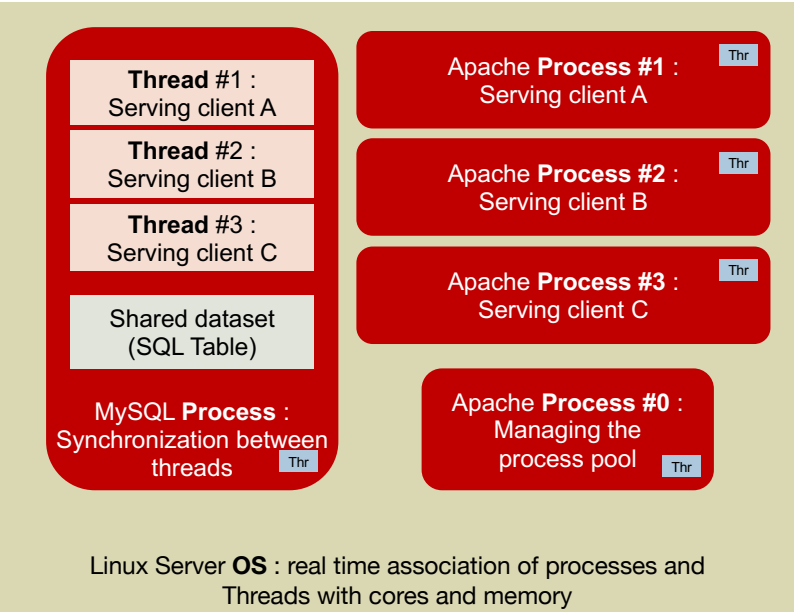
Example : MySQL Database

- 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”.

LAU
الجامعة اللبنانية الأميركية
Lebanese American University

Threads + Processes

Example : LAMP



Web Client A

Web Client B

Web Client C

intel Software

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.

intel Software

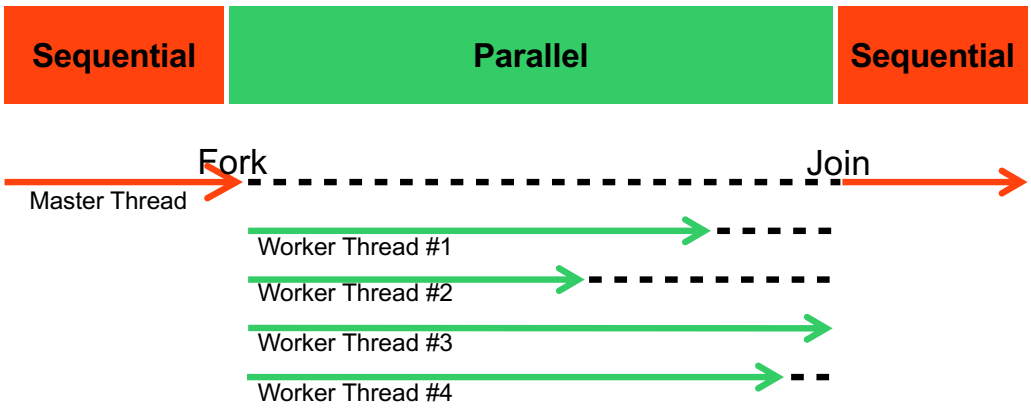
LAU
الجامعة اللبنانية الأميركية
Lebanese American University

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.

Fork/Join Model



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).

