Programming with Shared Memory

Part 1

Threads
Accessing shared data
Critical sections

Last update: 29/11/2021

Shared memory multiprocessor system

Any memory location can be accessible by any of the processors.

A *single address space* exists, meaning that each memory location is given a unique address within a single range of addresses.

Programming a shared memory multiprocessor system can take advantage of data stored in the shared memory, which is accessible by all processors without having to send the data to destination through message passing

Shared Memory Programming

Generally, shared memory programming more convenient although it does require access to shared data by different processors to be **carefully** controlled by the programmer.

Shared memory systems have been around for a long time but with the advent of multi-core systems, it has become very important to able to program for them

Methods for Programming Shared Memory Multiprocessors

• Using heavyweight processes.

• Using threads. Example Pthreads, Java threads

• Using a completely new programming language for parallel programming - not popular. Example **Ada**.

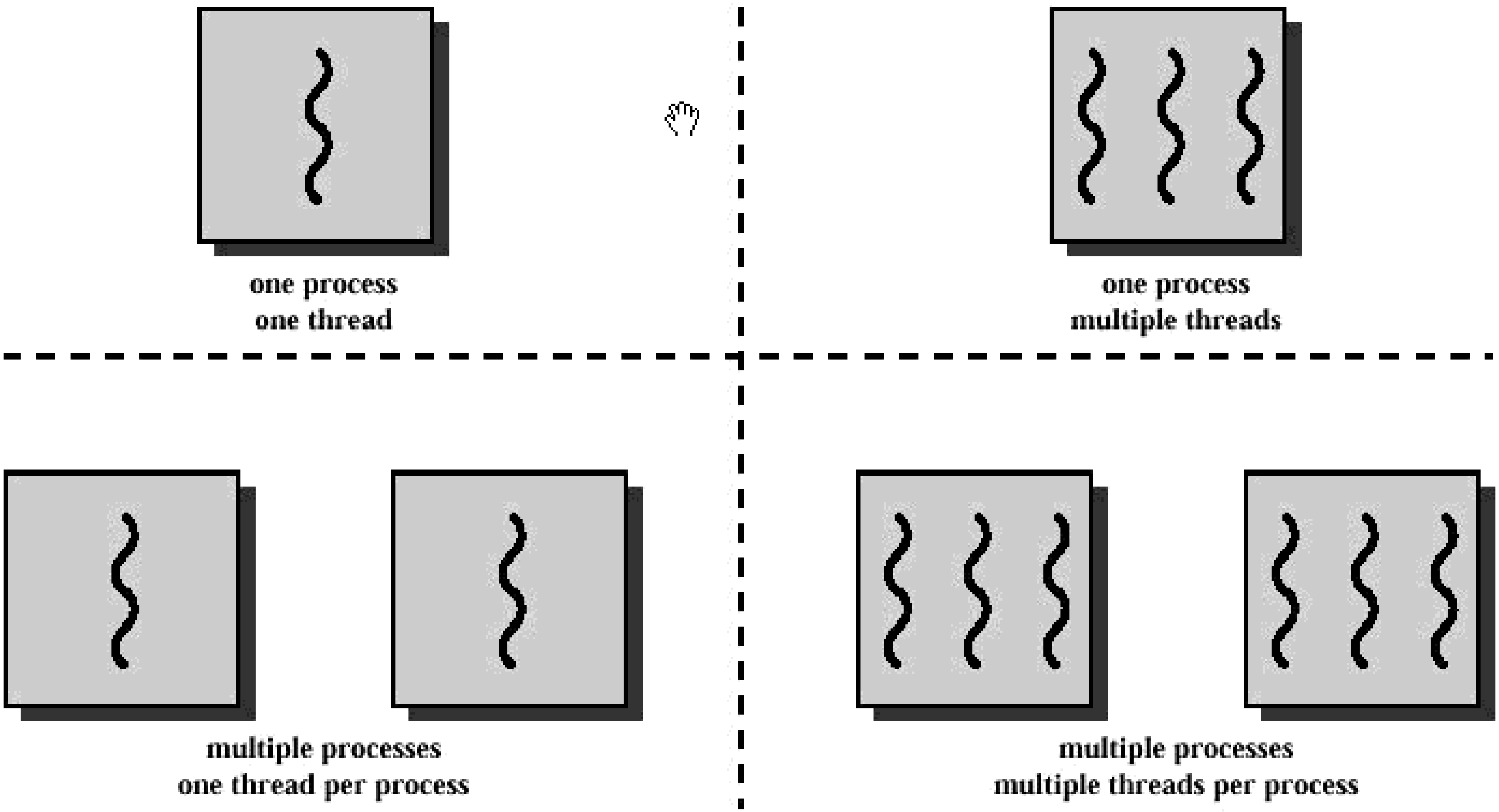
• Modifying the syntax of an existing sequential programming language to create a parallel programming language. Example **UPC – נמצאת על המכונה הוירטואלית**

• Using an existing sequential programming language supplemented with compiler directives and libraries for specifying parallelism. Example **OpenMP**

We will look mostly at threads and OpenMP

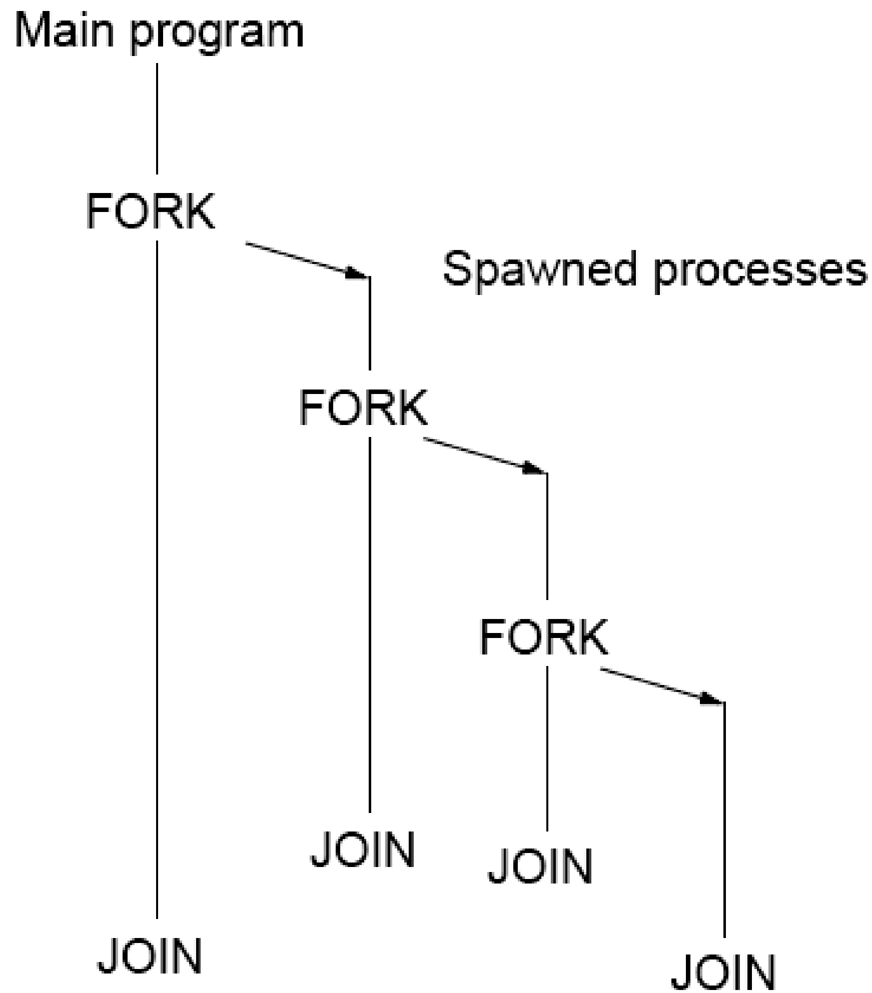
Threads and Processes

Added by Guy:



Reference: http://www.cs.cf.ac.uk/Dave/C/node29.html

FORK-JOIN construct



UNIX System Calls

No join routine - use exit() and wait()

SPMD model

```
      .
      .
   pid = fork();                          /* fork */
    Code to be executed by both child and parent
   if (pid == 0) exit(0); else wait(0); /* join */
      .
      .
      .
```

Wait()

The parent process will often want to wait until all child processes have been completed. this can be implemented with the wait() function call.

Wait() Blocks calling process until the child process terminates. If child process has already terminated, the wait() call returns immediately. if the calling process has multiple child processes, the function returns when one returns.

Open “top”:
top –u tel-zur

Guy - demo

Demos under
~/../08/code

eesrv.ee.bgu.ac.il - PuTTY

```
-bash-3.1$ cat ./threads1.c
#include <stdio.h>
int main() {
        int pid;
        printf("PID of this process: %d\n",getpid());
        pid = fork();
        printf("\n I am a threaded printf, pid=%d ",pid,"\n");
        printf("\n");
        if (pid == 0) return 0;
        else
        wait(0);

        return 0;
}

-bash-3.1$ gcc -o threads1 ./threads1.c
-bash-3.1$ ./threads1
PID of this process: 9313

 I am a threaded printf, pid=0

 I am a threaded printf, pid=9314
-bash-3.1$
```

UNIX System Calls

SPMD model with different code for master process and forked slave process.

Spawns a new process!

```
pid = fork();
if (pid == 0) {
  code to be executed by slave
} else {
  Code to be executed by parent
}
if (pid == 0) exit(0); else wait(0);
    :
```

Guy - demo

eesrv.ee.bgu.ac.il - PuTTY

```
-bash-3.1$ cat ./threads2.c
#include <stdio.h>
int main() {
        int pid;
        printf("PID of this proces: %d\n",getpid());
        pid = fork();
        if (pid==0)
           printf("I am thread 0, pid=%d\n",pid);
        else
           printf("I am thread 1, pid=%d\n",pid);
        if (pid == 0) return 0; else wait(0);

        return 0;
}

-bash-3.1$ gcc -o threads2 threads2.c
-bash-3.1$ ./threads2
PID of this proces: 9342
I am thread 0, pid=0
I am thread 1, pid=9343
-bash-3.1$
```

...same program with a delay added (cpu_burn): threads6.c

Threads> ./threads6
PID of this process: 22498

I am a threaded printf, pid=22499

I am a threaded printf, pid=0
^C

השם לא מוצלח.
לא מדובר ב- threads אלא ב- processes

fork() starts a new process (not a new thread!)

telzur@GL553VD ~/Documents/Teaching/PP2019A/lectures/08/code/Threads/solarisstudio

File Edit View Search Terminal Help

```
top - 14:18:48 up 7 days, 18:48,  1 user,  load average: 1.74, 1.21, 0.68
Threads: 1589 total,   3 running, 1506 sleeping,   0 stopped,   2 zombie
%Cpu(s): 25.6 us,  0.4 sy,  0.0 ni, 73.9 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem : 16307588 total,   276600 free, 11052272 used,  4978716 buff/cache
KiB Swap: 16659452 total, 15657860 free,  1001592 used.  4745948 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
22498 telzur    20   0    4508    860    796 R 99.9  0.0   0:58.75 threads6
22499 telzur    20   0    4508     80      0 R 99.9  0.0   0:58.70 threads6
 1472 root      20   0 1147328 265628 107512 S  2.6  1.6  52:09.70 Xorg
22504 telzur    20   0   43472   5248   3068 R  1.6  0.0   0:00.66 top
22515 telzur    20   0  495552  30780  24156 S  1.0  0.2   0:00.13 mate-screenshot
 8201 telzur    20   0 1121324  25500  17068 S  0.7  0.2   0:56.75 mate-settings-d
 8232 telzur    20   0  869488  44680  15208 S  0.7  0.3   6:37.10 marco
 8266 telzur    20   0  526136  49768  14952 S  0.7  0.3   1:04.26 wnck-applet
 1675 root      20   0 1147328 265628 107512 S  0.3  1.6   3:18.81 InputThread
```

This is the output of **`top -h`**

File Edit Selection View Go Debug Terminal Help

C threads6.c ×

```c
// This is needed in order to watch at the same time
// 'top' (task manager)
// Guy Tel-Zur, December 7, 2018
#include<stdio.h>
void cpu_burn() {
    int iMAX = 1000000;
    int i,j;
    float fNORM, fMAX;
    float x,y;
    fMAX = iMAX; // convert to float
    fNORM = fMAX * fMAX;
    for (j=0;j<iMAX;j++)
       for (i=0;i<iMAX;i++) {
           y = i;  // convert to float
           x = y*y/fNORM;
       }

return 0;
}
int main() {
    int pid;

    printf("PID of this process: %d\n",getpid());
    pid = fork();
        printf("\n I am a threaded printf, pid=%d ",pid);
        printf("\n");
        cpu_burn();
        if (pid == 0) return 0;
    else
        wait(0);
```

