PARALLEL PROCESSING

Mohammed Alabdulkareem

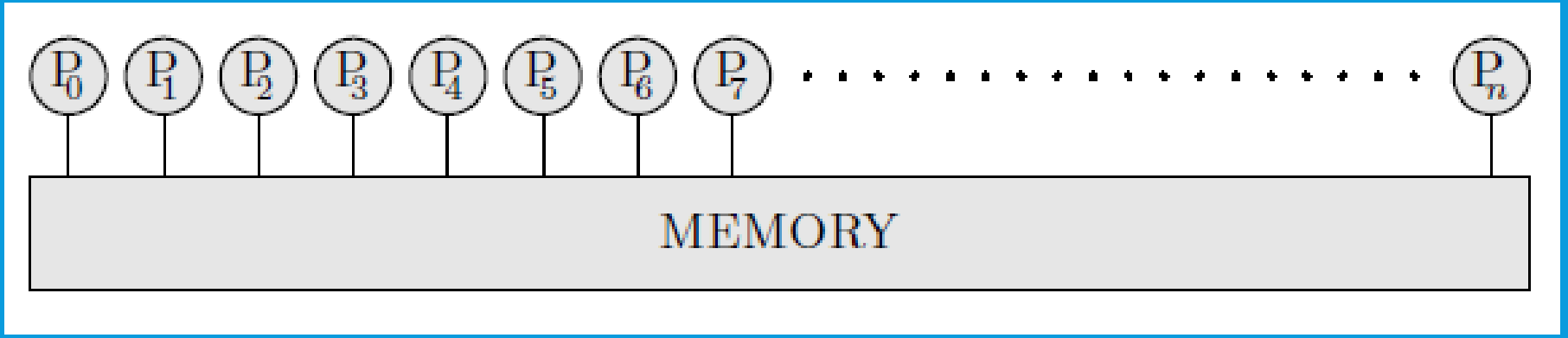
kareem@ksu.edu.sa

Office 2247

10

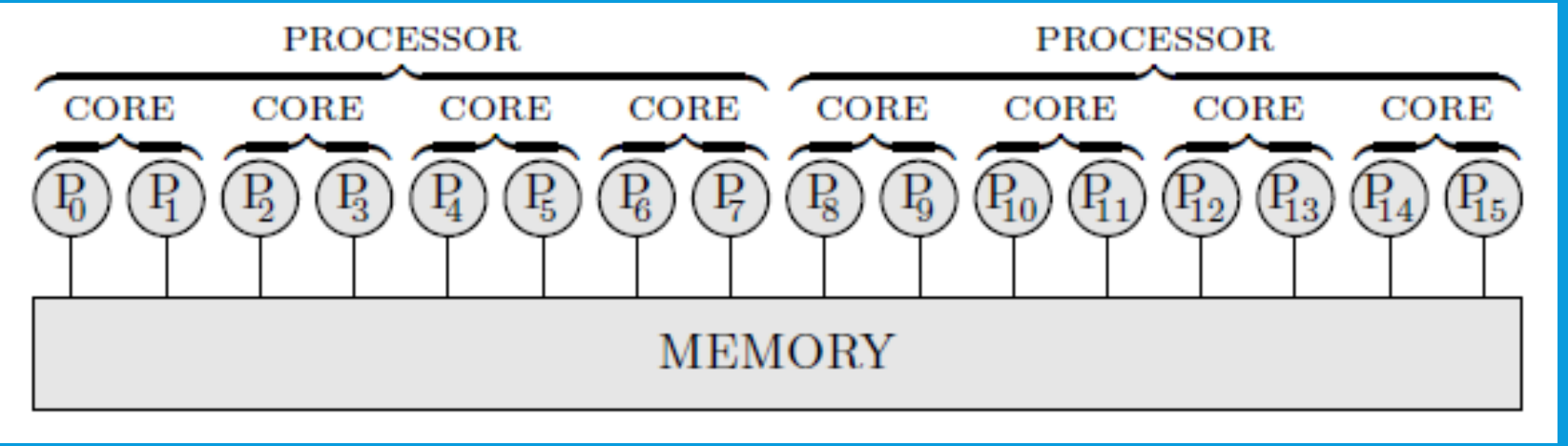
PROGRAMMING SHARED MEMORY SYSTEMS

- From the programmer's point of view, a model of a shared memory multiprocessor contains a number of independent processors all sharing a single main memory.



