COMP4300 - Course Update

- **Quiz 2 released 18 April**
  - Must be completed by Monday 28/04/2025, 11:55PM.
  - Will cover **lectures 7-13**

- **Assignment 2**
  - Will be released on 24 April
  - Due 26/05/2025, 11:55PM
  - Start early

References

- Chapter 12 from Computer Systems A Programmer's Perspective, Third Edition, Randal E. Bryant and David R. O'Hallaron, Pearson Education Heg USA, ISBN 9781292101767.

- Programming with POSIX Threads, David R. Butenhof, Addison-Wesley Professional, ISBN-13 : 978-0201633924.

SHARED MEMORY PARALLEL COMPUTING

THREAD SYNCRONIZATION

DEMO
Pthreads based Concurrency

- Pthreads have low creation overhead
- Pthreads allows rapid switching between threads
- Pthreads can deliver excellent weak and strong scaling
- Pthreads do not require message passing
- Accessing shared data requires careful control

Quick Review

Why threads tend to have higher efficiency than processes in shared-memory parallelization?

What is the main risk of using shared memory address space?

Synchronization Pitfalls

What do you expect the code is going to print when executed on a multiprocessor?

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int shared_counter = 0; // Shared global variable

void* increment_counter(void* arg) {
    for (int i = 0; i < 10000; ++i) {
        // Critical section: Increment the shared counter
        shared_counter++;
    }
    pthread_exit(NULL);
}

int main() {
    pthread_t thread1, thread2;

    // Create two threads
    pthread_create(&thread1, NULL, increment_counter, NULL);
    pthread_create(&thread2, NULL, increment_counter, NULL);

    // Wait for both threads to finish
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    // Print the final value of the shared counter
    printf("Final shared counter value: %d\n", shared_counter);

    return 0;
}
```

Synchronization Pitfalls

What do you expect the code is going to print when executed on a single processor?

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int shared_counter = 0; // Shared global variable

void* increment_counter(void* arg) {
  for (int i = 0; i < 10000; ++i) {
    // Critical section: Increment the shared counter
    shared_counter++;
  }
  pthread_exit(NULL);
}

int main() {
  pthread_t thread1, thread2;

  // Create two threads
  pthread_create(&thread1, NULL, increment_counter, NULL);
  pthread_create(&thread2, NULL, increment_counter, NULL);

  // Wait for both threads to finish
  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);

  // Print the final value of the shared counter
  printf("Final shared counter value: %d\n", shared_counter);

  return 0;
}
```

Synchronization Pitfalls

Results:

```
> Final shared counter value: 14765
> Final shared counter value: 16237
> Final shared counter value: 12583
```

???

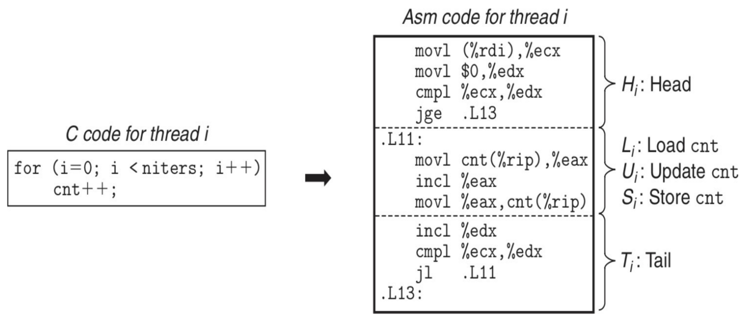
```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int shared_counter = 0; // Shared global variable

void* increment_counter(void* arg) {
  for (int i = 0; i < 10000; ++i) {
    // Critical section: Increment the shared counter
    shared_counter++;
  }
  pthread_exit(NULL);
}

int main() {
  pthread_t thread1, thread2;

  // Create two threads
  pthread_create(&thread1, NULL, increment_counter, NULL);
  pthread_create(&thread2, NULL, increment_counter, NULL);

  // Wait for both threads to finish
  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);

  // Print the final value of the shared counter
  printf("Final shared counter value: %d\n", shared_counter);

  return 0;
}
```

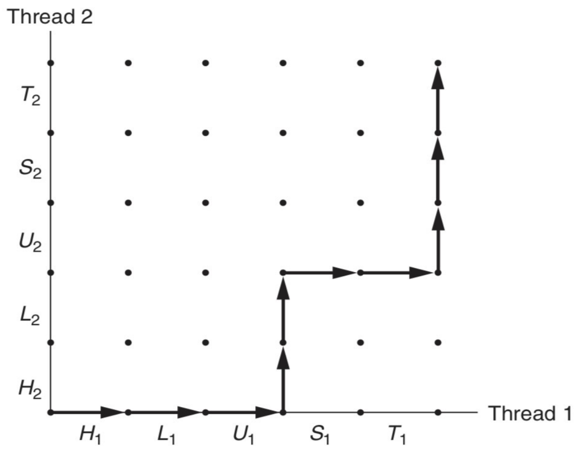
State Diagrams

- A *state diagram* models the execution of *n* concurrent execution flows as a trajectory through an *n*-dimensional Cartesian space (only uniprocessor!)