⊗ 0 ⚠ 0 (Global Scope) Ln 5, Col 5 Spaces: 4 UTF-8 LF C Linux

```
Threads> ./threads6
PID of this process: 23728

 I am a threaded printf, pid=23729

 I am a threaded printf, pid=23729

 I am a threaded printf, pid=0

 I am a threaded printf, pid=0
^C
Threads> 
```

4 processes

ents/Teaching/PP2019A/lectures/08/code/Threads/solarisstudio

ile Edit View Search Terminal Help

```
op - 14:28:44 up 7 days, 18:58,  1 user,  load average: 2.91, 1.59, 1.11
hreads: 1583 total,   5 running, 1502 sleeping,   0 stopped,   2 zombie
Cpu(s): 51.2 us,  2.5 sy,  0.0 ni, 46.3 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
iB Mem : 16307588 total,   409376 free, 11073884 used,  4824328 buff/cache
iB Swap: 16659452 total, 15664772 free,   994680 used.  4727368 avail Mem

 PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
3729 telzur    20   0    4508     80      0 R 99.9  0.0   1:15.93 threads6
3730 telzur    20   0    4508     80      0 R 99.9  0.0   1:15.93 threads6
3731 telzur    20   0    4508     80      0 R 99.9  0.0   1:15.92 threads6
3728 telzur    20   0    4508    820    756 R 99.7  0.0   1:15.88 threads6
1472 root      20   0 1146048 272212 106700 S  5.9  1.7  52:28.87 Xorg
3726 telzur    20   0   43472   5232   3048 R  1.3  0.0   0:01.30 top
8568 telzur    20   0 5675736 1.788g 115904 S  0.7 11.5  67:12.63 Chrome_IOThread
3917 telzur    20   0  495532  30880  24268 S  0.7  0.2   0:00.12 mate-screenshot
 568 root     -51   0       0      0      0 S  0.3  0.0   8:50.58 irq/128-iwlwifi
1213 telzur    20   0 2284352 494708  65468 S  0.3  3.0   9:24.36 chromium-browse
1629 root     -51   0       0      0      0 S  0.3  0.0   9:00.68 irq/130-nvidia
```

threads6.c - Visual Studio Code

File Edit Selection View Go Debug Terminal Help

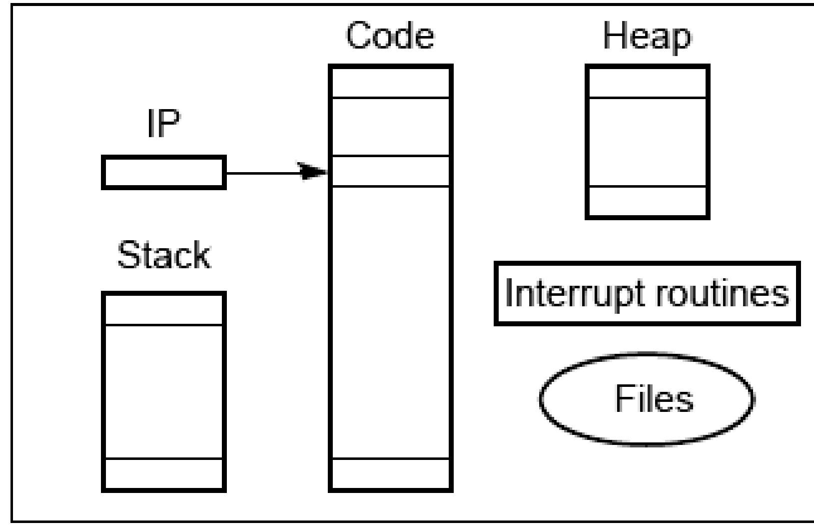
threads6.c

```c
    float x,y;
    fMAX = iMAX; // convert to float
    fNORM = fMAX * fMAX;
    for (j=0;j<iMAX;j++)
       for (i=0;i<iMAX;i++) {
           y = i;  // convert to float
           x = y*y/fNORM;
       }

return 0;
}
int main() {
    int pid,qid;

    printf("PID of this process: %d\n",getpid());
    pid = fork();
    qid = fork();
        printf("\n I am a threaded printf, pid=%d ",pid);
        printf("\n");
        cpu_burn();
        if (pid == 0) return 0;
    else
        wait(0);

    return 0;
}
```

Differences between a process and threads

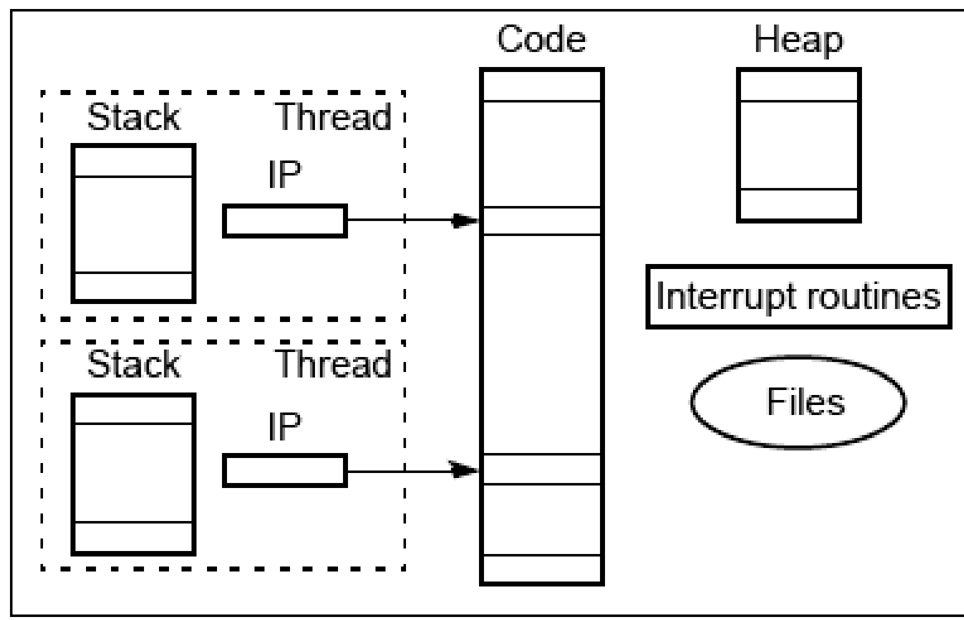
"heavyweight" process - completely separate program with its own variables, stack, and memory allocation.

(a) Process



Threads - shares the same memory space and global variables between routines.

(b) Threads



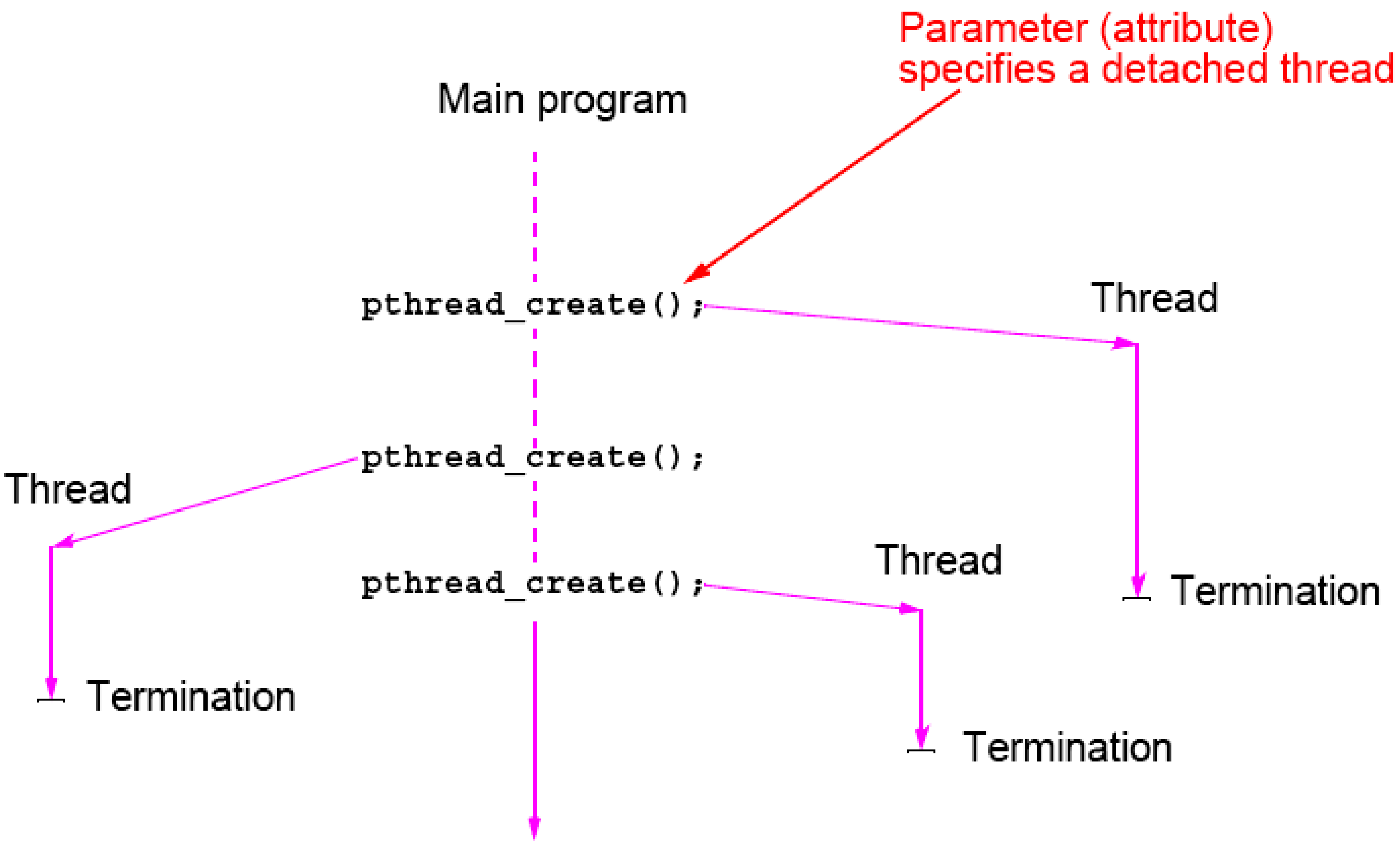
Detached Threads

It may be that thread are not bothered when a thread it creates terminates and then a join not needed.

Threads not joined are called *detached threads*.

When detached threads terminate, they are destroyed and their resource released.

Pthreads Detached Threads



Statement Execution Order

Single processor: Processes/threads typically executed until blocked.

Multiprocessor: Instructions of processes/threads interleaved in time.

Example

Process 1	*Process 2*
Instruction 1.1	Instruction 2.1
Instruction 1.2	Instruction 2.2
Instruction 1.3	Instruction 2.3

Many possible orderings, e.g.:

Instruction 1.1
Instruction 1.2
Instruction 2.1
Instruction 1.3
Instruction 2.2
Instruction 2.3

assuming instructions cannot be divided into smaller steps.

A sequence of instructions might perform a specific task, for example print out a message.

If two processes were to print messages, using interleaved instructions, the messages could appear garbled - the individual characters of each message could be interleaved if special care is not taken in coding the print routines.

Thread-Safe Routines

Thread safe if they can be called from multiple threads simultaneously and always produce correct results.

Standard I/O thread safe (prints messages without interleaving the characters).

System routines that return time may not be thread safe.

Routines that access shared data may require special care to be made thread safe.

Re-ordering code also done intentionally by:

- Compilers to optimize performance

- Processors internally, again to optimize performance

Compilers will re-order code prior to execution while processors will re-order the code during execution.

In both cases the objective is to best utilize the available computer resources and minimize any waiting time.

Compiler/Processor Optimizations

Compiler and processor reorder instructions to improve performance (execution speed).

Example

Suppose one had the code:

```
.
.
a = b + 5;
x = y * 4;
p = x + 9;
.
.
```

and the processor can perform, as is usual, multiple arithmetic operations at the same time.

Compiler/Processor Optimizations

Can reorder to:

```
.
.
x = y * 4;
a = b + 5;
p = x + 9;
.
.
```

and still be logically correct. This gives the multiply operation longer time to complete before the result (x) is needed in the last statement.

Very common for processors to execute machines instructions out of order for increased speed.