PROGRAMMING SHARED MEMORY SYSTEMS

- Most modern CPUs are multi-core processors that consists of a number of independent processing units called cores.
- Each core execute multiple independent streams of instructions called threads (logical core).
- Programmer can assume that such system contains 16 logical cores each acting as an individual processor.

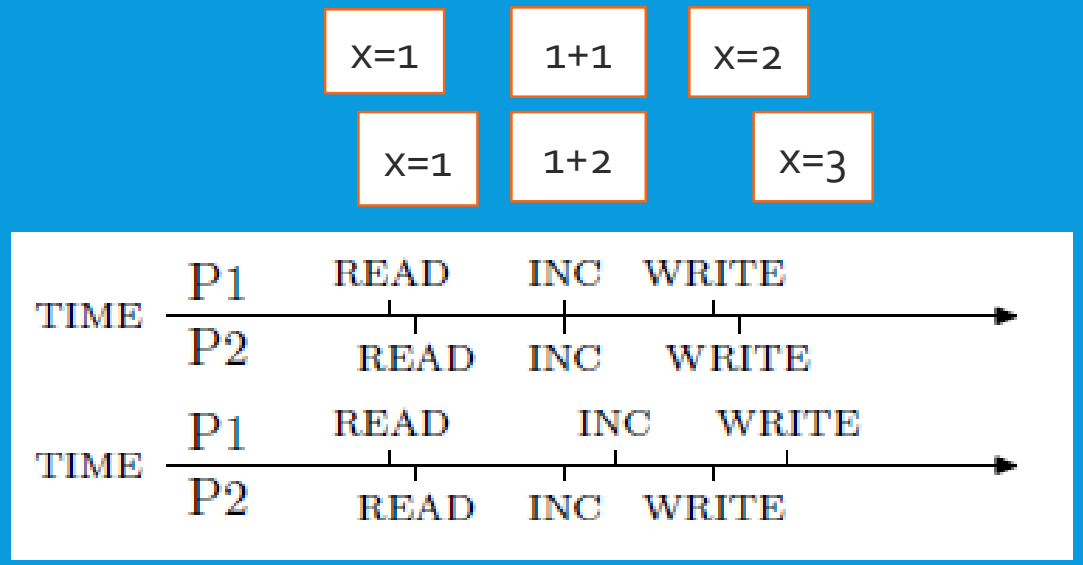


PROGRAMMING SHARED MEMORY SYSTEMS

- As individual threads can access any memory location in the main memory, but executing instruction streams independently may result in a race condition.
- Since threads execute instructions independently, the sequences of instructions executed by each thread may overlap in time.

PROGRAMMING SHARED MEMORY SYSTEMS

- Assume, for instance, that two threads must increase the value stored at the same memory location (**x=1**), one by **1** and the other by **2** so that in the end **x=x+3**.

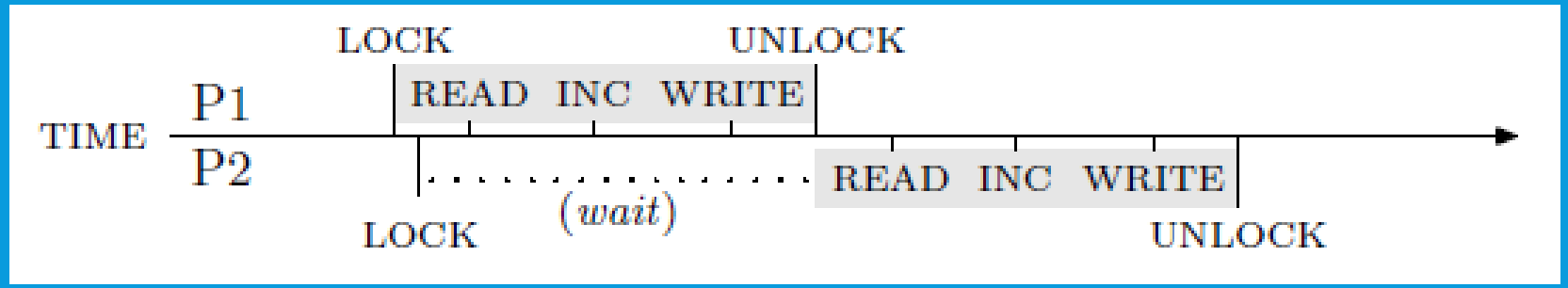


3 ✖

2 ✖

PROGRAMMING SHARED MEMORY SYSTEMS

- To avoid the race condition, exclusive access to the shared address in the main memory must be ensured using some mechanism like locking.
- Each thread must lock the access to the shared memory location before modifying it and unlock it afterwards.