- For thread *i* the instructions (L_i, U_i, S_i) that manipulate the contents of the shared variable `cnt` constitute a *critical section*

- The instructions of a critical section must be all executed by a single thread at a time.



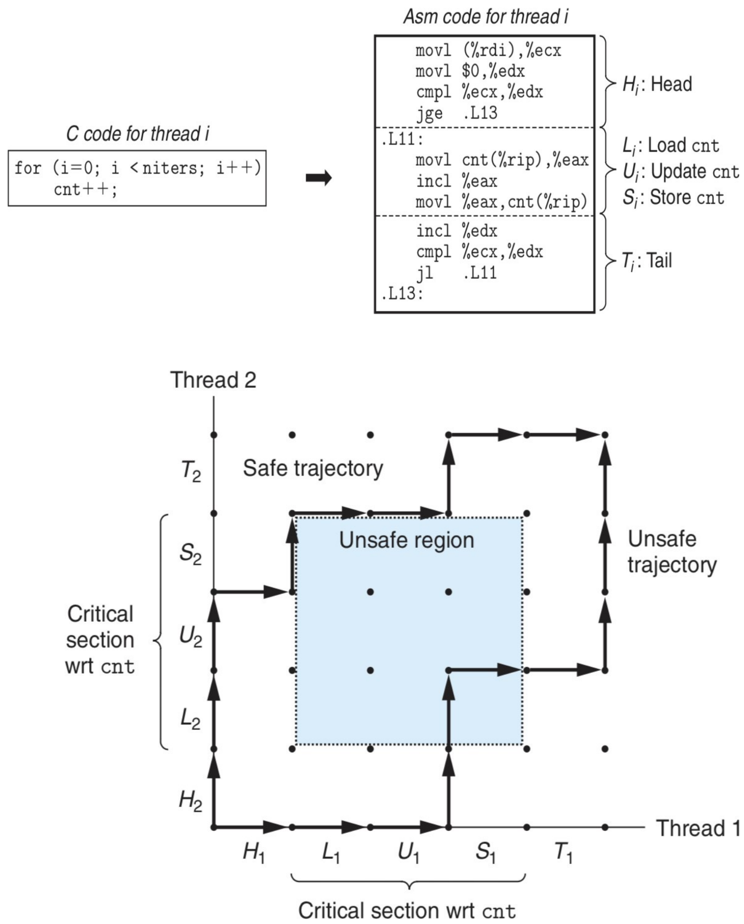


State Diagrams

- In order to obtain correct results, each threads must have *mutually exclusive access* to the shared variable while it is executing instructions in the critical section (*mutual exclusion*).

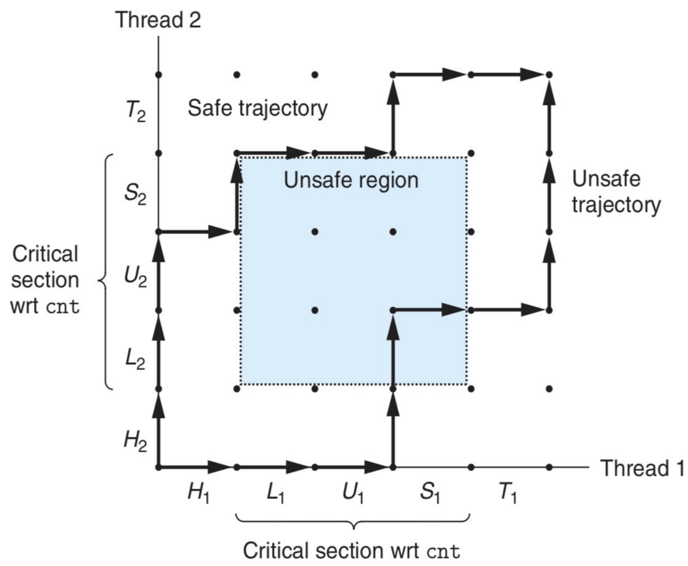
- A trajectory that skirts the *unsafe region* will not cause run time errors (*safe trajectory*).

- In order to guarantee correct execution, we must *synchronize* threads so that they take safe trajectories.



The Critical Section Problem

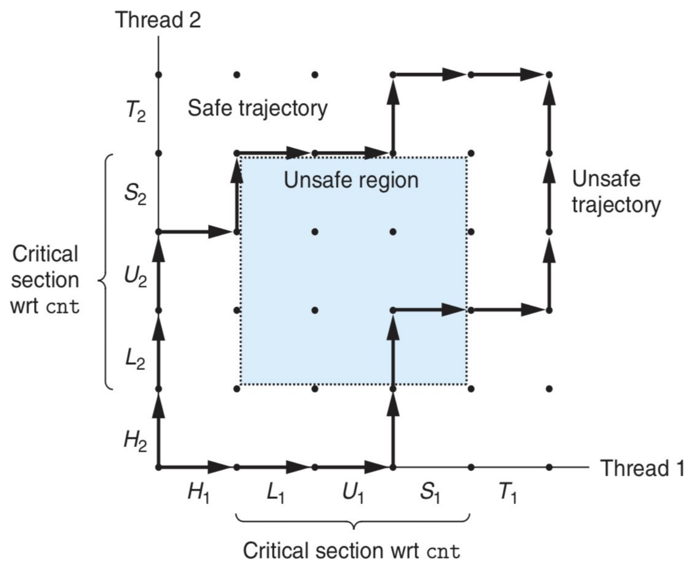
- The critical section problem is a fundamental synchronization problem in computer science and operating systems.
- It arises when multiple concurrent processes or threads share a common resource (such as memory, files, or hardware devices) and need to access it in an exclusive manner.
- The goal is to ensure that only one process can execute its critical section (the part of code that accesses the shared resource) at any given time.