Accessing Shared Data

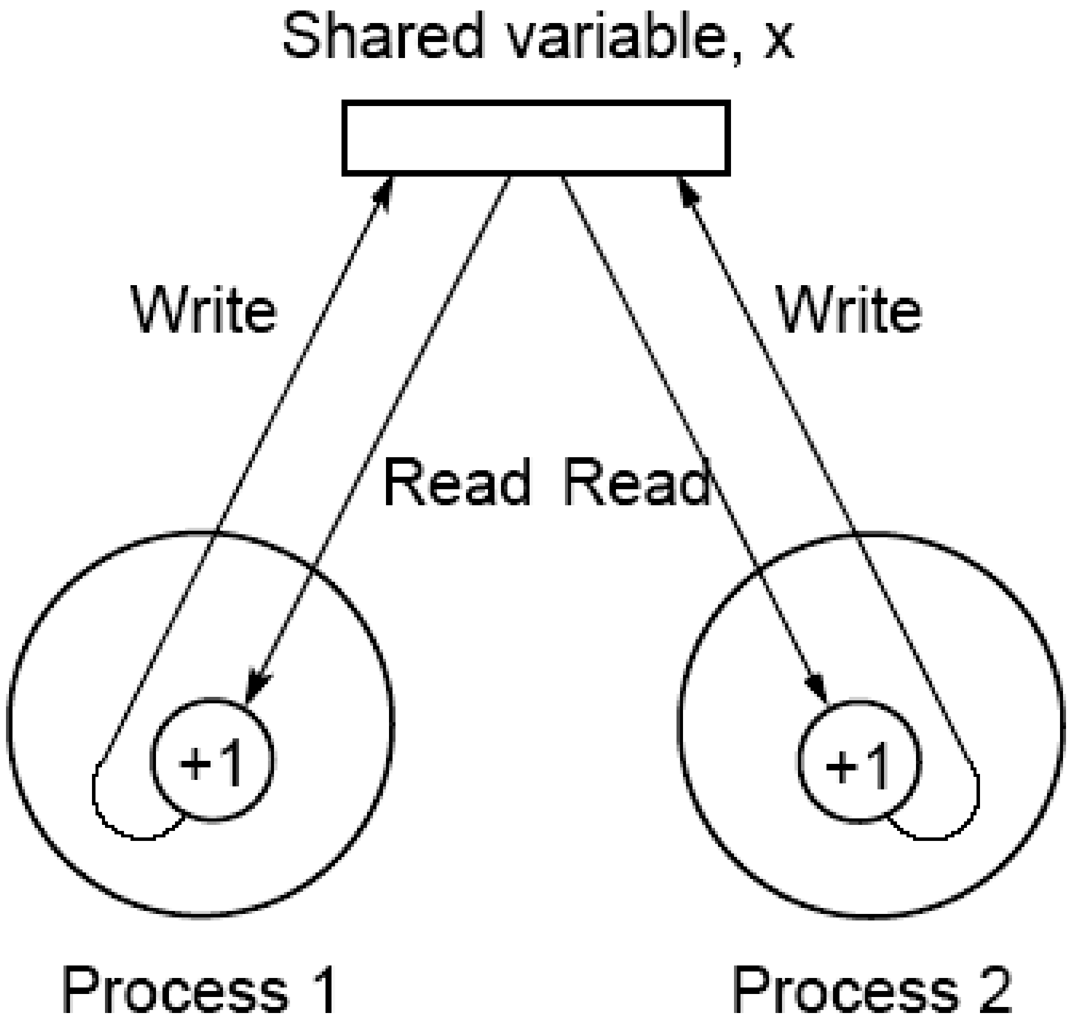
Accessing shared data needs careful control.

Consider two processes each of which is to add one to a shared data item, x.

Location x is read, x + 1 computed, and the result written back to the location:

	Instruction	Process 1	Process 2
	`x = x + 1;`	`read x`	`read x`
Time		`compute x + 1`	`compute x + 1`
		`write to x`	`write to x`

Conflict in accessing shared variable



Guy – demo

vdwarf2.ee.bgu.ac.il - PuTTY

```
vdwarf2.ee.bgu.ac.il> cat threads3.c
int main() {
        int pid;
        int x = 0;
        pid = fork();
        if (pid==0) {
           x = x+1;
           printf("I am thread 0, x=%d\n",x);
        }
        else {
           x = x+1;
           printf("I am thread 1, x=%d\n",x);
        }
        if (pid == 0) exit (0); else wait(0);
        printf("Finally x=%d\n",x);
        return 0;
}
vdwarf2.ee.bgu.ac.il> ./threads3
I am thread 0, x=1
I am thread 1, x=1
Finally x=1
vdwarf2.ee.bgu.ac.il>
```

Guy – demo; x is following the master thread

vdwarf2.ee.bgu.ac.il - PuTTY

```
vdwarf2.ee.bgu.ac.il> cat ./threads4.c
int main() {
        int pid;
        int x = 0;
        pid = fork();
        if (pid==0) {
           x = x+1;
           printf("I am thread 0, x=%d\n",x);
        }
        else {
           x = x+2;
           printf("I am thread 1, x=%d\n",x);
        }
        if (pid == 0) exit (0); else wait(0);
        printf("Finally x=%d\n",x);
        return 0;
}
vdwarf2.ee.bgu.ac.il> ./threads4
I am thread 0, x=1
I am thread 1, x=2
Finally x=2
vdwarf2.ee.bgu.ac.il>
```

Critical Section

A mechanism for ensuring that only one process accesses a particular resource at a time.

critical section – a section of code for accessing resource
Arrange that only one such critical section is executed at a time.

This mechanism is known as *mutual exclusion*.

Concept also appears in an operating systems.

Locks

Simplest mechanism for ensuring mutual exclusion of critical sections.

A lock - a 1-bit variable that is a 1 to indicate that a process has entered the critical section and a 0 to indicate that no process is in the critical section.

Operates much like that of a door lock:

A process coming to the “door” of a critical section and finding it open may enter the critical section, locking the door behind it to prevent other processes from entering. Once the process has finished the critical section, it unlocks the door and leaves.

Control of critical sections through busy waiting

Must be indivisible

Process 1

```
while (lock == 1) do_nothing;
lock = 1;
```

Critical section

```
lock = 0;
```

Process 2

```
while (lock == 1)do_nothing;
```

Better to deschedule process

```
lock = 1;
```

Critical section

```
lock = 0;
```

Critical Sections Serializing Code

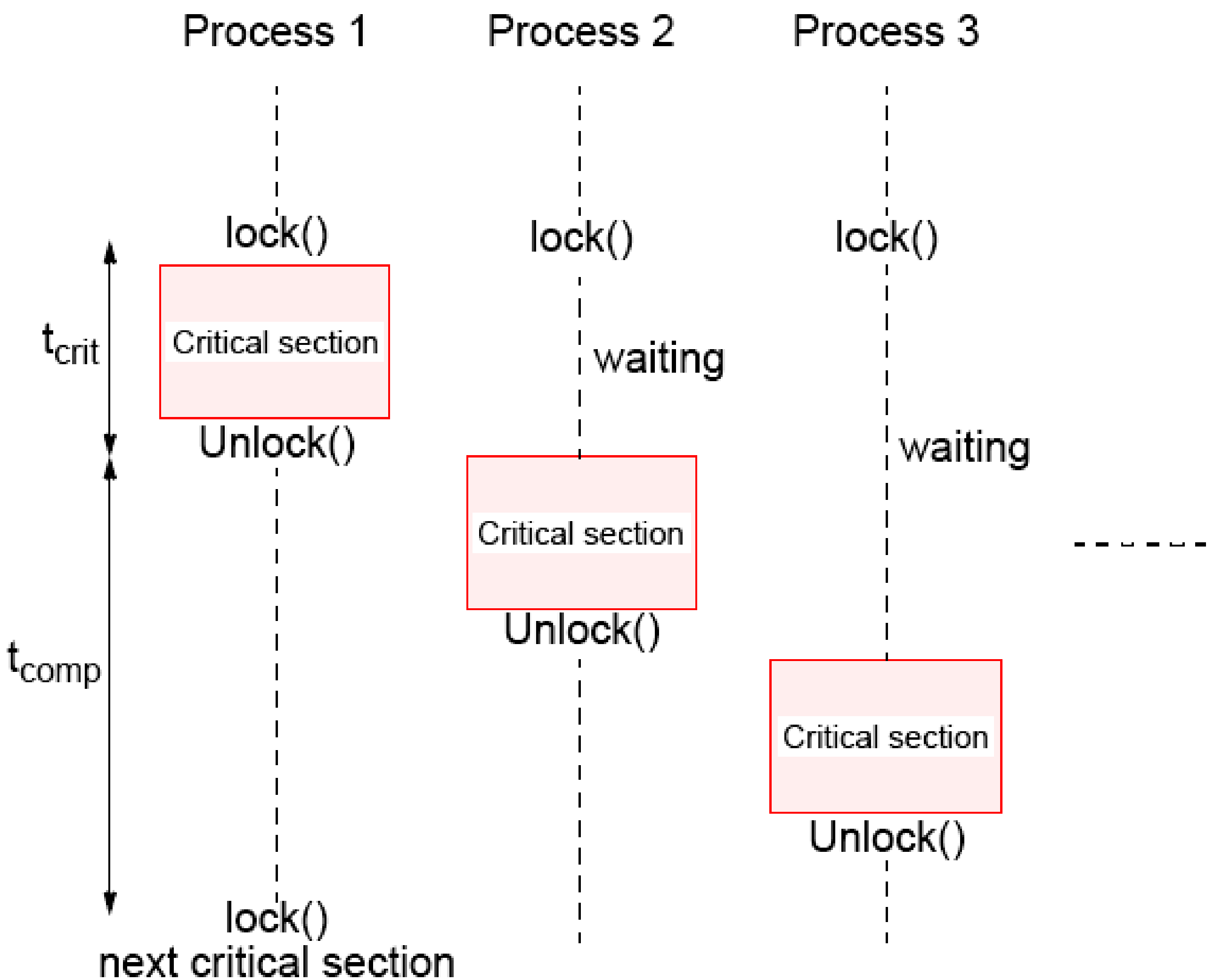
High performance programs should have as few as possible critical sections as their use can serialize the code.

Suppose, all processes happen to come to their critical section together.

They will execute their critical sections one after the other.

In that situation, the execution time becomes almost that of a single processor.