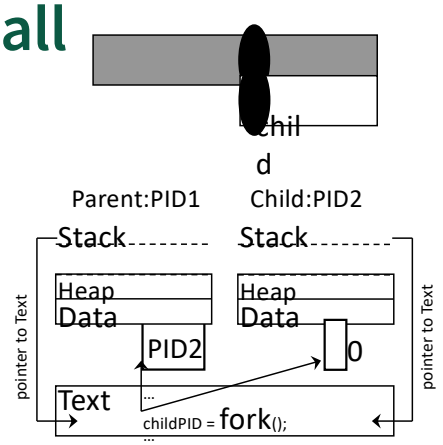
Fork System Call

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

s 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);
}
```

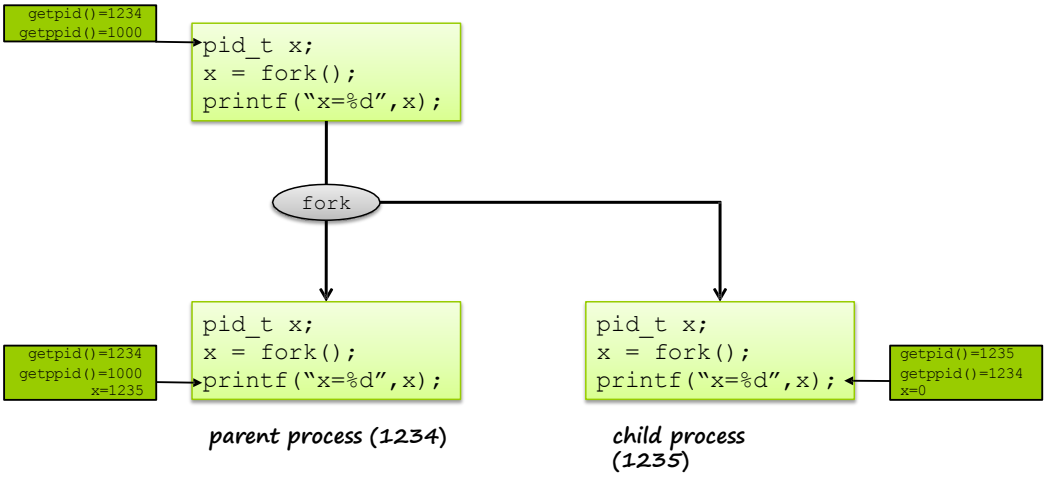


Child Process has Shared Text and Copied Data.

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!

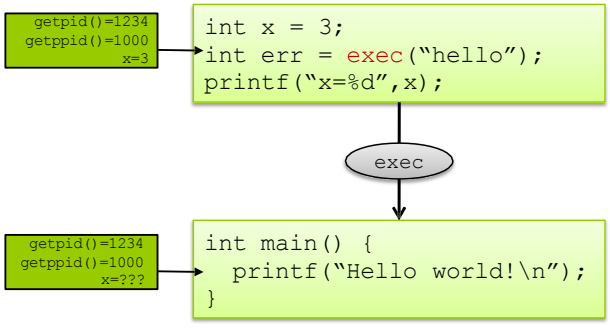
- You must be careful when using fork()!!
  - Very easy to create very harmful programs

```
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 */
{
  ...
}
```


Waiting for Children Processes [1/2]

- To block until some child process completes:

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
```

- wait() waits for any child to complete
  - Also handles an already completed (zombie) child returns instantly
  - Returns the PID of the completed child
  - status indicates the process' return status

Waiting for Children Processes [2/2]

- waitpid() gives you more control:

```
pid_t waitpid(pid_t pid, int *status, int option);
```

- pid specifies which child to wait for (-1 means any)
- option=WNOHANG makes the call return immediately, if no child has already completed (otherwise use option=0)

```
void sig_chld(int sig) {
  pid_t pid;
  int stat;
  while ( (pid=waitpid(-1, &stat, WNOHANG)) > 0 ) {
    printf("Child %d exited with status %d\n", pid, stat);
  }
  signal(sig, sig_chld);
}
```

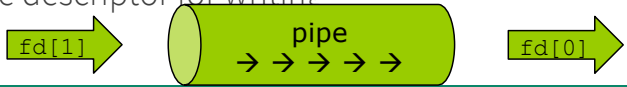
- Example: a SIGCHLD signal handler
- Question: Why is the while statement necessary?

Pipes [1/3]

- A pipe is a unidirectional communication channel between
- processes
  - Write data on one end – Read it at the other end
  - Bidirectional communication? Use TWO pipes.
- Creating a pipe

```
#include <unistd.h>
int pipe(int fd[2]);
```

- The return parameters are:
  - fd[0] is the file descriptor for reading
  - fd[1] is the file descriptor for writing



Pipes [2/3]

- Pipes are often used in combination with fork()

```
int main() {
  int pid, fd[2];
  char buf[64];

  if (pipe(fd)<0) exit(1);

  pid = fork();
  if (pid==0)        /* child */
  {
    close(fd[0]);    /* close reader */
    write(fd[1],"hello, world!",14);
  }
  else {             /* parent */
  {
    close(fd[1]);    /* close writer */
    if (read(fd[0],buf,64) > 0)
      printf("Received: %s\n", buf);
    waitpid(pid,NULL,0);
```


Pipes [3/3]

- Pipes are used for example for shell commands like:

```
sort foo | uniq | wc
```