The Critical Section Problem

Requirements:

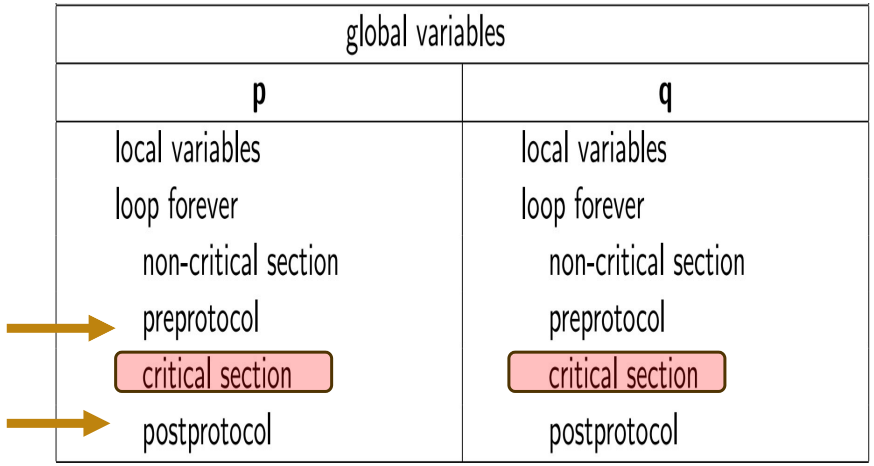
- **Mutual Exclusion**: At most one process can be in its critical section simultaneously.
- **Freedom from deadlock**: If no process is in its critical section and some processes want to enter, one of them should be allowed to enter.
- **Bounded Waiting**: There exists an upper bound on the number of times a process can wait to enter its critical section.



The Critical Section Problem

The synchronization mechanism ensures correctness.

- It uses statements places before and after the critical section, called *preprotocol* and *postprotocol*, respectively.
- **Assumption**: assignment statements are *atomic statements*, as are evaluations of boolean conditions in control statements.
- An ***atomic statement*** is executed to completion without the possibility of interleaving/interrupt from another thread.



Synchronization Pitfalls

Explanation:

- We have a shared global variable `shared_counter`.
- Two threads (thread1 and thread2) increment this counter independently.
- The critical section (increment operation) is not protected by any synchronization mechanism (e.g., mutex or semaphore).
- As a result, a **data race** occurs when both threads simultaneously read and modify `shared_counter`.
- The final value of `shared_counter` is unpredictable due to the race condition.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int shared_counter = 0; // Shared global variable

void* increment_counter(void* arg) {
  for (int i = 0; i < 10000; ++i) {
    // Critical section: Increment the shared counter
    shared_counter++;
  }
  pthread_exit(NULL);
}

int main() {
  pthread_t thread1, thread2;

  // Create two threads
  pthread_create(&thread1, NULL, increment_counter, NULL);
  pthread_create(&thread2, NULL, increment_counter, NULL);

  // Wait for both threads to finish
  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);

  // Print the final value of the shared counter
  printf("Final shared counter value: %d\n", shared_counter);

  return 0;
}
```

Synchronization Pitfalls

- To solve the synchronization pitfall in the previous example, we need to introduce proper synchronization mechanisms to protect the critical section (the shared counter increment).
- Specifically, we'll use a **mutex** (short for mutual exclusion) to ensure that only one thread can access the shared counter at a time.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int shared_counter = 0; // Shared global variable

void* increment_counter(void* arg) {
  for (int i = 0; i < 10000; ++i) {
    // Critical section: Increment the shared counter
    shared_counter++;
  }
  pthread_exit(NULL);
}

int main() {
  pthread_t thread1, thread2;

  // Create two threads
  pthread_create(&thread1, NULL, increment_counter, NULL);
  pthread_create(&thread2, NULL, increment_counter, NULL);

  // Wait for both threads to finish
  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);

  // Print the final value of the shared counter
  printf("Final shared counter value: %d\n", shared_counter);

  return 0;
}
```

Mutexes and Locks with Pthreads

- Specifically, we'll use a **mutex** (short for mutual exclusion) to ensure that only one thread can access the shared counter at a time.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int shared_counter = 0; // Shared global variable
pthread_mutex_t counter_mutex; // Mutex for synchronization

void* increment_counter(void* arg) {
  for (int i = 0; i < 10000; ++i) {
    // Acquire the mutex before accessing the shared counter
    pthread_mutex_lock(&counter_mutex);
    shared_counter++;
    // Release the mutex after modifying the shared counter
    pthread_mutex_unlock(&counter_mutex);
  }
  pthread_exit(NULL);
}
```

Mutexes and Locks with Pthreads

- In main we now initialize the **mutex** (short for mutual exclusion) to ensure that only one thread can access the shared counter at a time