PROGRAMMING SHARED MEMORY SYSTEMS

- On a shared memory multiprocessor a parallel program usually consists of multiple threads.
- The number of threads may vary during execution, but the each thread is running on one logical core.
- If number of threads is less than number of logical cores some logical cores are kept idle.
- Keeping some logical cores idle reduce utilization. (less efficiency).

PROGRAMMING SHARED MEMORY SYSTEMS

- When the number of threads is more than logical cores, the operating system applies multitasking among threads running on the same logical cores.
- During program execution the operating system may perform load balancing by migrating threads from one logical core to another in an attempt to keep all logical cores equally utilized.

PARALLEL PROGRAMMING ENVIRONMENT

- OpenMP is a parallel programming environment suitable for writing parallel programs to be run on shared memory systems.

- OpenMP is not a programming language but an add-on to an existing languages like C, C++, and Fortran.

PARALLEL PROGRAMMING ENVIRONMENT

- The application programming interface (API) of OpenMP is a collection of
  - Compiler directives: tell the compiler about the parallelism in the source code and provide instructions for generating the parallel code.
  - Supporting functions: that enable programmers to exploit and control the parallelism during the execution of a program.
  - Shell variables: permit tuning of compiled programs to a particular parallel system.

PARALLEL PROGRAMMING ENVIRONMENT

- Compiler directives: tell the compiler about the parallelism in the source code and provide instructions for generating the parallel code.
  - In **C** code it is written as ***#pragma omp <…>***

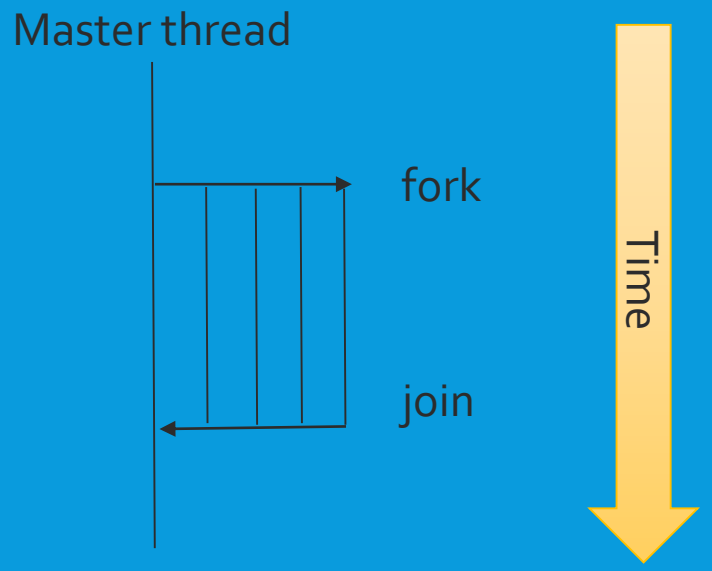
  A parallel region within a program is specified as

  #pragma omp parallel [clause [[,] clause] . . .]

  structured-block

PARALLEL PROGRAMMING ENVIRONMENT

A team of threads is formed and the thread that encountered the ***omp parallel*** directive becomes the **master thread** within this team.



PARALLEL PROGRAMMING ENVIRONMENT

- **Supporting functions:** that enable programmers to exploit and control the parallelism during the execution of a program.
- Example:
  - To set the number of threads:

    void omp_set_num_threads( )

  - To get the maximal number of threads available

    int omp_get_max_threads()

PARALLEL PROGRAMMING ENVIRONMENT

- **Shell variables:** permit tuning of compiled programs to a particular parallel system.
- Example:

  OMP_NUM_THREADS=8

PARALLEL PROGRAMMING ENVIRONMENT

To compile and run the program named hello-world.c using GNU GCC C/C++ compiler, use the command-line option -fopenmp as follows:

```
$ gcc -fopenmp -o hello-world hello-world.c
$ env OMP_NUM_THREADS=8 ./hello-world
```

When the number of threads in the program was not specified explicitly the number of threads will match the shell variable OMP_NUM_THREADS

PARALLEL PROGRAMMING ENVIRONMENT


```
# include <stdio .h>
# include <omp.h>

int main () {
printf ("Hello , world :");
# pragma omp parallel
printf (" %d", omp_get_thread_num ());
printf ("\n");
return o;
 }
```

```
Hello, world: 2 1 0 7 6 5 3 4
```

PARALLEL PROGRAMMING ENVIRONMENT

- If the shell variable OMP_NUM_THREADS was not set then the program would set the number of threads to match the number of logical cores threads can run on.
- Using shell variable is useful when you need multiple runs with different number of thread for each run without recompiling the code.
- When running the code without specifying the number of threads the program will use the number of logical cores available.