- Pipes can only link processes which have a common ancestor
  - Because children processes inherit the file descriptors from their parent
- What happens when two processes with no common ancestor want to communicate?
  - Named pipes (also called FIFO)
  - A special file behaves like a pipe
  - Assuming both processes agree on the pipe's filename, they can
- communicate
  - mkfifo() for creating a named pipe
  - open(), read(), write(), close()

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);
```

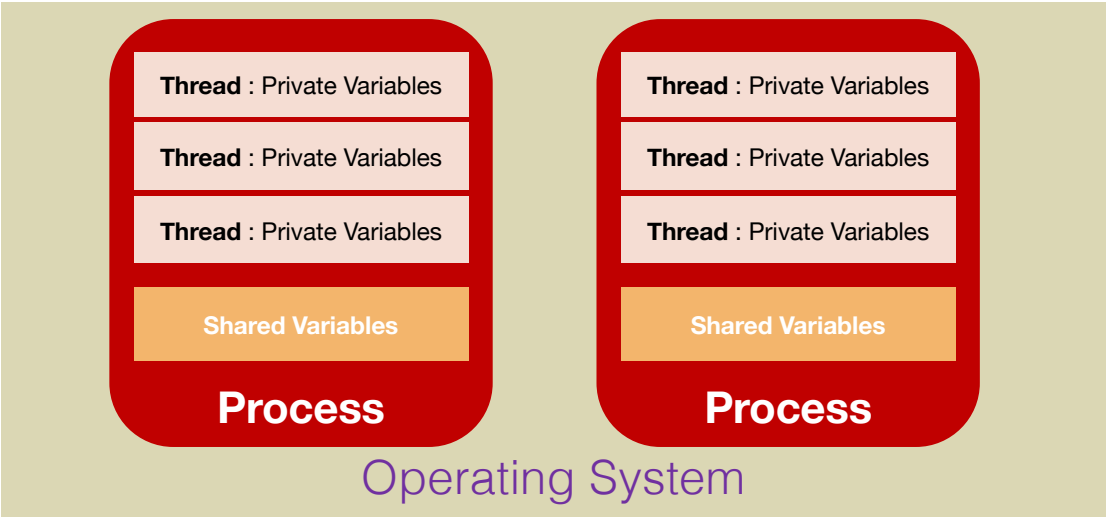

Example: Shell Skeleton

```
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

Sharing variables – Simple code

- Let's compute the sum of integers between 0 and n using the following loop :

```
int sum = 0;
int i, n = 3;

for (i = 1; i < n; i++) {
    sum += i;
}
printf("%d\n", sum);
```


Sharing variables – Serial run

- If you run it serially (2 iterations, one at a time) :

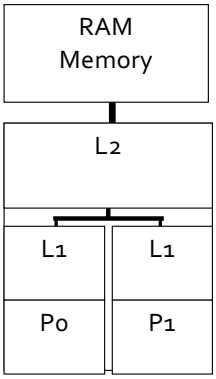
```
int n = 3;
for (i = 1; i < n; i++) {
    sum += i;
}

Starting point : sum=0
After 1st iteration : i=1 sum=1
After 2nd iteration : i=2 sum=3
Final result : sum=3
```

Write A Parallel Program

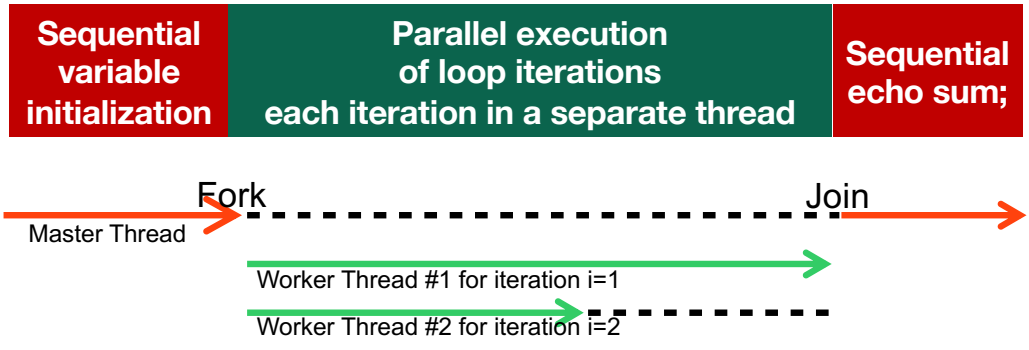
- Need to know something about machine … use multicore architecture

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

Sharing variables – Parallel run

- When you run it in parallel, each thread has to execute independently “sum += i;”
- It means 3 operations :
  - Read sum
  - Compute +i
  - Write sum.
- Depending on the timing of the two execution, you'll have a different result. Let's see why.