```
> Final shared counter value: 20000
```

Thread Safety and `volatile`:

- While volatile ensures proper reads and writes, it **does not provide thread safety**
- It does not prevent data races or guarantee atomicity
- For synchronization between threads, use mutexes, semaphores, or other synchronization primitives

```
int main() {
    pthread_t thread1, thread2;

    // Initialize the mutex
    pthread_mutex_init(&counter_mutex, NULL);

    // Create two threads
    pthread_create(&thread1, NULL, increment_counter, NULL);
    pthread_create(&thread2, NULL, increment_counter, NULL);

    // Wait for both threads to finish
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    // Clean up: Destroy the mutex
    pthread_mutex_destroy(&counter_mutex);

    // Print the final value of the shared counter
    printf("Final shared counter value: %d\n", shared_counter);

    return 0;
}
```

Synchronization Pitfalls

Why does this code give the incorrect result on a uniprocessor?

- The OS will run threads concurrently: on a uniprocessor instructions will be interleaved.
- Some of the interleaving ordering will produce the correct results, others will not.
- What about a multiprocessor execution?

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int shared_counter = 0; // Shared global variable

void* increment_counter(void* arg) {
  for (int i = 0; i < 10000; ++i) {
    // Critical section: Increment the shared counter
    shared_counter++;
  }
  pthread_exit(NULL);
}

int main() {
  pthread_t thread1, thread2;

  // Create two threads
  pthread_create(&thread1, NULL, increment_counter, NULL);
  pthread_create(&thread2, NULL, increment_counter, NULL);

  // Wait for both threads to finish
  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);

  // Print the final value of the shared counter
  printf("Final shared counter value: %d\n", shared_counter);

  return 0;
}
```

Example Solution: Dekker's Algorithm

- The variables `wantp` and `wantq` ensure mutual exclusion.
- Suppose p detects contention by finding `wantp == true` (p3): it will consult the shared variable `turn`, to check whether it is its turn (`turn` == 1) to insist on entering its critical section.
- If so, it executes the loop at p3 and p4, called a busy-wait loop, until q resets `wantq` to `false`, either by terminating its critical section at q10 or by deferring in q5.
- If not, p will reset `wantp` to false and defer to thread q, waiting until q changes the value of turn after executing its critical section.

Algorithm 3.10: Dekker's algorithm

```
boolean wantp ← false, wantq ← false
integer turn ← 1
```

p	q
loop forever	loop forever
p1: non-critical section	q1: non-critical section
p2: wantp ← true	q2: wantq ← true
p3: while wantq	q3: while wantp
p4: if turn = 2	q4: if turn = 1
p5: wantp ← false	q5: wantq ← false
p6: await turn = 1	q6: await turn = 2
p7: wantp ← true	q7: wantq ← true
p8: critical section	q8: critical section
p9: turn ← 2	q9: turn ← 1
p10: wantp ← false	q10: wantq ← false

Semaphores

A **semaphore** ensures that only one process can access the shared variable at a time.

We create a semaphore using `semget` and initialize it with an initial value (e.g., 1).

Both the parent and child processes use `sem_wait` to wait for the semaphore before entering the critical section.

```
#include <sys/sem.h>
#include <sys/ipc.h>

// Define a semaphore
int semid;

// Initialize the semaphore
void initSemaphore() {
   key_t key = 1234; // Unique key for the semaphore
   int nsems = 1;    // Number of semaphores in the set
   semid = semget(key, nsems, IPC_CREAT | 0666);
   if (semid < 0) {
      perror("Semaphore creation failed");
      exit(1);
   }
   // Set the initial value of the semaphore (e.g., 1)
   semctl(semid, 0, SETVAL, 1);
}

// Perform the critical section operation
void criticalSection(int* sharedVar) {
   (*sharedVar)++;
   printf("Shared variable value: %d\n", *sharedVar);
}
```

Semaphores

In this example, we create two processes (parent and child) that share a common variable using a semaphore.

Both the parent and child processes use `sem_wait` to wait for the semaphore before entering the critical section.

After performing the critical section operation (incrementing the shared variable), they release the semaphore using `sem_post`.

```
int main() {
  int sharedVar = 0;

  // Initialize the semaphore
  initSemaphore();

  int pid = fork();
  if (pid < 0) {
    perror("Fork failed");
    exit(1);
  } else if (pid == 0) {
    // Child process
    sem_wait(&semid); // Wait for the semaphore
    printf("Child process entered critical section\n");
    criticalSection(&sharedVar);
    sem_post(&semid); // Release the semaphore
  } else {
    // Parent process
    sem_wait(&semid); // Wait for the semaphore
    printf("Parent process entered critical section\n");
    criticalSection(&sharedVar);
    sem_post(&semid); // Release the semaphore
  }

  return 0;
}
```

Semaphores

Signaling Mechanism:

- Semaphores work based on signaling.
- Two fundamental atomic operations:
  - **Wait (P)**: Decrements the semaphore value. If the value becomes negative, the calling thread waits (blocks).
  - **Signal (V)**: Increments the semaphore value. If any threads were waiting, one of them is unblocked.