Illustration

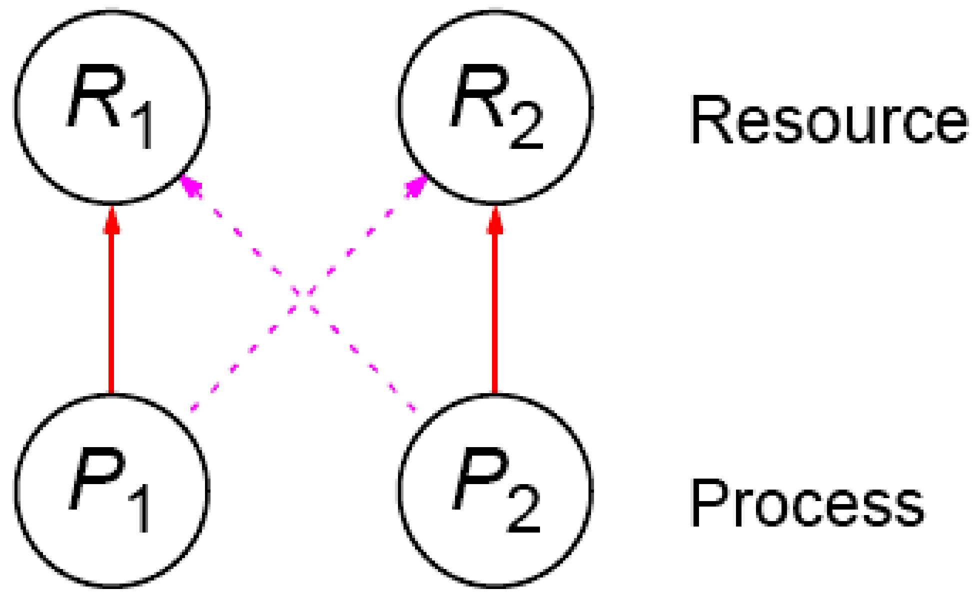


When $t_{comp} < pt_{crit}$, less than p processor will be active

Deadlock

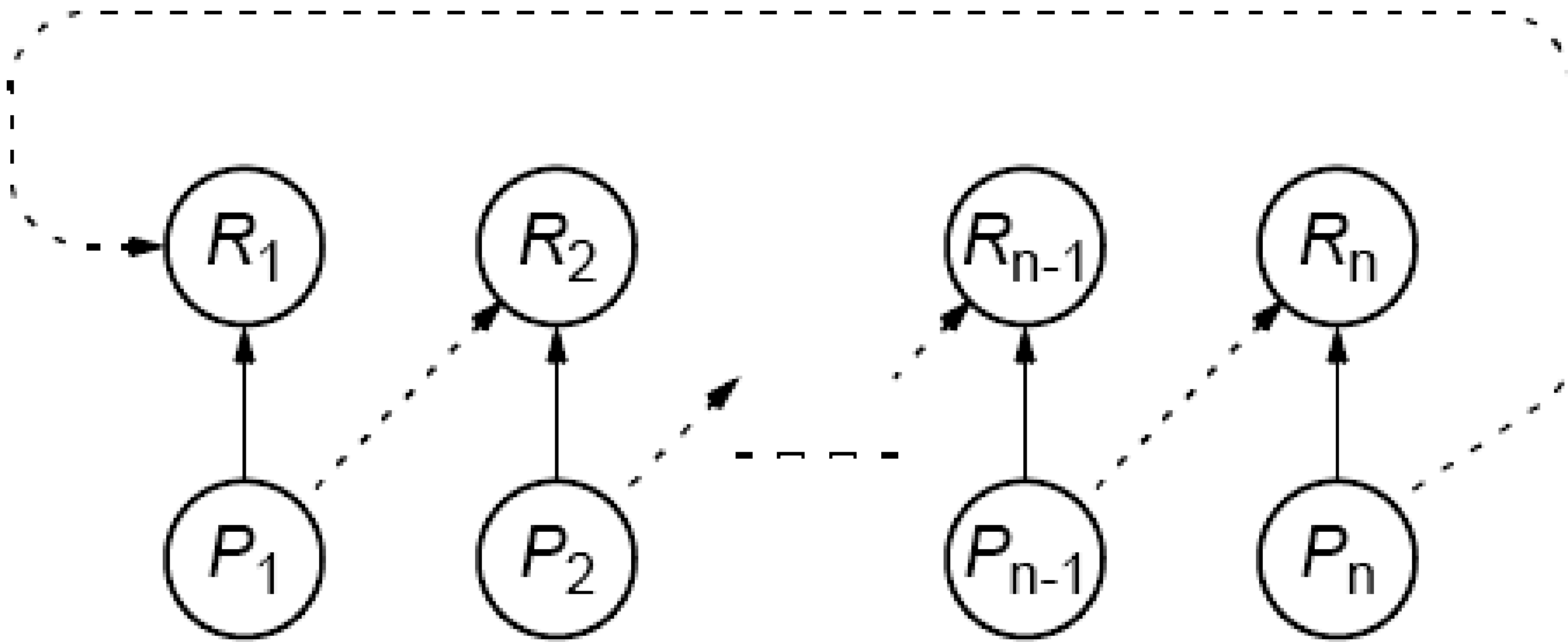
Can occur with two processes when one requires a resource held by the other, and this process requires a resource held by the first process.

Two-process deadlock



Deadlock (deadly embrace)

Deadlock can also occur in a circular fashion with several processes having a resource wanted by another.



Semaphores

A positive integer (including zero) operated upon by two operations:

P operation on semaphore *s*

Waits until *s* is greater than zero and then decrements *s* by one and allows the process to continue.

V operation **on semaphore *s***

Increments *s* by one and releases one of the waiting processes (if any).

P and **V** operations are performed indivisibly.

Mechanism for activating waiting processes implicit in **P** and **V** operations.

Though exact algorithm not specified, algorithm expected to be fair. Processes delayed by **P**(s) are kept in abeyance(*) until released by a **V**(s) on the same semaphore.

(*) השהייה, אי-הפעלה
Devised by Dijkstra in 1968.

Letter **P** from Dutch word *passeren*, meaning “to pass”
Letter **V** from Dutch word *vrijgeven*, meaning “to release”

Mutual exclusion of critical sections can be achieved with one semaphore having the value 0 or 1 (a binary semaphore), which acts as a lock variable, but the P and V operations include a process scheduling mechanism:

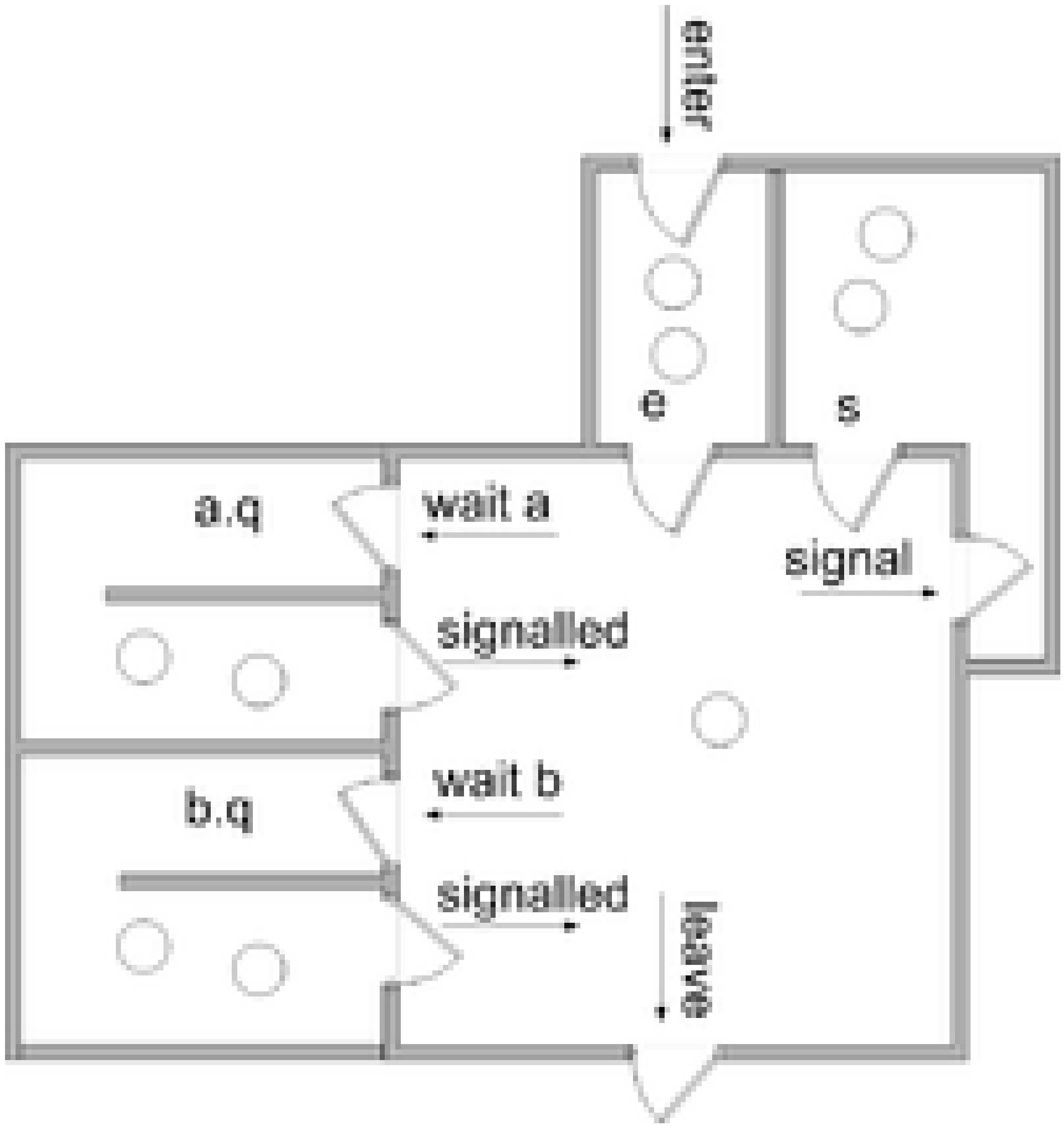
Process 1	***Process 2***	***Process 3***
Noncritical section	Noncritical section	Noncritical section
.	.	.
P(s)	**P(s)**	**P(s)**
Critical section	Critical section	Critical section
V(s)	**V(s)**	**V(s)**
.	.	.
Noncritical section	Noncritical section	Noncritical section

Monitor (=condition variable + lock)

Suite of procedures that provides only way to access shared resource. Only one process can use a monitor procedure at any instant.

Could be implemented using a semaphore or lock to protect entry, i.e.:

```
monitor_proc1() {
lock(x);
        .
monitor body
        .
unlock(x);
return;
}
```



Source: http://en.wikipedia.org/wiki/Monitor_(synchronization)

Program example

To sum the elements of an array, **a[1000]**:

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

UNIX Processes

Calculation will be divided into two parts, one doing even *i* and one doing odd *i*; i.e.,

Process 1

```
sum1 = 0;
for (i = 0; i < 1000; i = i + 2)
sum1 = sum1 + a[i];
```

Process 2

```
sum2 = 0;
for (i = 1; i < 1000; i = i + 2)
sum2 = sum2 + a[i];
```

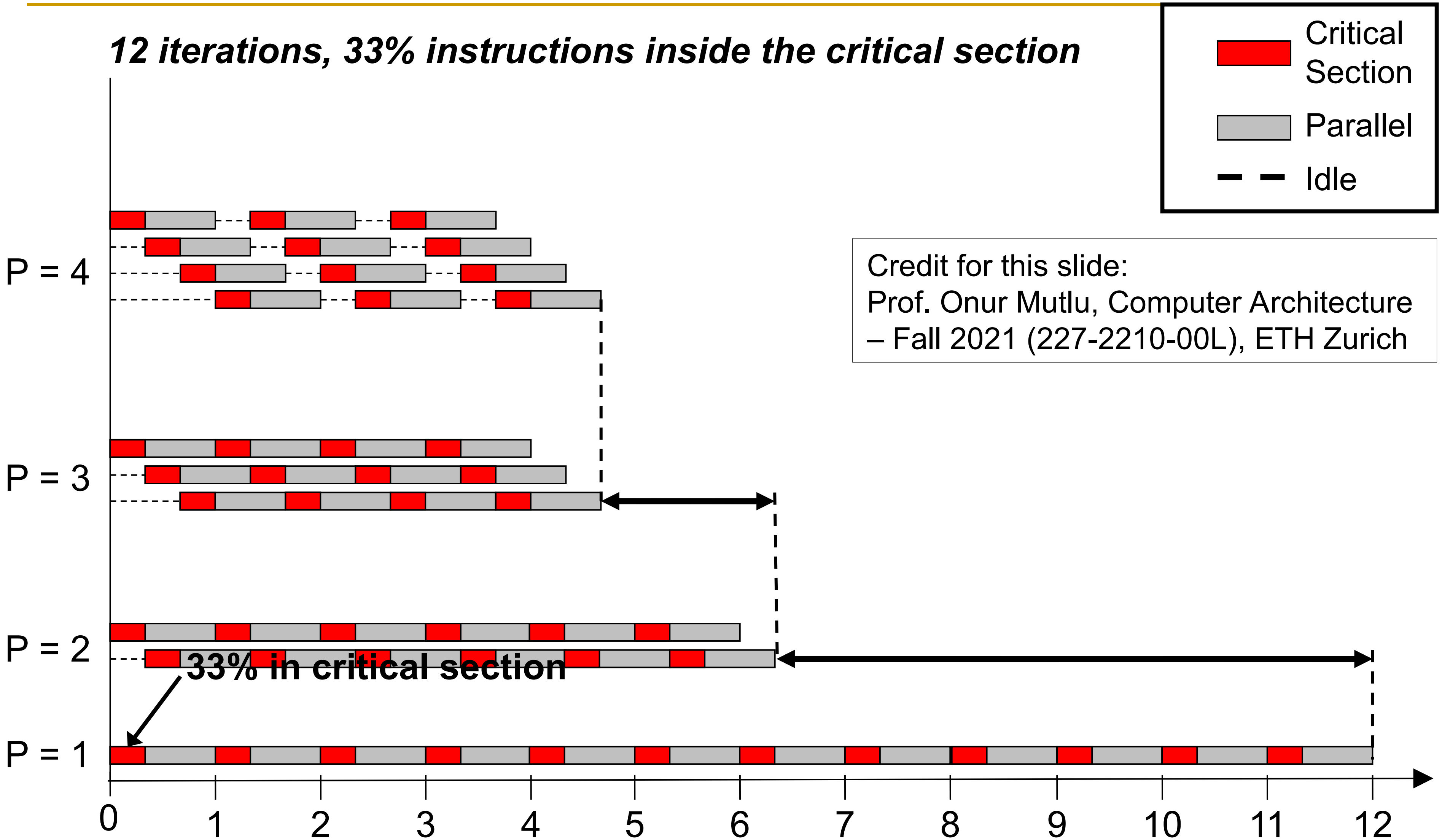
Each process will add its result (sum1 or sum2) to an accumulating result, sum :

```
sum = sum + sum1;
```

```
sum = sum + sum2;
```