Sharing variables – Parallel run

- Case 1 :
  thread #1 and #2 read at the same time,
  thread #1 is writing first, then thread #2.

```
Starting point for thread #1 : sum=0
After thread #1 1rst iteration : i=1 sum=1
Starting point for thread #2 : sum=0
After thread #2 1rst iteration : i=2 sum=2
Final result : sum=2 (instead of 3)
```

Sharing variables – Parallel run

- Case 2 :
  thread #1 and #2 read at the same time,
  thread #2 is writing first, then thread #1.

```
Starting point for thread #2 : sum=0
After thread #2 1rst iteration : i=2 sum=2
Starting point for thread #1 : sum=0
After thread #1 1rst iteration : i=1 sum=1
Final result : sum=1 (instead of 3)
```


Sharing variables – Parallel run

- Case 3 :
  thread #1 read after thread #2 finished writing.
  (or the contrary)

```
Starting point for thread #1 : sum=0
After thread #1 1rst iteration : i=1 sum=1
Starting point for thread #2 : sum=1
After thread #2 1rst iteration : i=2 sum=3
Final result : sum=3
```

Race condition

- Launching threads is easy, and computing `sum += i` in parallel will use *multiple core*.
- But, depending on various uncontrollable conditions, your code running in parallel will return variable results.
  - It's called a *race condition*.
- Other types of typically parallel bugs are also possible.

Race condition

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

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

Solution: Semaphore / Mutex

```
void *sum_sem(void *vargp)
{
    int myid = *((int *)vargp);
    size_t start = myid * nelems_per_thread;
    size_t end = start + nelems_per_thread;
    size_t i;

    for (i = start; i < end; i++) {
        sem_wait(&semaphore);
        global_sum += i;
        sem_post(&semaphore);
    }
    return NULL;
}
```

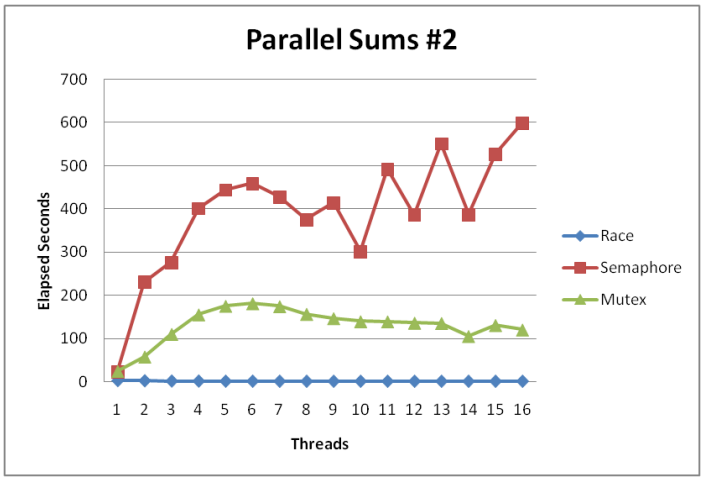
Semaphore

```
pthread_mutex_lock(&mutex);
global_sum += i;
pthread_mutex_unlock(&mutex);
```

Mutex

Semaphore / Mutex Performance

- Terrible Performance
  - 2.5 seconds ➔ ~10 minutes
- Mutex 3X faster than semaphore
- Clearly, neither is successful



Conclusions

- Parallel Compared to Sequential Programming
  - Has different costs, different advantages
  - Requires different, unfamiliar algorithms
  - Must use different abstractions
  - More complex to understand a program's behavior
  - More difficult to control the interactions of the program's components
  - Knowledge/tools/understanding more primitive

LAU
الجامعة اللبنانية الأميركية
Lebanese American University

Count 3s

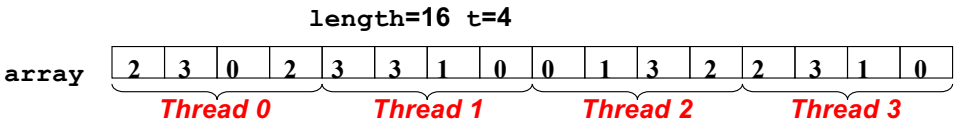
Consider a Simple Problem

- 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;
  }
```


Divide Into Separate Parts

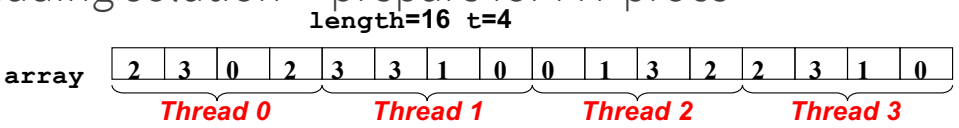
- Threading solution -- prepare for MT procs