Advantages:

- Multiple threads can access the critical section simultaneously (controlled by the semaphore value).
- Semaphores are machine-independent.
- Allows a specified number of processes to enter (useful for limiting resources).

Common Use Cases:

- Controlling access to a pool of resources (e.g., limiting the number of concurrent database connections).
- Implementing producer-consumer patterns.
- Coordinating multiple threads or processes.

Demo
Race Condition & Synchronisation

- Three C programs to illustrate managing the race condition shown previously.
  - A version in which no synchronization mechanisms are used
  - An alternative version using sempahores
  - An alternative version using mutexes

- **Each program**
  - Creates two threads that each increment a shared counter 10,000 times
  - Final counter value is printed
  - Measures execution time for both individual threads and the overall program

Demo
Race Condition & Synchronisation

- Race conditions produce inconsistent results.
- While *semaphores* offer greater flexibility and can be used for a variety of synchronization tasks, they come with additional overhead that can make them slower compared to *mutexes* for simple mutual exclusion scenarios.
- For tasks that require only mutual exclusion (binary), mutexes are generally the preferred and more efficient choice.

OpenMP: Part I

OpenMP Reference Material

- **Using OpenMP – The Next Step**, R. van der Pas, E. Stotzer, and C. Terboven, Chapter 1
- http://www.openmp.org/
- *Introduction to High Performance Computing for Scientists and Engineers*, Hager and Wellein, Chapter 6 & 7
- *High Performance Computing*, Dowd and Severance, Chapter 11
- *Introduction to Parallel Computing, 2nd Ed*, A. Grama, A. Gupta, G. Karypis, V. Kumar
- *Parallel Programming in OpenMP*, R. Chandra, L.Dagum, D.Kohr, D.Maydan. J.McDonald, R.Menon

History of Concurrent Programming

Fork

- The concept of "fork" originated in Unix operating systems.
- Forking was a fundamental mechanism for multitasking in Unix.
- The fork() system call is used to create a new process by duplicating the existing process.

Threads

- As computing needs grew, the limitations of process-based concurrency became apparent, particularly the overhead associated with creating and managing processes.
- Threads enabled finer-grained parallelism and improved performance for applications requiring concurrent execution of tasks.

OpenMP

- OpenMP (Open Multi-Processing) is an API that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran.
- OpenMP uses compiler directives, library routines, and environment variables to control parallelism.

Shared Memory Parallel Programming

- Explicit thread programming is messy
  - low-level primitives
  - complex data scoping and initialization not easy to port
  - significant amount of boiler-plate code
  - used by system programmers, but …. application programmers have OpenMP!
- Many application codes can be supported by higher level constructs with the same performance
  - led to proprietary directive based approaches of Cray, SGI, Sun, etc.
- OpenMP is an API for shared memory parallel programming targeting Fortran, C and C++
  - standardizes the form of the proprietary directives
  - avoids the need for explicitly setting up mutexes, condition variables, data scope, and a good part of explicit initialization

OpenMP

- Specifications maintained by OpenMP Architecture Review Board (ARB)
  - members include AMD, Intel, Fujitsu, IBM, NVIDIA
- Versions 1.0 (Fortran ’97, C ’98), 1.1 and 2.0 (Fortran ’00, C/C++ ’02), 2.5 (unified Fortran and C, 2005), 3.0 (2008), 3.1 (2011), 4.0 (2013), 4.5 (2015), 5.0 (2018), 6.0 (2024)
- Comprises compiler directives, library routines and environment variables
  - C directives (case sensitive)

    ```
    #pragma omp directive_name [clause-list]
    ```
  - library calls begin with omp_

    ```
    void omp_set_num_threads(nthreads)
    ```
  - environment variables begin with OMP_

    ```
    export OMP_NUM_THREADS=4
    ```
- OpenMP requires compiler support
  - set `-fopenmp` (gcc) or `-qopenmp` (icc) compiler flags

The `Parallel` Directive

- OpenMP uses a fork/join model, i.e. programs execute serially until they encounter a parallel directive:
  - this creates a group of threads
  - the number of threads dependent on an environment variable or set via function call
  - the main thread becomes master with thread id 0

```
#pragma omp parallel [ clause - list ]
    /* structured block */
```

- Each thread executes the *structured block*
- In C/C++ this is a brace-enclosed (`{ code }`) sequence of statements and declarations.