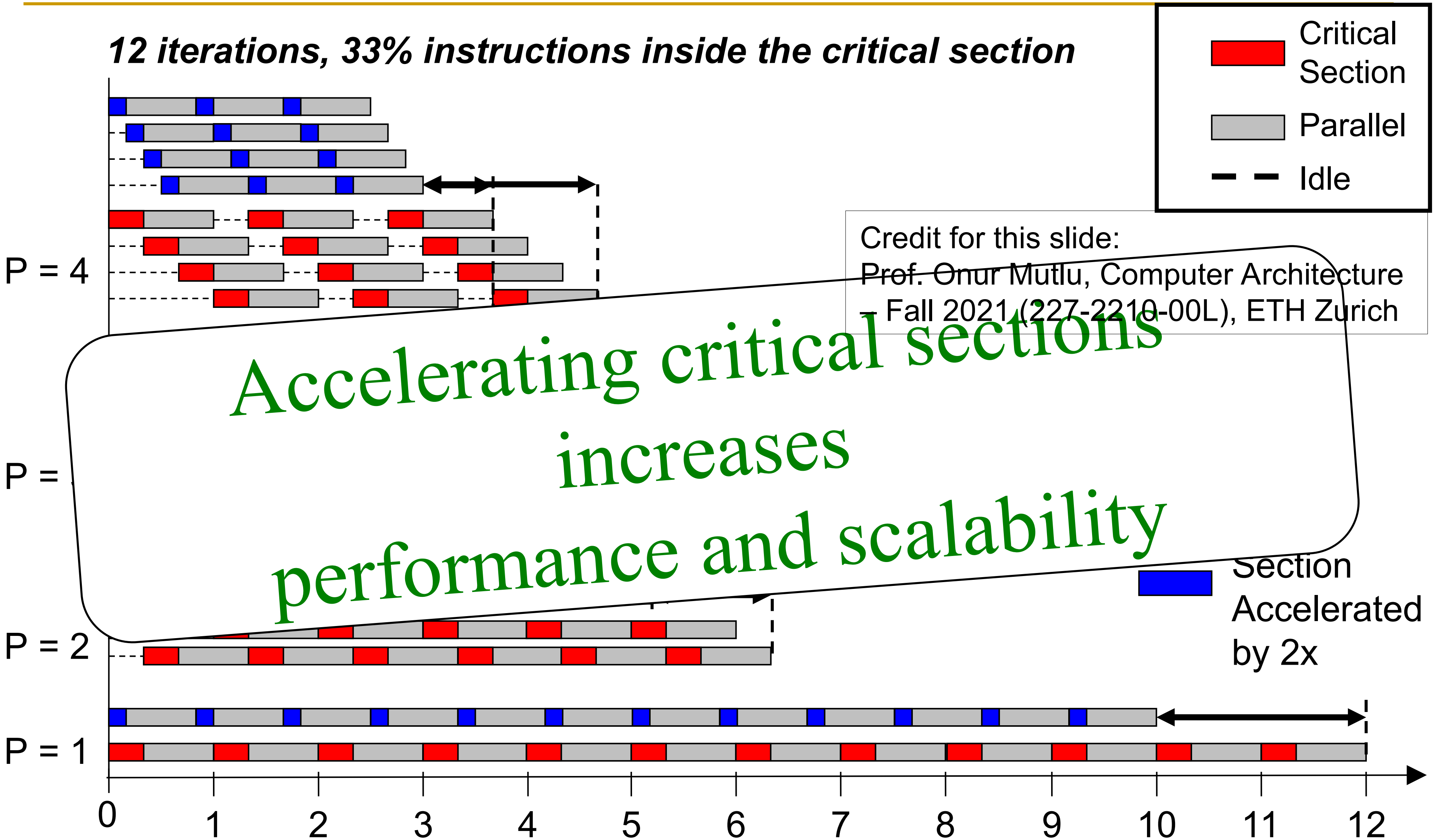
Sum will need to be shared and protected by a lock. Shared data structure is created:

Contention for Critical Sections



Credit for this slide:
Prof. Onur Mutlu, Computer Architecture – Fall 2021 (227-2210-00L), ETH Zurich

Contention for Critical Sections



Credit for this slide:
Prof. Onur Mutlu, Computer Architecture – Fall 2021 (227-2210-00L), ETH Zurich

עד כאן מצגת זו

Programming with Shared Memory

Part 2

Introduction to OpenMP

OpenMP

An accepted standard developed in the late 1990s by a group of industry specialists.

Consists of a small set of compiler directives, augmented with a small set of library routines and environment variables using the base language Fortran and C/C++.

Several OpenMP compilers available.

OpenMP

- Uses a thread-based shared memory programming model
- OpenMP programs will create multiple threads
- All threads have access to global memory
- Data can be shared among all threads or private to one thread
- Data transfer hidden from programmer
- Synchronization occurs but often implicit

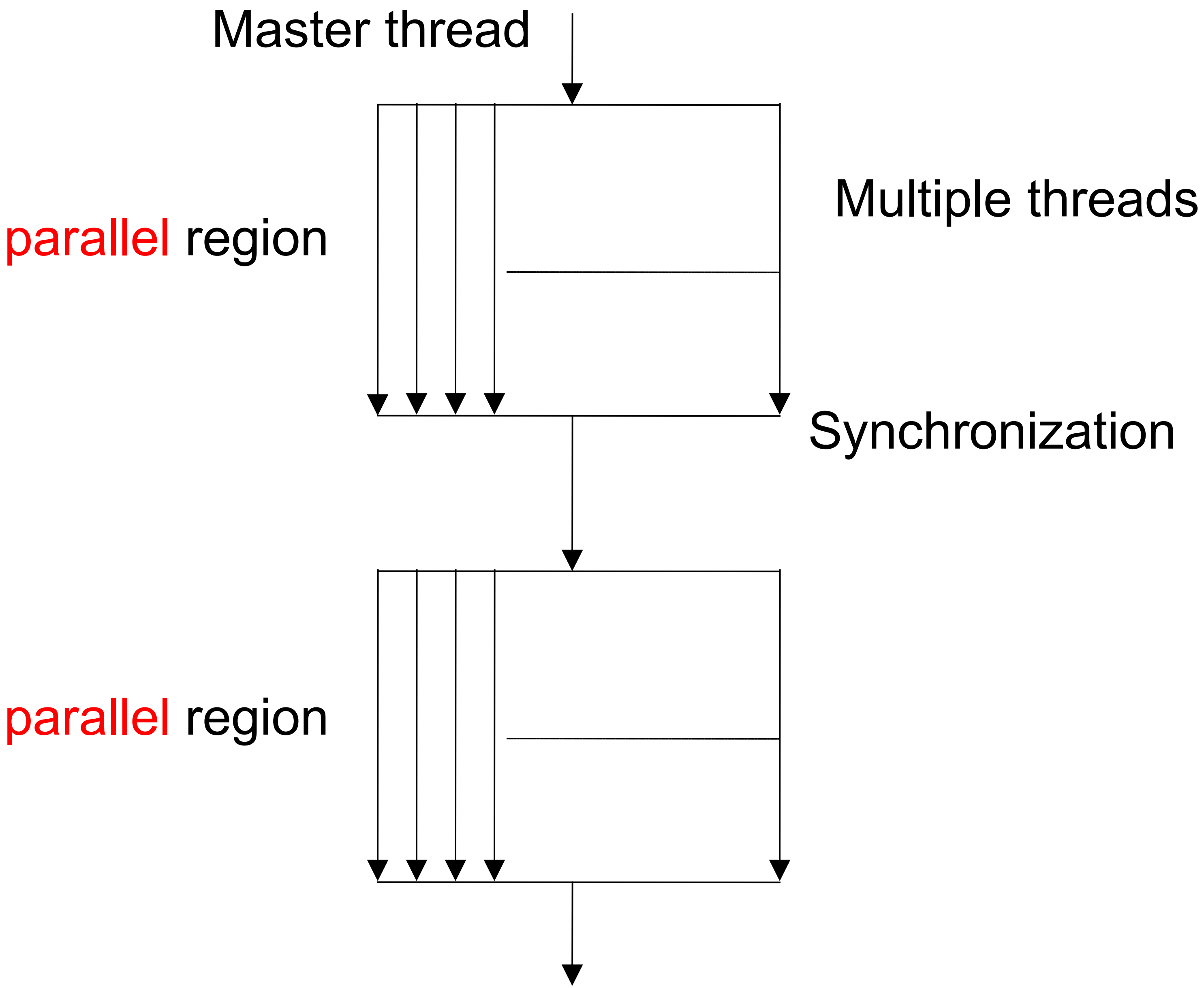
OpenMP uses "**fork-join**" model but thread-based.

Initially, a single thread is executed by a **master thread**. **Parallel regions** (sections of code) can be executed by multiple threads (a team of threads).

parallel directive creates a team of threads with a specified block of code executed by the multiple threads in parallel. The exact number of threads in the team determined by one of several ways.

Other directives used within a parallel construct to specify parallel for loops and different blocks of code for threads.

Fork/join model



For C/C++, the OpenMP directives are contained in #pragma statements. The OpenMP #pragma statements have the format:

#pragma omp directive_name ...

where omp is an OpenMP keyword.

May be additional parameters (clauses) after the directive name for different options.

Some directives require code to specified in a structured block that follows the directive and then the directive and structured block form a “construct”.

Parallel Directive

```
#pragma omp parallel
    structured_block
```

creates multiple threads, each one executing the specified structured_block, either a single statement or a compound statement created with { ...} with a single entry point and a single exit point.

There is an implicit barrier at the end of the construct. The directive corresponds to forall construct.

Hello world example

OpenMP directive for a parallel region

```
#pragma omp parallel  {

printf("Hello World from thread = %d\n", omp_get_thread_num(),
                                          omp_get_num_threads());
}
```

From an 8-processor/core machine:

Hello World from thread 0 of 8
Hello World from thread 4 of 8
Hello World from thread 3 of 8
Hello World from thread 2 of 8
Hello World from thread 7 of 8
Hello World from thread 1 of 8
Hello World from thread 6 of 8
Hello World from thread 5 of 8

Private thread variables and shared variables

Could be declared within each parallel region but OpenMP provides private and shared clauses

```
int tid;
…
#pragma omp parallel private(tid) {
        tid = omp_get_thread_num();
        printf("Hello World from thread = %d\n", tid);
}
```

Guy: demo

```
bash% export OMP_NUM_THREADS=8


csh% setenv OMP_NUM_THREADS 8
```

Folder:
…/lectures/08/code/HelloOpenMP

Program:
`omp1.c`

Compile:
```
gcc -fopenmp  -o omp1 ./omp1.c
```

Guy: Demo

eesrv.ee.bgu.ac.il - PuTTY

```
-bash-3.1$ pwd
/users/agnon/misc/tel-zur/teaching/sharedmem
-bash-3.1$ cat ./omp1.c
#include <omp.h>
#include <stdio.h>

int main() {
int tid;

#pragma omp parallel private(tid)
{
        tid = omp_get_thread_num();
        printf("Hello World from thread = %d\n", tid);
}

return 0;
}
-bash-3.1$
```

Guy: demo

File Edit View Search Terminal Help

```
telzur@GL553VD ~/science/Teaching/PP/lectures/08/code/HelloOpenMP $ export OMP_NUM_THREADS=8
telzur@GL553VD ~/science/Teaching/PP/lectures/08/code/HelloOpenMP $ gcc -fopenmp -o omp1 ./omp1.
c
telzur@GL553VD ~/science/Teaching/PP/lectures/08/code/HelloOpenMP $ ./omp1
Hi, now I am serial
Hello World from thread 0=
Hello World from thread 7=
Hello World from thread 5=
Hello World from thread 3=
Hello World from thread 1=
Hello World from thread 6=
Hello World from thread 2=
Hello World from thread 4=
Now I am serial again
telzur@GL553VD ~/science/Teaching/PP/lectures/08/code/HelloOpenMP $
```

Example

```
#pragma omp parallel private(x, num_threads)
{
  x = omp_get_thread_num();
  num_threads = omp_get_num_threads();
  a[x] = 10*num_threads;
}
```

Two library routines

`omp_get_num_threads()` returns number of threads that are currently being used in parallel directive

`omp_get_thread_num()`returns thread number (an integer from 0 to `omp_get_num_threads()`– 1 where thread 0 is the master thread).

Array `a[]` is a global array, and `x` and `num_threads`are declared as private to the threads.

Number of threads in a team

Established by either:

1. num_threads clause after the parallel directive, or
2. omp_set_num_threads() library routine being previously called,

or

3. the environment variable OMP_NUM_THREADS is defined

in the order given or is system dependent if none of the above.

Number of threads available can also be altered automatically to achieve best use of system resources by a “dynamic adjustment” mechanism.

Work-Sharing

Three constructs in this classification:

sections
for
single

In all cases, there is **an implicit barrier** at the end of the construct unless a nowait clause is included.

Sections

The construct

```
#pragma omp sections
{
    #pragma omp section
        structured_block
    #pragma omp section
        structured_block
            .
            .
            .
}
```

cause the structured blocks to be shared among threads in team.
#pragma omp sections precedes the set of structured blocks.
#pragma omp section prefixes each structured block.

The first section directive is optional.