```
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

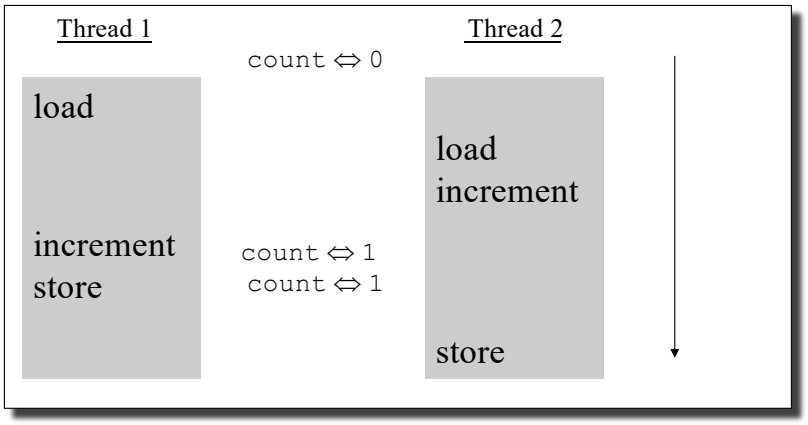



```
int length_per_thread = length/t;
int start = id * length_per_thread;
for (i=start; i<start+length_per_thread; i++)
    {
       if (array[i] == 3)
         count += 1;
    }
```

Doesn't actually get the right answer

Races

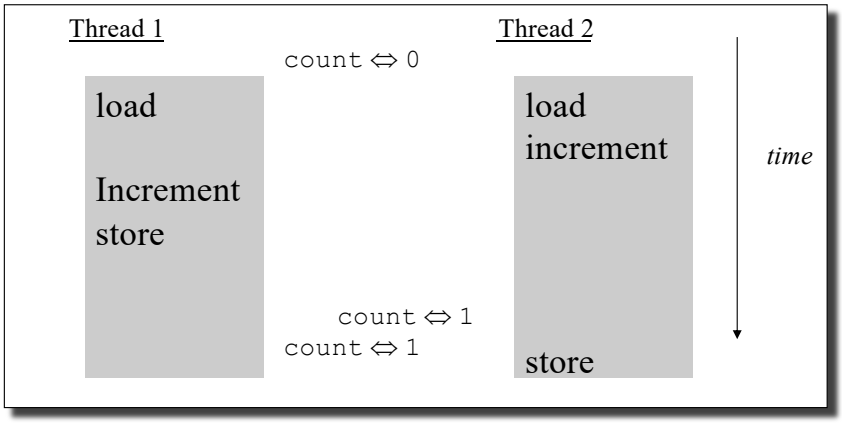
- Two processes interfere on memory writes



Races

- Two processes interfere on memory writes

Try 1



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

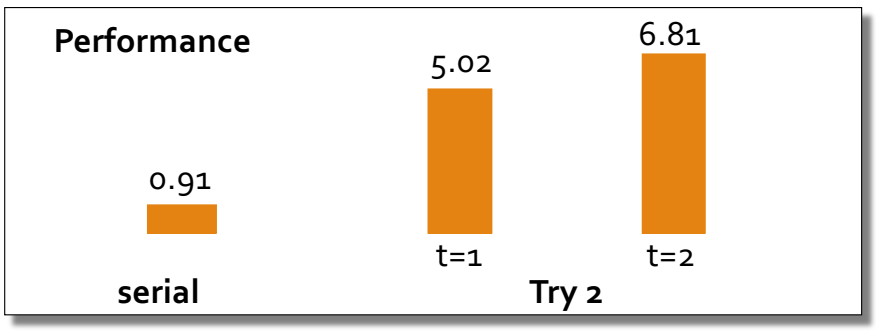
- Protect Memory References

```
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

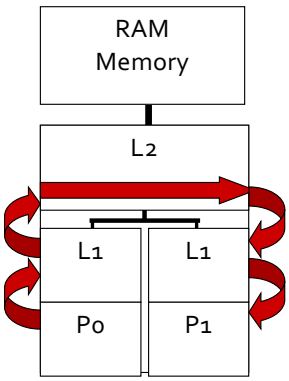
- Serializing at the mutex



  - 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

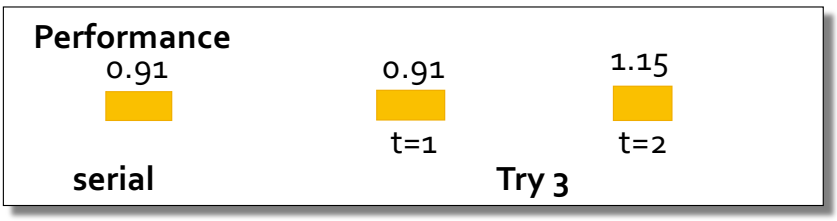
- Each processor adds into its own memory; combine at the end