The OpenMP Execution Model

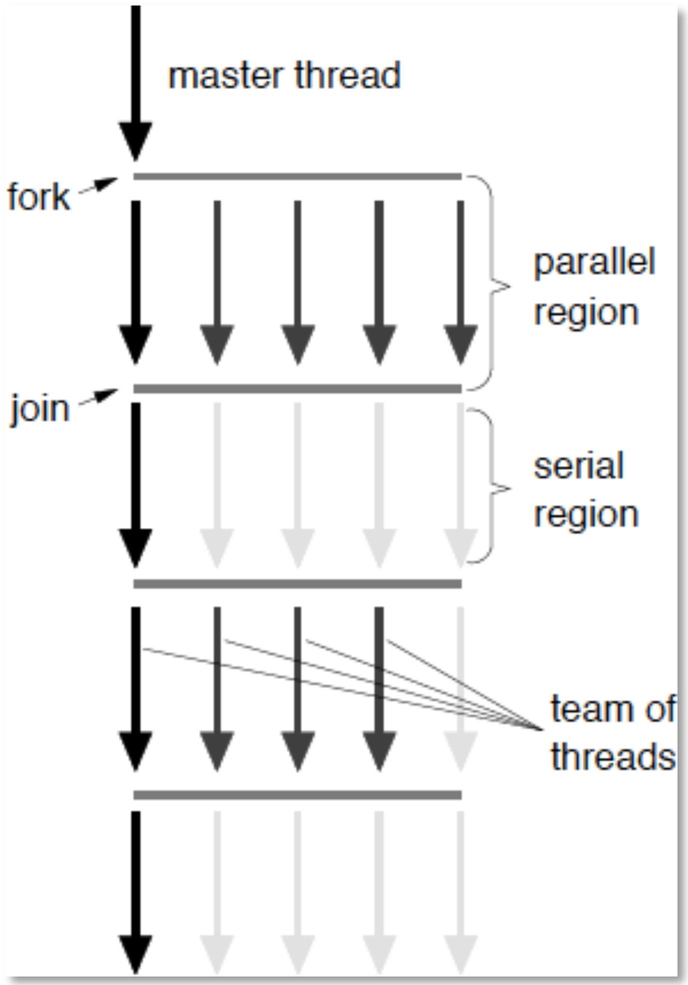
- An OpenMP starts in serial mode with one thread executing the serial code (*master thread*)

- At the beginning of the parallel region additional threads are created (***forking*** from the master) by the runtime system forming a *thread team*

- All threads are active in the parallel regions, executing the program in parallel.

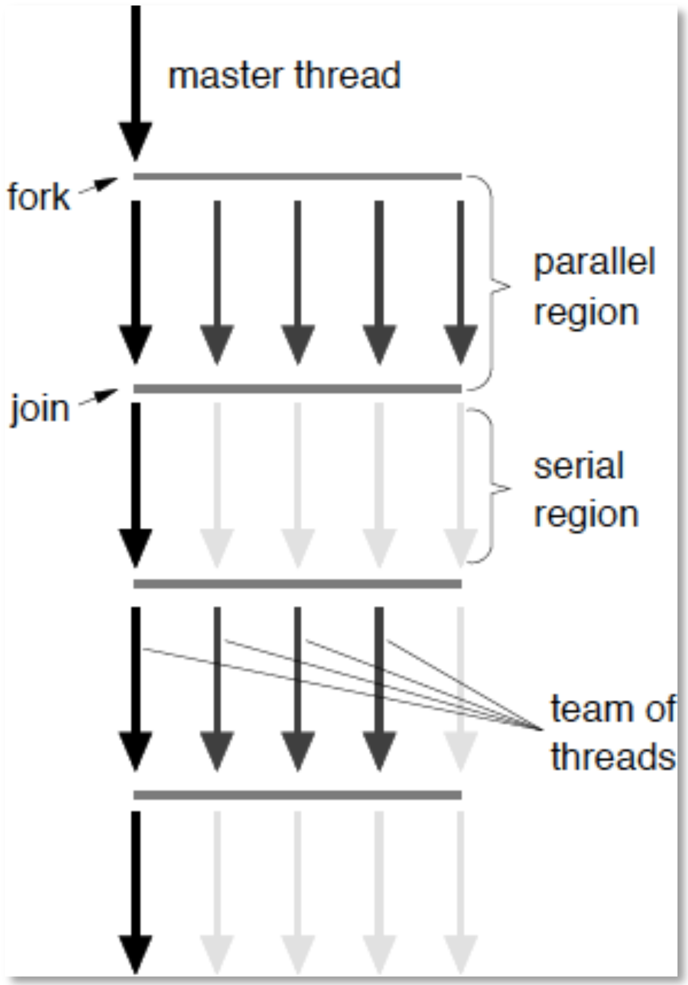
- At the end of the parallel region threads are ***joined***, with only the master continuing through the serial portion.