PARALLEL PROGRAMMING ENVIRONMENT

Another way to set the number of threads is using the run time function

```
omp_set_num_threads(8);
```

This will set the number of threads to 8.

Or using the parallel clause to request a certain number of threads

```
#pragma omp parallel num_threads(8)
```

PARALLEL PROGRAMMING ENVIRONMENT

The parallel region is specified by the clause:

#pragma omp parallel

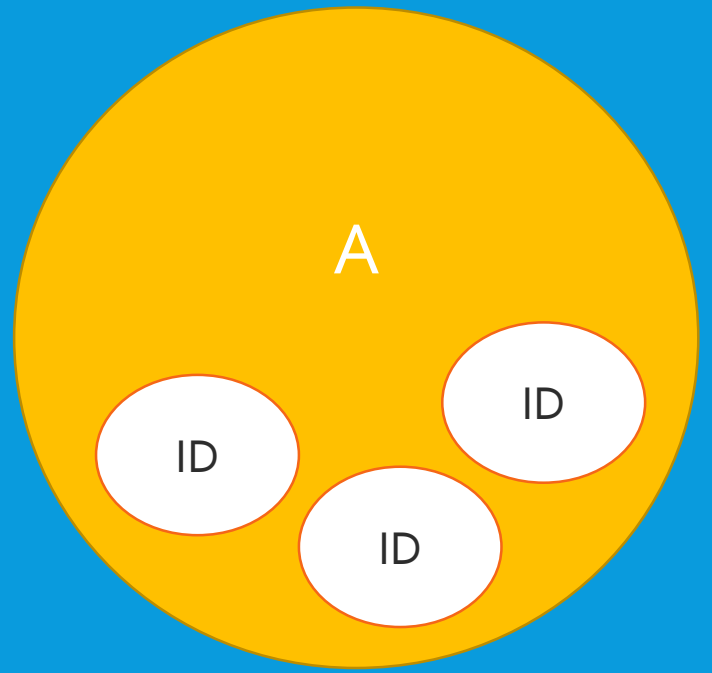
```
#include “omp.h”
int main()
{
#pragma omp parallel
{
int ID = omp_get_thread_num();
printf(“ hello(%d) ”, ID);;
printf(“ world(%d) \n”, ID);;
}
}
```

PARALLEL PROGRAMMING ENVIRONMENT

جامعة الملك سعود
King Saud University
1957

Private and global variables

```
double A[1000];
omp_set_num_threads(8);
#pragma omp parallel
{
int ID = omp_get_thread_num();
foo(ID,A);
}
```

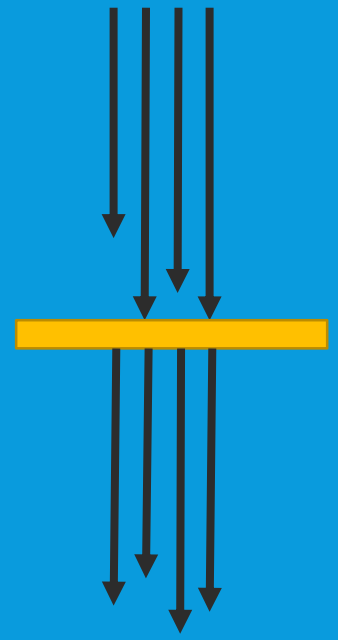


PARALLEL PROGRAMMING ENVIRONMENT

Synchronization

```
#pragma omp barrier
```

```
#pragma omp parallel
{
int id=omp_get_thread_num();
A[id] = big_calc1(id);
#pragma omp barrier
B[id] = big_calc2(id, A);
}
```



PARALLEL PROGRAMMING ENVIRONMENT

Parallel for loop: example of adding 2 vectors

```
        # pragma omp parallel for
```