```
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);
```

Try 3

Keeping Up, But Not Gaining

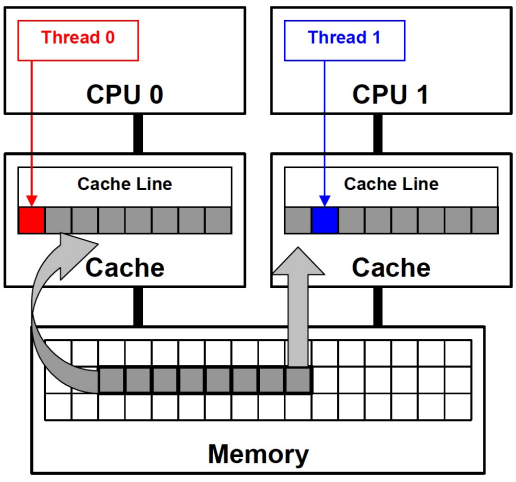
- Sequential and 1 processor match, but it's a loss with 2 processors



False Sharing

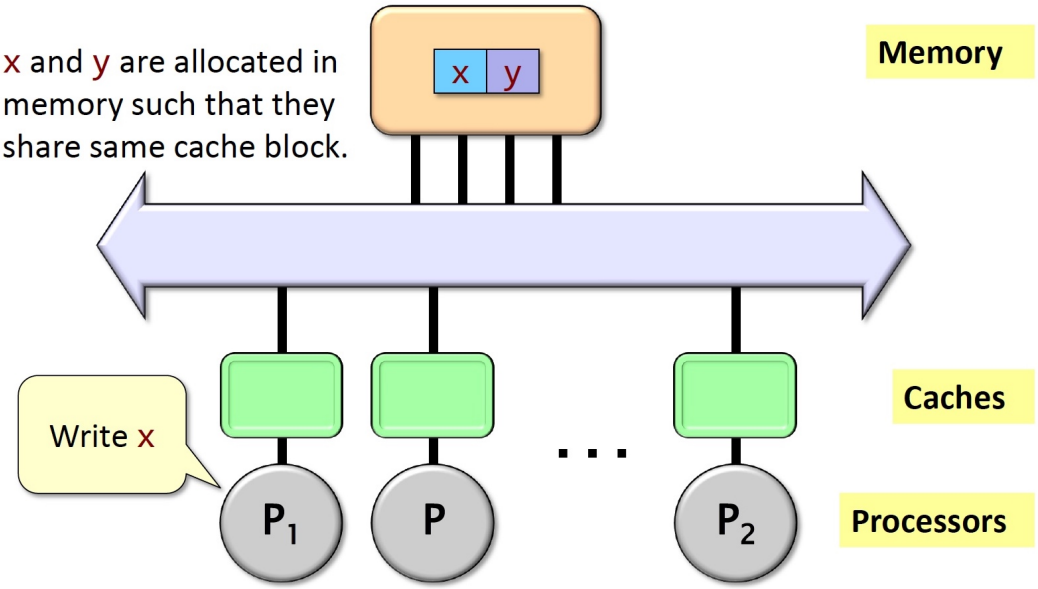
False Sharing

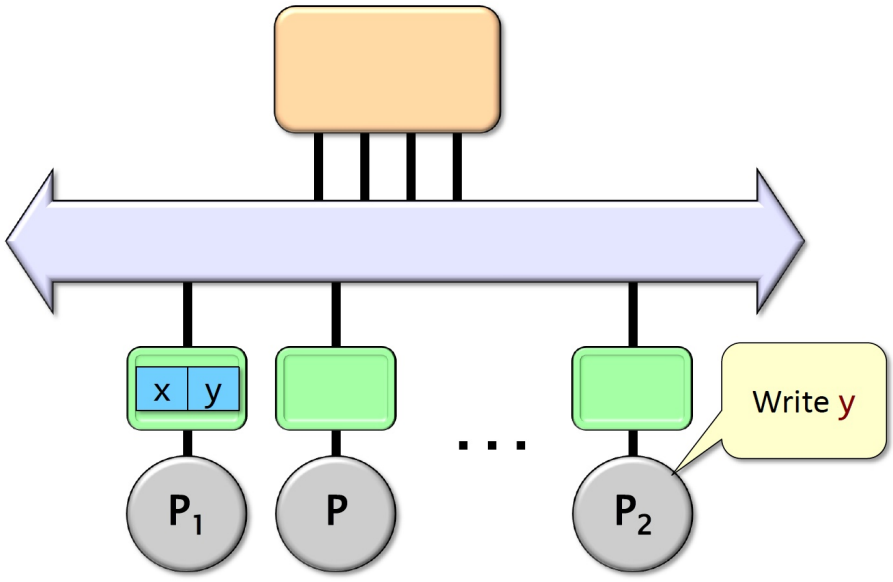
- A well-known performance issue on SMP systems where each processor has a local cache
- False sharing occurs when threads on different processors modify variables that reside on the same cache line