Example

One thread does this

Another thread does this

```
#pragma omp parallel shared(a,b,c,d,nthreads) private(i,tid) {
  tid = omp_get_thread_num();

#pragma omp sections nowait  {
        #pragma omp section  {
        printf("Thread %d doing section 1\n",tid);
        for (i=0; i<N; i++) {
                c[i] = a[i] + b[i];
                printf("Thread %d: c[%d]= %f\n",tid,i,c[i]);
        }
        }

        #pragma omp section {
        printf("Thread %d doing section 2\n",tid);
        for (i=0; i<N; i++) {
                d[i] = a[i] * b[i];
                printf("Thread %d: d[%d]= %f\n",tid,i,d[i]);
        }
        }
 }  /* end of sections */
}  /* end of parallel section */
```

```
#include<stdio.h>
#include<omp.h>
int main() {
int id;
#pragma omp parallel
{
#pragma omp sections
{
    #pragma omp section
    {
        printf ("Section 1.1 id = %d, \n", omp_get_thread_num());
    }
    #pragma omp section
    {
        printf ("Section 1.2 id = %d, \n", omp_get_thread_num());
    }
}
#pragma omp sections
{
    #pragma omp section
    {
        printf ("Section 2.1 id = %d, \n", omp_get_thread_num());
    }
    #pragma omp section
    {
        printf ("Section 2.2 id = %d, \n", omp_get_thread_num());
    }
}
} // end pragma omp parallel
return 0;
}
```

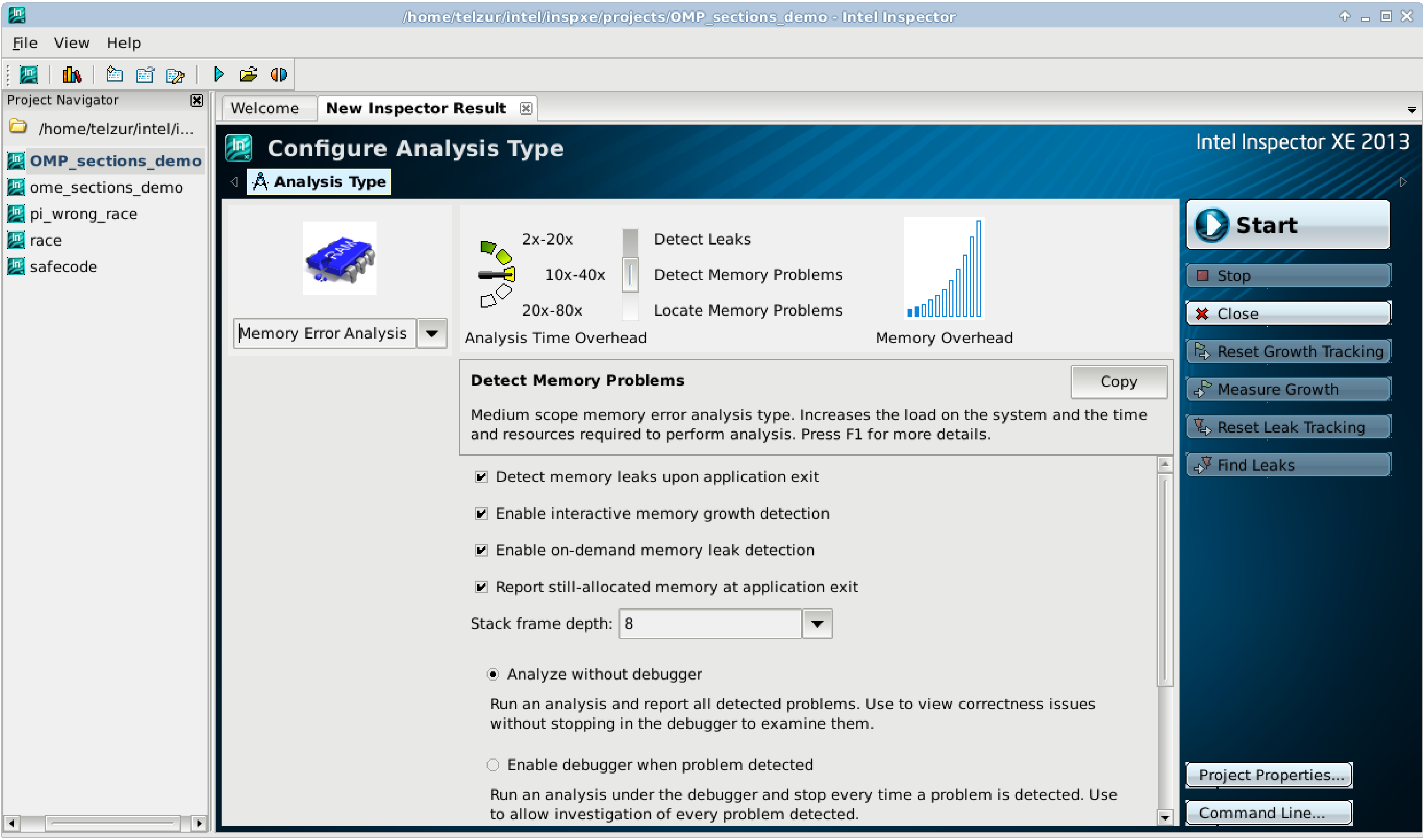
Demo: /home/telzur/Documents/Teaching/BGU/PP/PP2015A/lectures/08/code

```
telzur@LIFEBOOK ~/Documents/Teaching/BGU/PP/PP2015A/lectures/08/code $ gcc -fopenmp -o omp_
sections_demo ./omp_sections_demo.c
telzur@LIFEBOOK ~/Documents/Teaching/BGU/PP/PP2015A/lectures/08/code $ ./omp_sections_demo
Section 1.1 id = 5,
Section 1.2 id = 7,
Section 2.1 id = 1,
Section 2.2 id = 6,
telzur@LIFEBOOK ~/Documents/Teaching/BGU/PP/PP2015A/lectures/08/code $ ./omp_sections_demo
Section 1.1 id = 6,
Section 1.2 id = 3,
Section 2.1 id = 1,
Section 2.2 id = 0,
telzur@LIFEBOOK ~/Documents/Teaching/BGU/PP/PP2015A/lectures/08/code $ ./omp_sections_demo
Section 1.2 id = 5,
Section 1.1 id = 4,
Section 2.2 id = 2,
Section 2.1 id = 0,
telzur@LIFEBOOK ~/Documents/Teaching/BGU/PP/PP2015A/lectures/08/code $
```

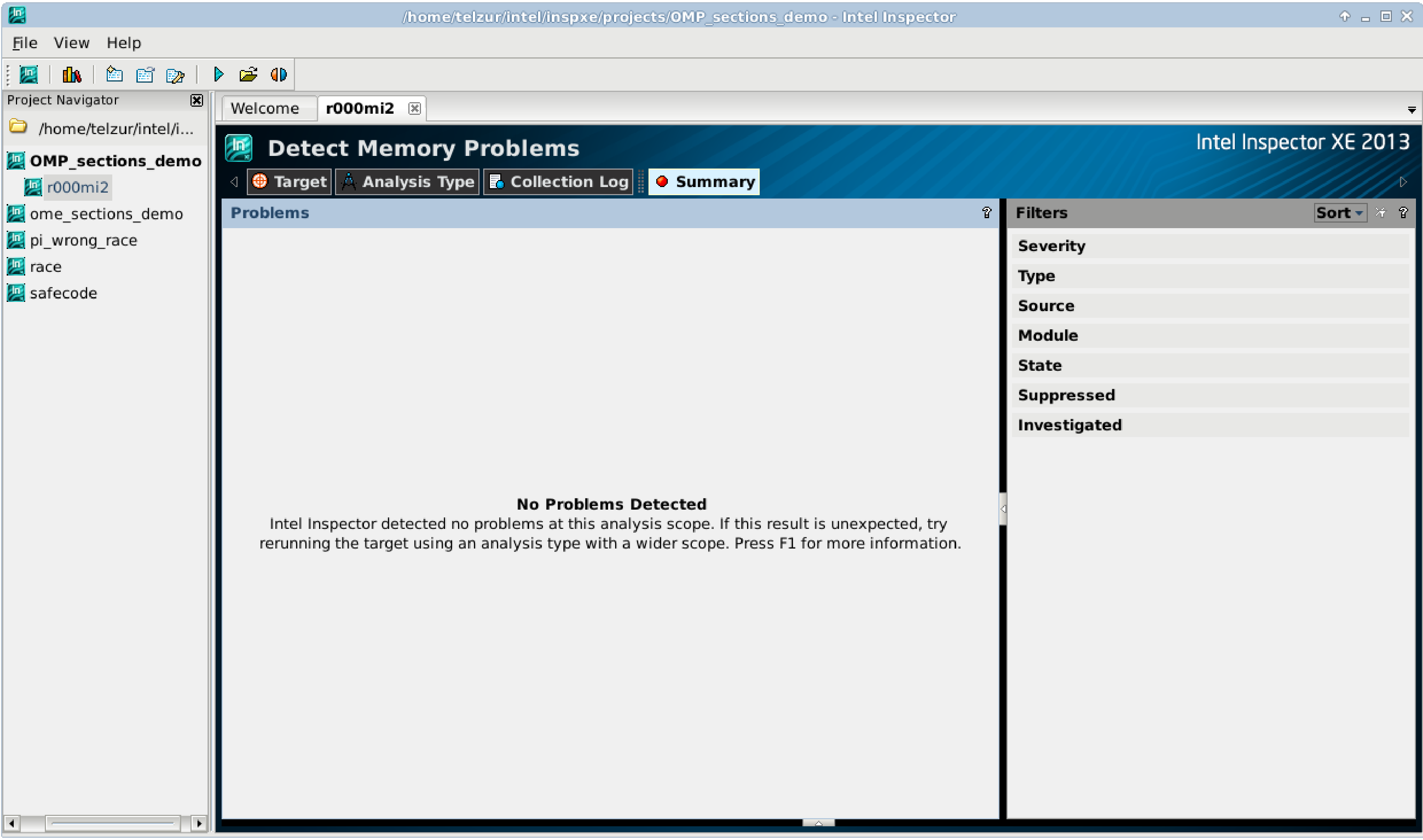
```
$ export OMP_NUM_THREADS=2
$ ./omp_sections_demo
Section 1.1 id = 0,
Section 1.2 id = 1,
Section 2.1 id = 0,
Section 2.2 id = 1,
$ export OMP_NUM_THREADS=3
$ ./omp_sections_demo
Section 1.1 id = 0,
Section 1.2 id = 1,
Section 2.2 id = 1,
Section 2.1 id = 2,
$ export OMP_NUM_THREADS=4
$ ./omp_sections_demo
Section 1.1 id = 0,
Section 1.2 id = 3,
Section 2.1 id = 0,
Section 2.2 id = 1,
$
```

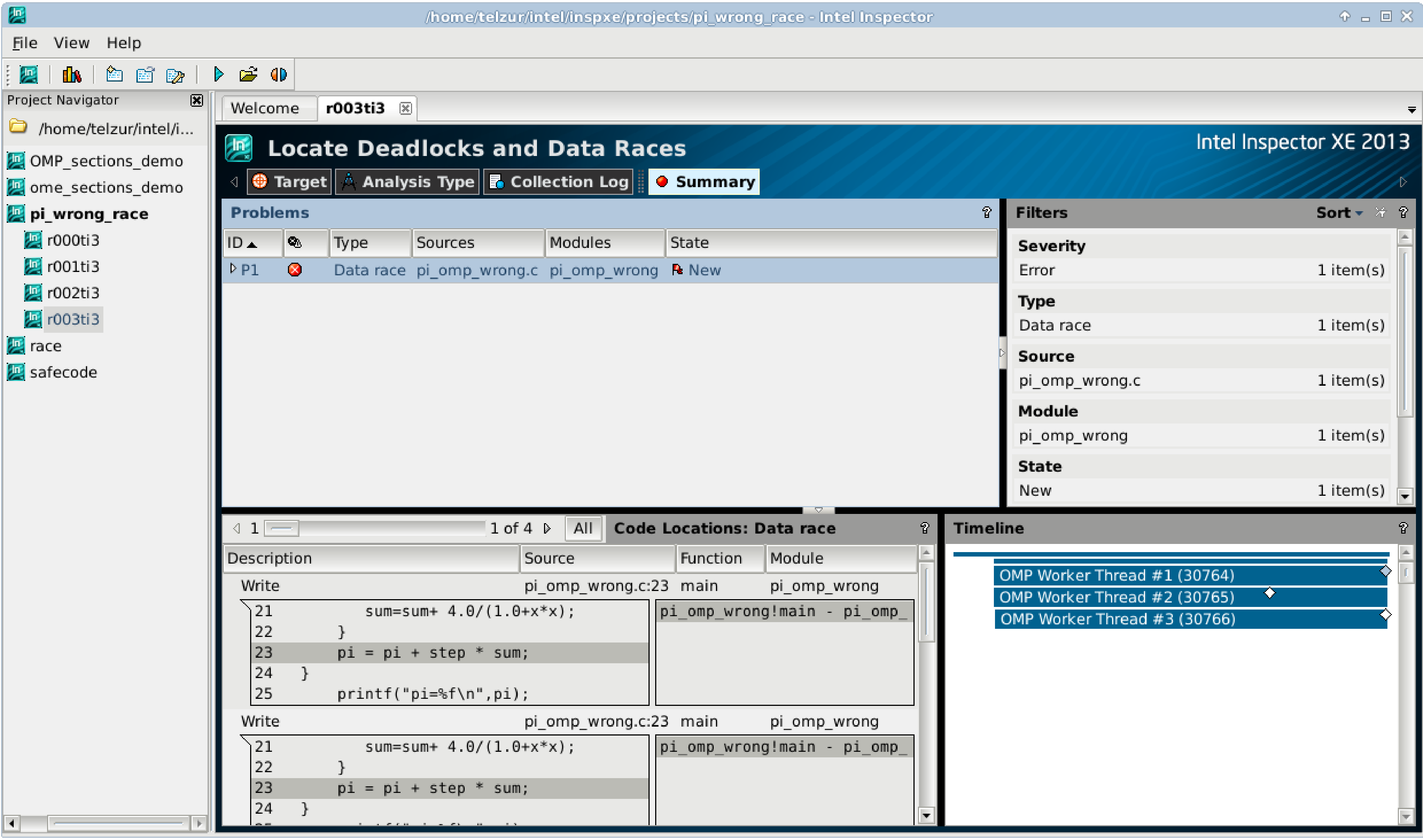
Intel Parallel Studio

- `export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/opt/intel/composer_xe_2013_sp1.3.174/compiler/lib/intel64`
- `echo 0 | sudo tee /proc/sys/kernel/yama/ptrace_scope`
- `/opt/intel/bin/icc -openmp -g -o ./omp_sections_demo_intel ./omp_sections_demo.c`
- **Intel Inspector:** `/opt/intel/vtune_amplifier_xe/bin64/amplxe-gui &`