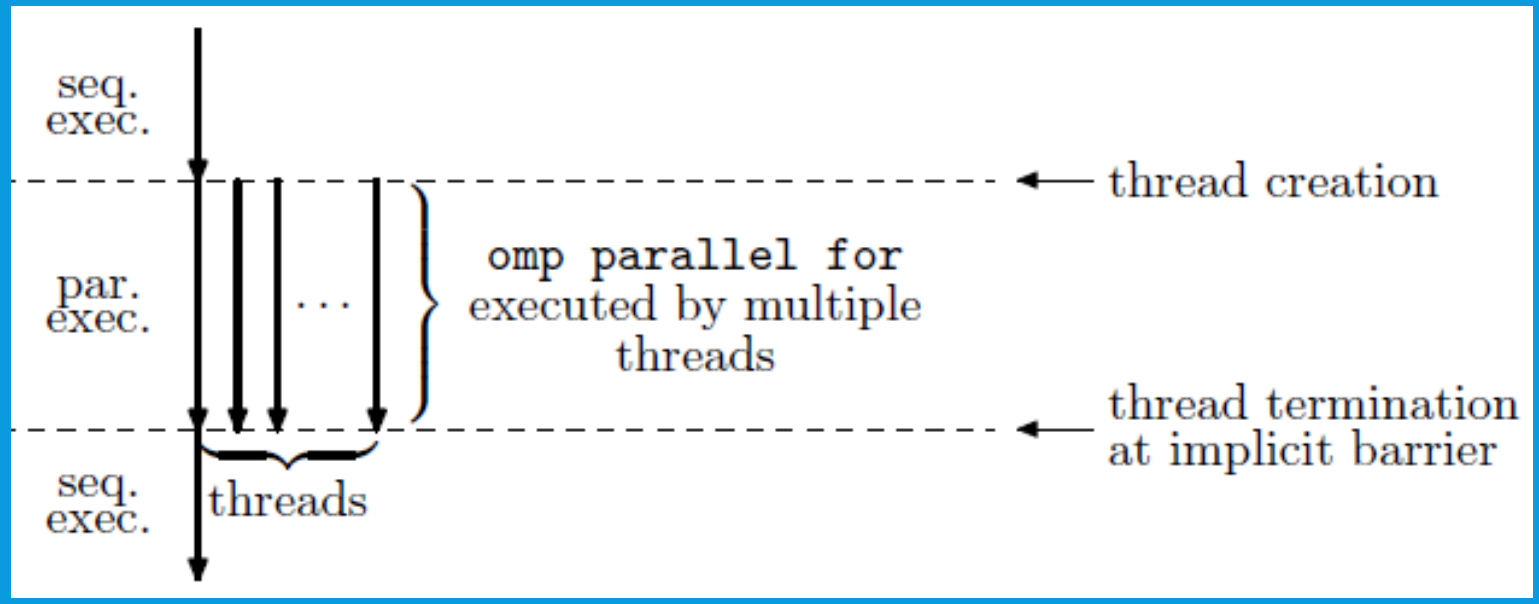
```
double * vectAdd ( double *c, double *a, double *b, int n)
{
# pragma omp parallel for
        for ( int i = 0; i < n; i++)
        c[i] = a[i] + b[i];
 return c;
}
```

PARALLEL PROGRAMMING ENVIRONMENT

Parallel for loop variable *i* is made private in each thread executing a chunk of iterations as each thread must have its own copy of *i*.

Note that different iterations access different array elements. Since they read from and write to completely different memory locations. Hence, no race conditions can occur.

PARALLEL PROGRAMMING ENVIRONMENT



PARALLEL PROGRAMMING ENVIRONMENT

Parallel for loops in matrix multiplication:

```
double **   mtxMul ( double **c, double **a, double **b, int n)
{
# pragma omp parallel for collapse (2)
for ( int i = 0; i < n; i++)
            for ( int j = 0; j < n; j++)
            {
            c[i][j] = 0.0;
             for ( int k = 0; k < n; k++)
                        c[i][j] = c[i][j] + a[i][k] * b[k][j];
            }
return c;
}
```

PARALLEL PROGRAMMING ENVIRONMENT

Parallel for loops in matrix multiplication:

```
double ** mtxMul ( double **c, double **a, double **b, int n)
 {
# pragma omp parallel for
for ( int i = 0; i < n; i++)
            # pragma omp parallel for
            for ( int j = 0; j < n; j++)
             {
                        c[i][j] = 0.0;
                        for ( int k = 0; k < n; k++)
                              c[i][j] = c[i][j] + a[i][k] * b[k][j];
             }
return c;
 }
```

PARALLEL PROGRAMMING ENVIRONMENT