  - It is called false sharing because each thread is not actually sharing access to the same variable

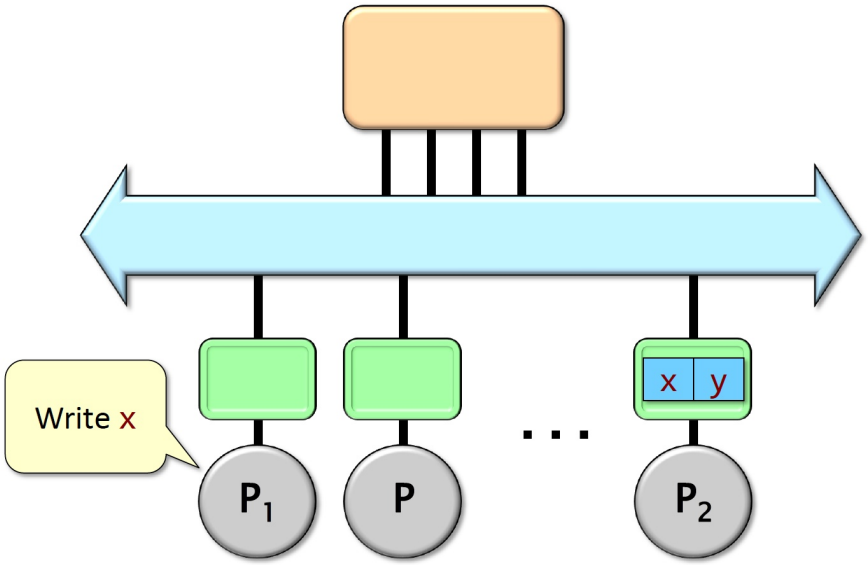


What is False Sharing?

x and y are allocated in memory such that they share same cache block.

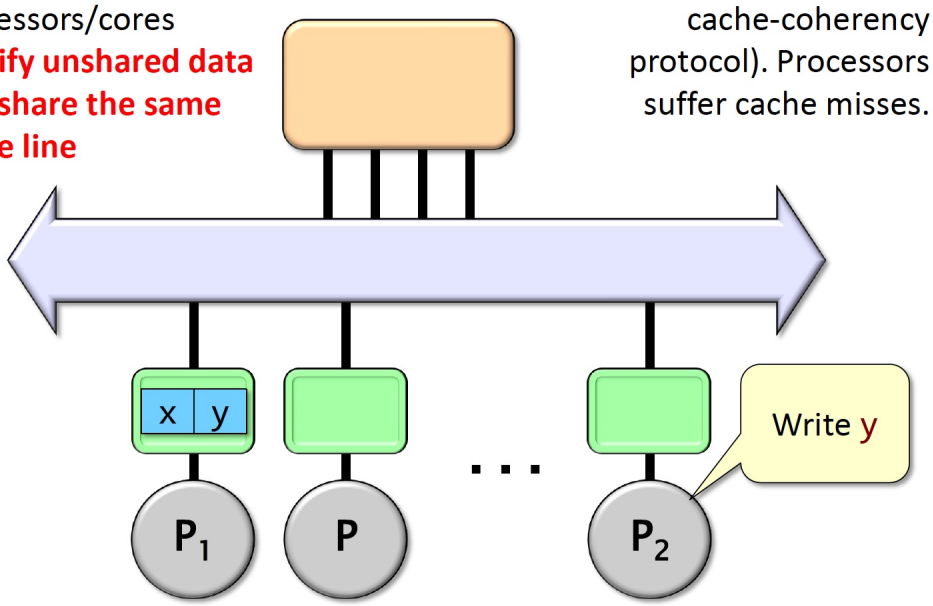






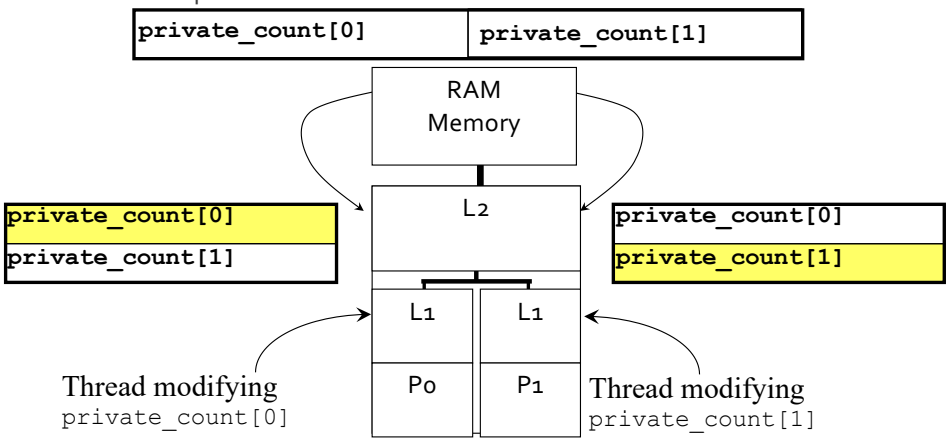
False sharing: threads running on different processors/cores **modify unshared data** that **share the same cache line**

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



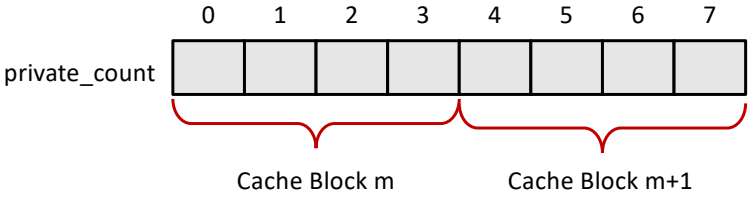
False Sharing

- 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];
```