/home/telzur/intel/inspxe/projects/OMP_sections_demo - Intel Inspector
File
View
Help
Project Navigator
/home/telzur/intel/i...
OMP_sections_demo
ome_sections_demo
pi_wrong_race
race
safecode
Welcome
New Inspector Result
Configure Analysis Type
Intel Inspector XE 2013
Analysis Type
Memory Error Analysis
2x-20x
10x-40x
20x-80x
Detect Leaks
Detect Memory Problems
Locate Memory Problems
Analysis Time Overhead
Memory Overhead
Start
Stop
Close
Reset Growth Tracking
Measure Growth
Reset Leak Tracking
Find Leaks
Detect Memory Problems
Copy
Medium scope memory error analysis type. Increases the load on the system and the time and resources required to perform analysis. Press F1 for more details.
Detect memory leaks upon application exit
Enable interactive memory growth detection
Enable on-demand memory leak detection
Report still-allocated memory at application exit
Stack frame depth:
8
Analyze without debugger
Run an analysis and report all detected problems. Use to view correctness issues without stopping in the debugger to examine them.
Enable debugger when problem detected
Run an analysis under the debugger and stop every time a problem is detected. Use to allow investigation of every problem detected.
Project Properties...
Command Line...

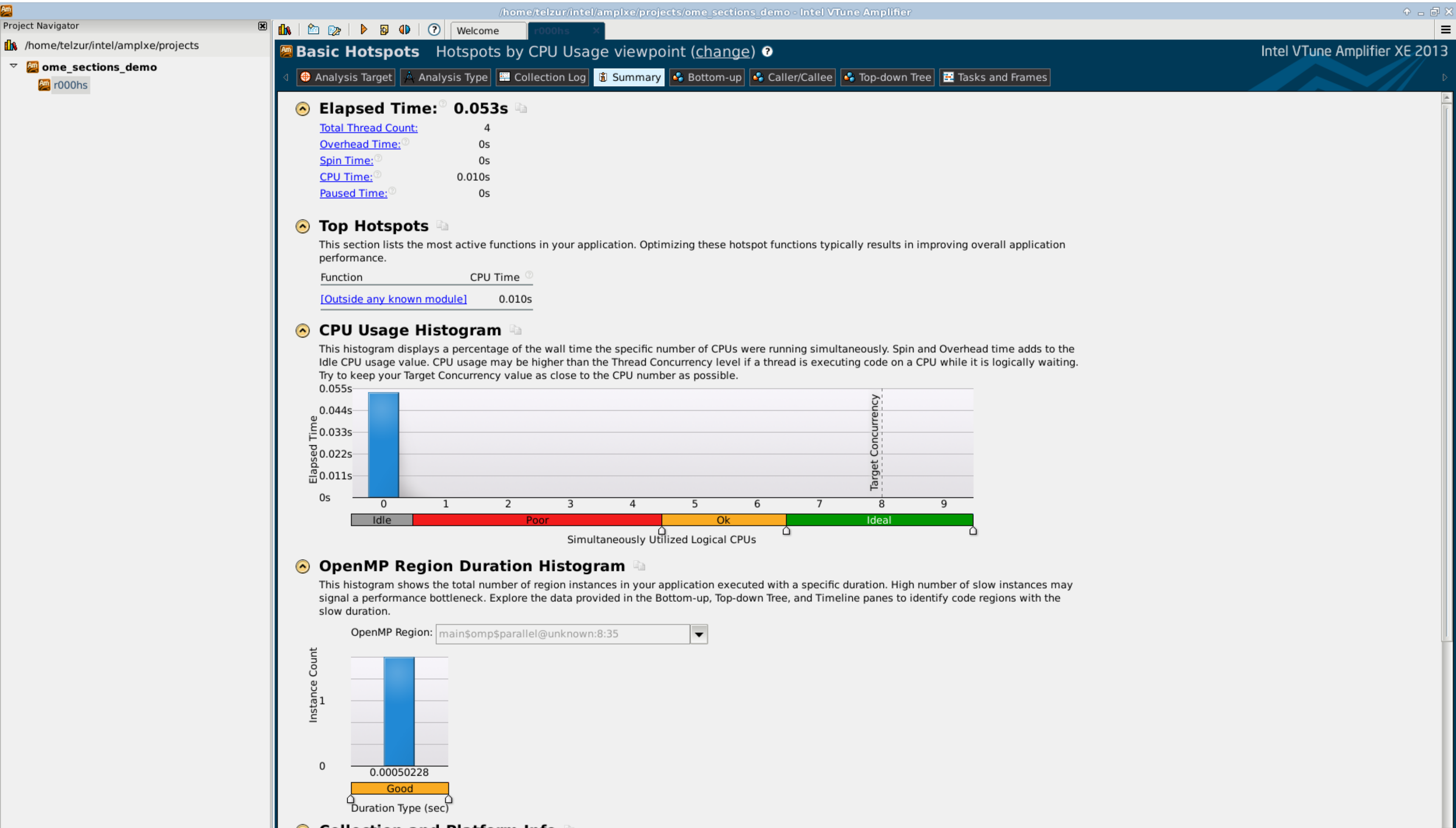
No race conditions were detected

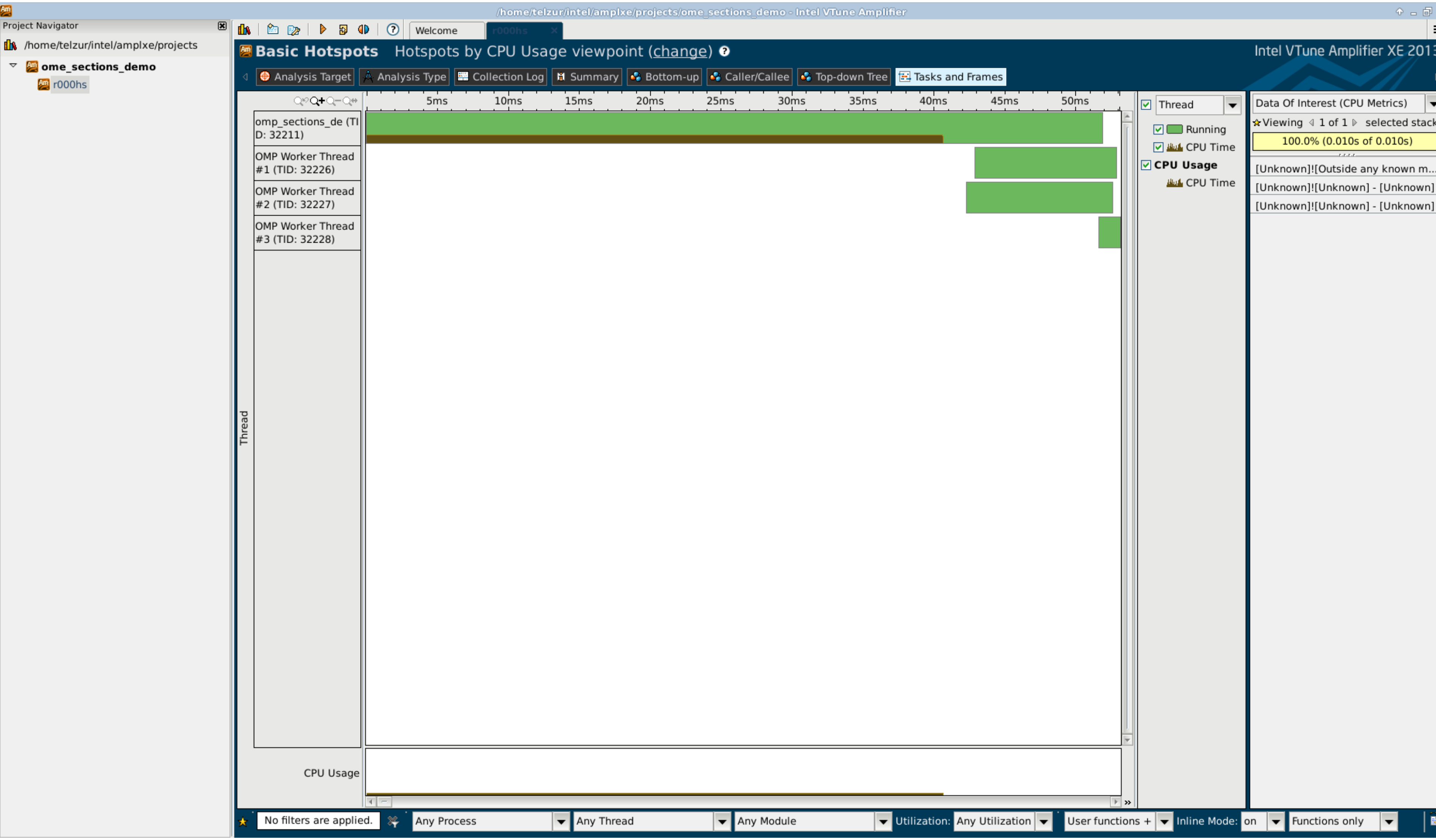



/home/telzur/intel/inspxe/projects/pi_wrong_race - Intel Inspector
File View Help
Project Navigator
/home/telzur/intel/i...
OMP_sections_demo
ome_sections_demo
pi_wrong_race
r000ti3
r001ti3
r002ti3
r003ti3
race
safecode
Welcome
r003ti3
Locate Deadlocks and Data Races
Intel Inspector XE 2013
Target
Analysis Type
Collection Log
Summary
Problems
ID
Type
Sources
Modules
State
P1
Data race
pi_omp_wrong.c
pi_omp_wrong
New
Filters
Sort
Severity
Error 1 item(s)
Type
Data race 1 item(s)
Source
pi_omp_wrong.c 1 item(s)
Module
pi_omp_wrong 1 item(s)
State
New 1 item(s)
1
1 of 4
All
Code Locations: Data race
Description
Source
Function
Module
Write pi_omp_wrong.c:23 main pi_omp_wrong
21 sum=sum+ 4.0/(1.0+x*x);
22 }
23 pi = pi + step * sum;
24 }
25 printf("pi=%f\n",pi);
pi_omp_wrong!main - pi_omp_
Write pi_omp_wrong.c:23 main pi_omp_wrong
21 sum=sum+ 4.0/(1.0+x*x);
22 }
23 pi = pi + step * sum;
24 }
pi_omp_wrong!main - pi_omp_
Timeline
OMP Worker Thread #1 (30764)
OMP Worker Thread #2 (30765)
OMP Worker Thread #3 (30766)

Intel's Vtune

```
$  /opt/intel/vtune_amplifier_xe/bin64/amplxe-gui
```




/home/telzur/intel/amplxe/projects/ome_sections_demo - Intel VTune Amplifier
Project Navigator
/home/telzur/intel/amplxe/projects
ome_sections_demo
r000hs
Welcome
r000hs
Basic Hotspots
Hotspots by CPU Usage viewpoint (change)
Intel VTune Amplifier XE 2013
Analysis Target
Analysis Type
Collection Log
Summary
Bottom-up
Caller/Callee
Top-down Tree
Tasks and Frames
5ms
10ms
15ms
20ms
25ms
30ms
35ms
40ms
45ms
50ms
omp_sections_de (TID: 32211)
OMP Worker Thread #1 (TID: 32226)
OMP Worker Thread #2 (TID: 32227)
OMP Worker Thread #3 (TID: 32228)
Thread
CPU Usage
Thread
Running
CPU Time
CPU Usage
CPU Time
Data Of Interest (CPU Metrics)
Viewing 1 of 1 selected stack(s
100.0% (0.010s of 0.010s)
[Unknown]![Outside any known m...
[Unknown]![Unknown] - [Unknown]
[Unknown]![Unknown] - [Unknown]
No filters are applied.
Any Process
Any Thread
Any Module
Utilization:
Any Utilization
User functions +
Inline Mode:
on
Functions only

Single

The directive

```
#pragma omp single
    structured block
```

cause the structured block to be executed by one thread only.