- **This is called the *fork-join* model.**



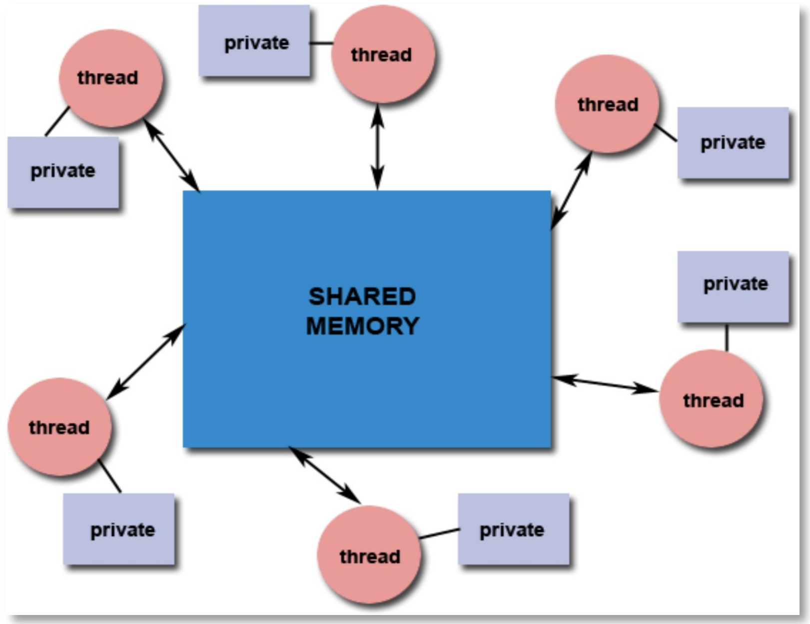
The OpenMP Execution Model

- The number of threads execution in the parallel region can be set through the `OMP_NUM_THREADS` environment variable.
- If the number of threads need to be more dynamic, the `omp_set_num_threads` may be used *prior* to a parallel region.
- An alternative is to use the `num_threads<nt>` clause on the `parallel` directive.
- Because of the join operation, the end of the parallel region is an *implicit synchronization point* (barrier).



The OpenMP Memory Model

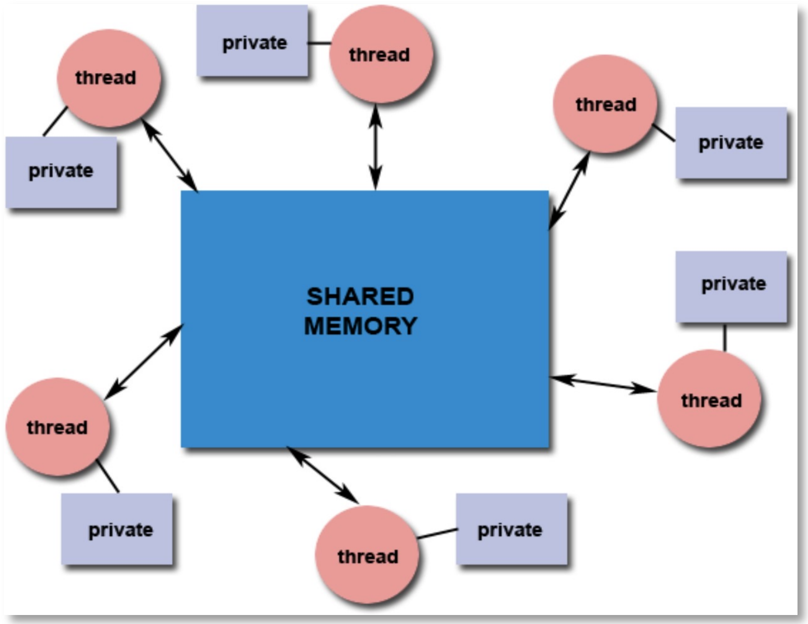
- Underlying the OpenMP standard is the `pthreads` memory model, but the distinction between *private* and *shared* is clearer.
- Whether a variable is private or shared as well as their initialization can be defined by *default rules*
- These can also be explicitly controlled through appropriate clauses on a construct.
- It is recommended to <u>not</u> rely on the default rules and explicitly label or "scope" variables.



The OpenMP Memory Model

Private and Shared Variables

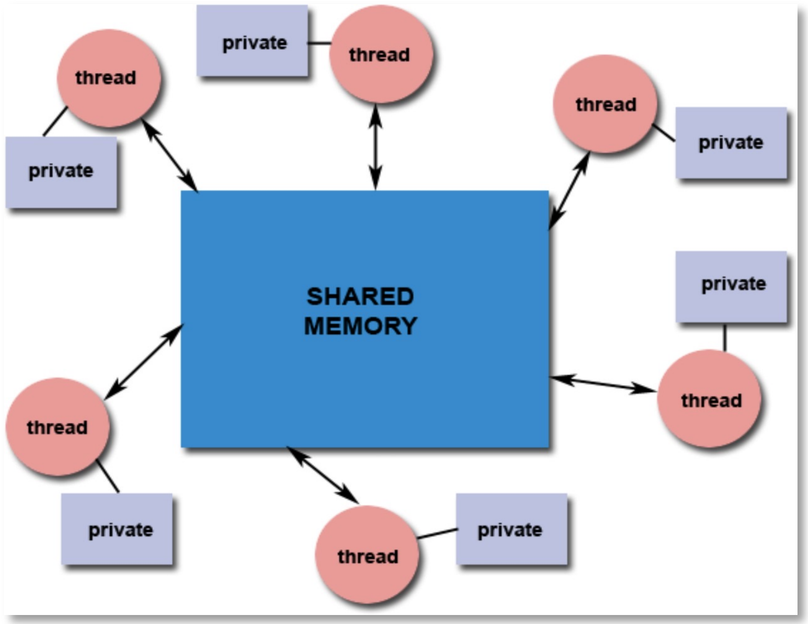
- Private variables can be accessed only by the owning thread, no other thread may interfere.
- Threads may even use the same name for a private variable without the risk of any conflict.
- Each thread has read and write access to the same shared variable, that is only one instance of a given shared variable exists.
- Global or static variables are shared by default.



The OpenMP Memory Model

Default Rules

- Variables declared outside the parallel region are shared by default.
- Global and static variables are also shared by default.
- Variables declared inside the parallel region are private by default.