Force Into Different Lines

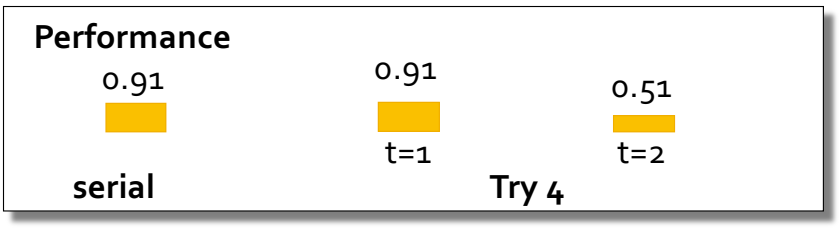
- Padding the private variables forces them into separate cache lines and removes false sharing

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

Try 4

Success!!

- Two processors are almost twice as fast



- Is this the best solution???

Variations

- What happens when more processors are available?
  - 4 processors
  - 8 processors
  - 256 processors
  - 32,768 processors

Parallel Programming Goals

- Goal: Scalable programs with performance and portability
  - Scalable: More processors can be "usefully" added to solve the problem faster
  - Performance: Programs run as fast as those produced by experienced parallel programmers for the specific machine
  - Portability: The solutions run well on all parallel platforms

False Sharing: Solution #1

- 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
}
```

Count 3s Summary

- Recapping the experience of writing the program, we
  - Wrote the obvious "break into blocks" program
  - We needed to protect the count variable
  - We got the right answer, but the program was slower … lock congestion
  - Privatized memory and 1-process was fast enough, 2- processes slow … false sharing
  - Separated private variables to own cache line

Finally, success