Master Directive

The master directive:

```
#pragma omp master
    structured_block
```

causes the master thread to execute the structured block.

Different to those in the work sharing group in that there is no implied barrier at the end of the construct (nor the beginning). Other threads encountering this directive will ignore it and the associated structured block, and will move on.

If a `parallel` directive is followed by a single `sections` directive, it can be combined into:

```
#pragma omp parallel sections {
  #pragma omp section
    structured_block
  #pragma omp section
    structured_block
        .
        .
        .
}
```

with similar effect.

(In both cases, the `nowait` clause is not allowed.)

For Loop

```
#pragma omp for
for_loop
```

causes the for loop to be divided into parts and parts shared among threads in the team. The for loop must be of a simple form.

Way that for loop divided can be specified by an additional “schedule” clause.

Example: the clause schedule (static, chunk_size) cause for loop be divided into sizes specified by chunk_size and allocated to threads in a round robin fashion.

Example

```
#pragma omp parallel shared(a,b,c,nthreads,chunk) private(i,tid) {
    tid = omp_get_thread_num();

    if (tid == 0) {
        nthreads = omp_get_num_threads();
        printf("Number of threads = %d\n", nthreads);
    }
    printf("Thread %d starting...\n",tid);

 #pragma omp for schedule(dynamic,chunk)
    for (i=0; i<N; i++) {
        c[i] = a[i] + b[i];
        printf("Thread %d: c[%d]= %f\n",tid,i,c[i]);
    }

}  /* end of parallel section */
```

Combined Parallel Work-sharing Constructs

If a parallel directive is followed by a single for directive, it can be combined into:

```
#pragma omp parallel for
    for_loop
```

with similar effects, i.e. it has the effect of each thread executing the same for loop.

Loop Scheduling and Partitioning

OpenMP offers scheduling clauses to add to for construct:

- Static

```
#pragma omp parallel for schedule (static,chunk_size)
```

Partitions loop iterations into equal sized chunks specified by chunk_size. Chunks assigned to threads in round robin fashion.

- Dynamic

```
#pragma omp parallel for schedule (dynamic,chunk_size)
```

Uses internal work queue. Chunk-sized block of loop assigned to threads as they become available.

- Guided

    #pragma omp parallel for schedule (guided,chunk_size)

Similar to dynamic but chunk size starts large and gets smaller to reduce time threads have to go to work queue.

$$\text{chunk size} = \left\lceil \frac{\text{number of iterations remaining}}{2 * \text{number of threads}} \right\rceil$$

- Runtime

    #pragma omp parallel for schedule (runtime)

Uses OMP_SCEDULE environment variable to specify which of static, dynamic or guided should be used.

SCHEDULE: Describes how iterations of the loop are divided among the threads in the team. The default schedule is implementation dependent.

STATIC

Loop iterations are divided into pieces of size chunk and then statically assigned to threads. If chunk is not specified, the iterations are evenly (if possible) divided contiguously among the threads.

DYNAMIC

Loop iterations are divided into pieces of size chunk, and dynamically scheduled among the threads; when a thread finishes one chunk, it is dynamically assigned another. The default chunk size is 1.

GUIDED

Iterations are dynamically assigned to threads in blocks as threads request them until no blocks remain to be assigned. Similar to DYNAMIC except that the block size decreases each time a parcel of work is given to a thread. The size of the initial block is proportional to: number_of_iterations / number_of_threads Subsequent blocks are proportional to number_of_iterations_remaining / number_of_threads The chunk parameter defines the minimum block size. The default chunk size is 1.

RUNTIME

The scheduling decision is deferred until runtime by the environment variable OMP_SCHEDULE. It is illegal to specify a chunk size for this clause.

AUTO

The scheduling decision is delegated to the compiler and/or runtime system.

https://computing.llnl.gov/tutorials/openMP/

Source: https://software.intel.com/en-us/articles/openmp-loop-scheduling

Example

#pragma omp parallel for schedule(kind [,chunk size])

Four different loop scheduling types (kinds) can be provided to OpenMP, as shown in the following table. The optional parameter (chunk), when specified, must be a positive integer.

Kind	Description
static	Divide the loop into equal-sized chunks or as equal as possible in the case where the number of loop iterations is not evenly divisible by the number of threads multiplied by the chunk size. By default, chunk size is loop_count/number_of_threads.Set chunk to 1 to interleave the iterations.
dynamic	Use the internal work queue to give a chunk-sized block of loop iterations to each thread. When a thread is finished, it retrieves the next block of loop iterations from the top of the work queue. By default, the chunk size is 1. Be careful when using this scheduling type because of the extra overhead involved.
guided	Similar to dynamic scheduling, but the chunk size starts off large and decreases to better handle load imbalance between iterations. The optional chunk parameter specifies them minimum size chunk to use. By default the chunk size is approximately loop_count/number_of_threads.
auto	When schedule (auto) is specified, the decision regarding scheduling is delegated to the compiler. The programmer gives the compiler the freedom to choose any possible mapping of iterations to threads in the team.
runtime	Uses the OMP_schedule environment variable to specify which one of the three loop-scheduling types should be used. OMP_SCHEDULE is a string formatted exactly the same as would appear on the parallel construct.

Example: OpenMP

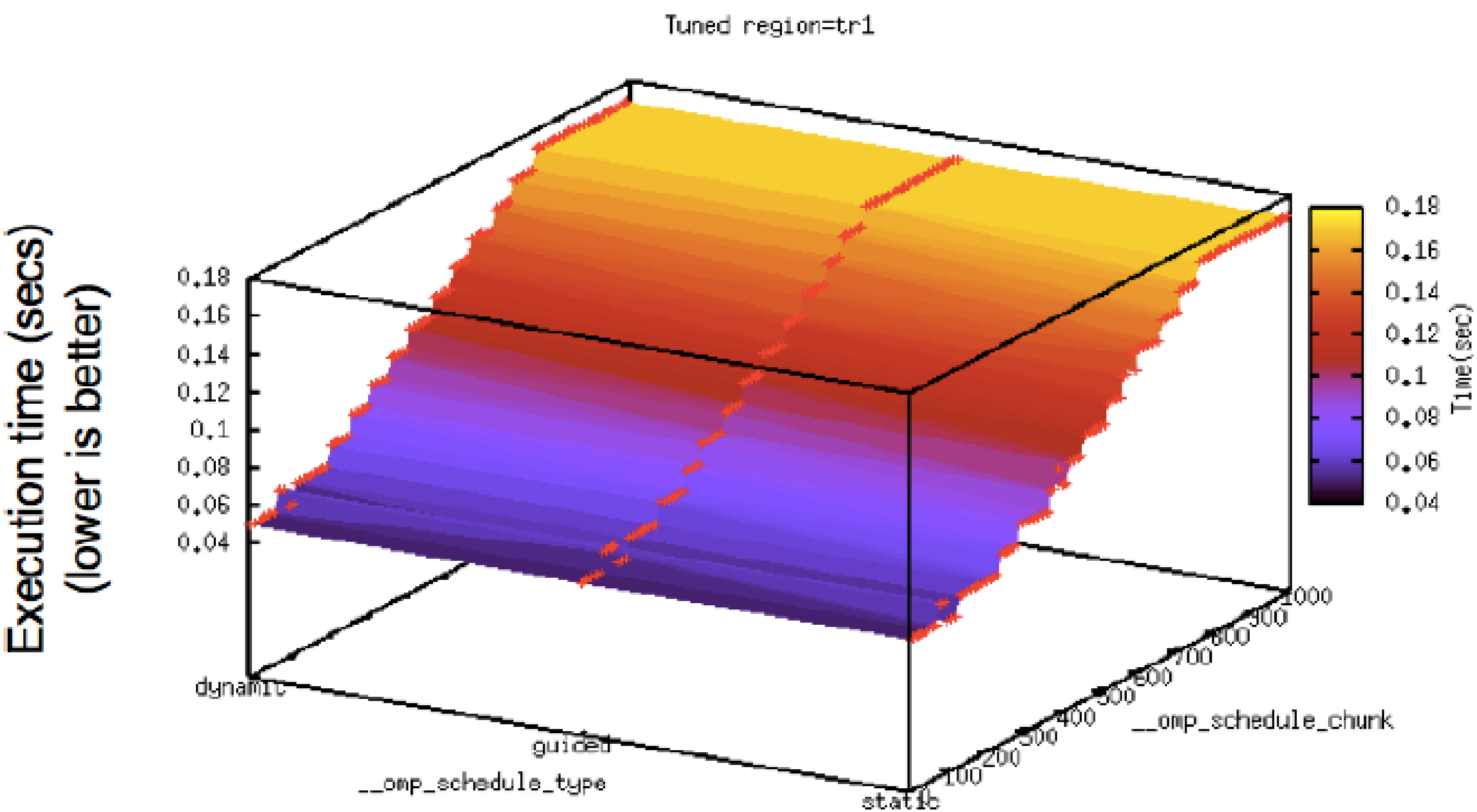
```
#pragma isat tuning
variable(@omp_schedule_type, [static, dynamic, guided])
variable(@omp_schedule_chunk, range(0, 1000, 100))
search(dependent)

#pragma omp parallel for
for(i = 0; i < N; i++)
    C[i] = A[i] * B[i];
```

- Two tunable parameters: schedule type and chunk size

source: http://www.ckluk.org/ck/talks/cilkplus-tutorial-ppopp11.pdf

Visualization of Result

Tuning the schedule for `parellel_for`



source: http://www.ckluk.org/ck/talks/cilkplus-tutorial-ppopp11.pdf

Reduction clause

Used combined the result of the iterations into a single value c.f. with MPI _Reduce().

Can be used with parallel, for, and sections,

Operation

Variable

Example

```
sum = 0
#pragma omp parallel for reduction(+:sum)
for (k = 0; k < 100; k++ ) {
    sum = sum + funct(k);
}
```

Private copy of sum created for each thread by complier.
Private copy will be added to sum at end.
Eliminates here the need for critical sections.

Private variables

private clause – creates private copies of variables for each thread

firstprivate clause - as private clause but initializes each copy to the values given immediately prior to parallel construct.

lastprivate clause – as private but “the value of each lastprivate variable from the sequentially last iteration of the associated loop, or the lexically last section directive, is assigned to the variable’s original object.”

Synchronization Constructs
Critical

The critical directive will only allow one thread execute the associated structured block. When one or more threads reach the critical directive:

```
#pragma omp critical name
    structured_block
```

they will wait until no other thread is executing the same critical section (one with the same name), and then one thread will proceed to execute the structured block. name is optional. All critical sections with no name map to one undefined name.

Barrier

When a thread reaches the barrier

```
#pragma omp barrier
```

it waits until all threads have reached the barrier and then they all proceed together.

There are restrictions on the placement of barrier directive in a program. In particular, all threads must be able to reach the barrier.

Atomic

The atomic directive

```
#pragma omp atomic
 expression_statement
```

implements a critical section efficiently when the critical section simply updates a variable (adds one, subtracts one, or does some other simple arithmetic operation as defined by expression_statement).

ensures the serialisation of a particular operation and its much faster (less overhead)

More information

Full information on OpenMP at

http://openmp.org/wp/

Guy: **Dependency Analysis**
Next two slides taken from
"slides8d.ppt"

Bernstein's Conditions

Set of conditions sufficient to determine whether two processes can be executed simultaneously. Given:

I_i is the set of memory locations read (input) by process P_i.

O_j is the set of memory locations written (output) by process P_j.

For two processes P_1 and P_2 to be executed simultaneously, inputs to process P_1 must not be part of outputs of P_2, and inputs of P_2 must not be part of outputs of P_1; i.e.,

$$I_1 \cap O_2 = \phi$$

$$I_2 \cap O_1 = \phi$$

where ϕ is an empty set. Set of outputs of each process must also be different; i.e.,

$$O_1 \cap O_2 = \phi$$

If the three conditions are all satisfied, the two processes can be executed concurrently.

Example

Suppose the two statements are (in C)

```
a = x + y;
b = x + z;
```

We have

I_1 = (x, y) $\qquad$ O_1 = (a)

I_2 = (x, z) $\qquad$ O_2 = (b)

and the conditions

$$I_1 \cap O_2 = \phi$$
$$I_2 \cap O_1 = \phi$$
$$O_1 \cap O_2 = \phi$$

are satisfied. Hence, the statements `a = x + y` and `b = x + z` can be executed simultaneously.