Data Sharing Clauses

- The `private (shared)` clause may be used to make a variable thread private (shared).
- One needs to be careful with initialization.
- The `firstprivate` clause guarantees that all threads have a pre-initialized copy of a variable
- The `default` clause is used to give a default data sharing attribute (`none, shared, private`) to all variables.
- When `default(none)` is used, the programmer is forced to specify data-sharing attributes for all variables in the construct.

```
int x = 5; int y = 20;
int z[ NUM_THREADS ] = {0};
# pragma omp parallel default ( none ) shared ( z)
   private ( x) firstprivate ( y)
{
 x = 10; // x is undefined on entry , but now set
    to 10
 z[ omp_get_thread_num () ] = omp_get_thread_num ();
 int w = x + y+ z[ omp_get_thread_num () ]; // y pre
    - initialized to a value of 20
 ...
 y = 30 // firstprivate var may be modified
}
```

Data Sharing Attributes

Default Rules

- Variables declared outside the parallel region are **shared** by default.
- Global and static variables are also **shared** by default.
- Variables declared inside the parallel region are **private** by default.

```c
int g = 0; // g is shared
int main (){
 int i = 0; // i is shared
 static int a = 7; // a is shared
 # pragma omp parallel
 {
  int b = a + i + g; // b is private
  ...
 }
return 0;
}
```

OpenMP:The Work-Sharing Directives

- Used to distribute work among threads in a team.
- They specify the way the work has to be distributed among threads.
- Work-sharing directive must bind to a parallel region, otherwise is simply ignored.
- Work-sharing constructs do not have a barrier at entry.
- By default, a barrier is implemented at the end of the work-sharing region. The programmer can suppress the barrier with use of the nowaitclause.

Functionality	Syntax in C/C++	Syntax in Fortran
Distribute iterations over the threads	**#pragma omp for**	**!$omp do**
Distribute independent work units	**#pragma omp sections**	**!$omp sections**
Only one thread executes the code block	**#pragma omp single**	**!$omp single**
Parallelize array-syntax		**!$omp workshare**

The `for` Work-Sharing Directives

Used in conjunction with `parallel` directive to partition the `for` loop immediately afterwards

- The loop index (i) is made private by default
- Only two directives plus the sequential code (code is easy to read/maintain)
- Limited to loops where number of iterations can be counted

```
# pragma omp parallel shared ( n)
{
# pragma omp for
for ( i = 0; i < n; i ++) {
    printf (" Thread % d , iteration % d\ n",
               omp_get_thread_num , i);
    }
} /* End of parallel region */
```

There is implicit synchronization at the end of the loop

- Can add a `nowait` clause to prevent synchronization

The `for` Work-Sharing Directives

- The order in which threads execute is not predictable (OS scheduled).
- The way to map iterations to threads can be specified by the programmer (see later `schedule` clause).
- If the programmer does not specify the mapping between threads and iterations, the compiler decides which strategy to use.

```
Thread 0 executes loop iteration 0
Thread 0 executes loop iteration 1
Thread 0 executes loop iteration 2
Thread 3 executes loop iteration 7
Thread 3 executes loop iteration 8
Thread 2 executes loop iteration 5
Thread 2 executes loop iteration 6
Thread 1 executes loop iteration 3
Thread 1 executes loop iteration 4
```

The `sections` Work-Sharing Directives

- Consider partitioning of fixed number of tasks across threads
- Each section must be a structured block that is independent from the other sections.
- Separate threads will run `taskA` and `taskB`
- Illegal to branch in or out of section blocks

Note:

- Much less common than `for` loop partitioning
- Explicit programming naturally limits number of threads (scalability)
- Potential load imbalance

```
# pragma omp parallel
{
  # pragma omp sections
  {
    # pragma omp section
          task A ()

    # pragma omp section
          task B ()

  } /* End of sections block */

} /* End of parallel region */
```

The `single` Work-Sharing Directives

- This directive specifies that only one thread must execute the code in the structured block following it.
- It does not state which thread should execute the code.

```
# pragma omp parallel shared ( a, b, n)
{
  # pragma omp single
  {
     a = 10;
  } /* A barrier is automatically inserted here */
  # pragma omp for
  for ( i = 0; i < n; ++i)
  {
    b[ i] = a;
  }/* Another barrier is automatically inserted here */
}
```

Combined `parallel` Work-Sharing Directives

- When there is only one work-sharing directive it can be combined with the `parallel` one to improve readability.
- Only clauses that are allowed by both the `parallel` and the specific work-share directive are allowed, otherwise the code is illegal.
- The compiler may optimize code further (*e.g.* remove redundant barriers).

Full version	Combined construct

#pragma omp parallel
{
  #pragma omp for
    `for-loop`
}
	#pragma omp parallel for
	  `for-loop`

#pragma omp parallel
{
#pragma omp sections
{
  [**#pragma omp section**]
    structured block
  [**#pragma omp section**
    structured block]
  . . .
}
}
	#pragma omp parallel sections
	{
	  [**#pragma omp section**]
	    structured block
	  [**#pragma omp section**
	    structured block]
	  . . .
	}