Chapter 8

# Programming with Shared Memory

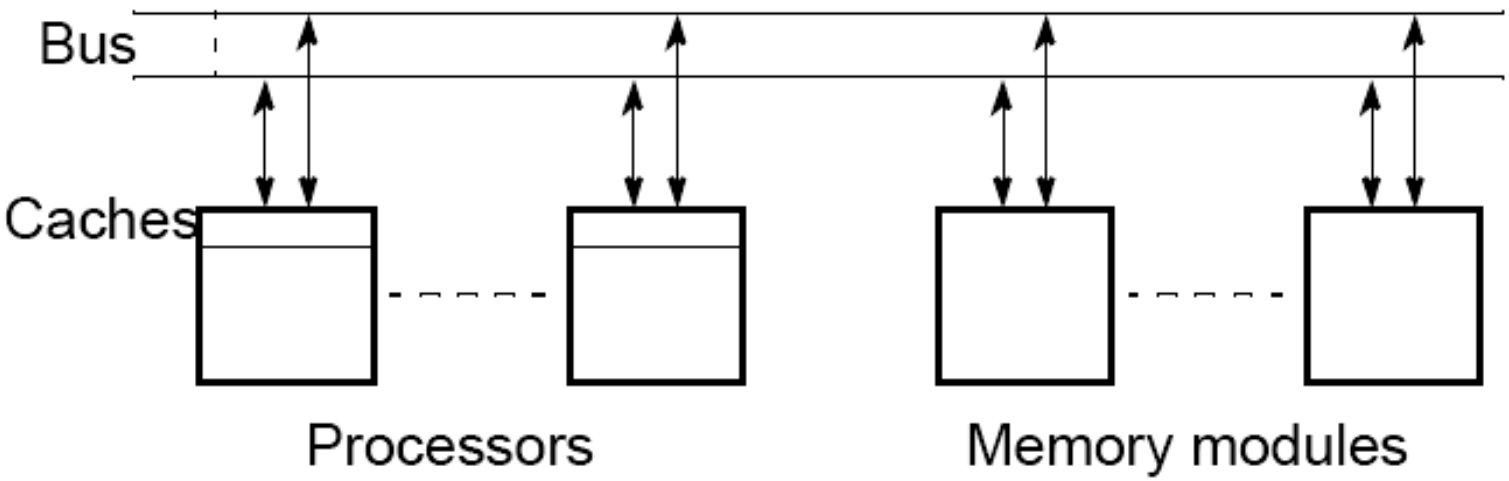
# Shared memory multiprocessor system

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

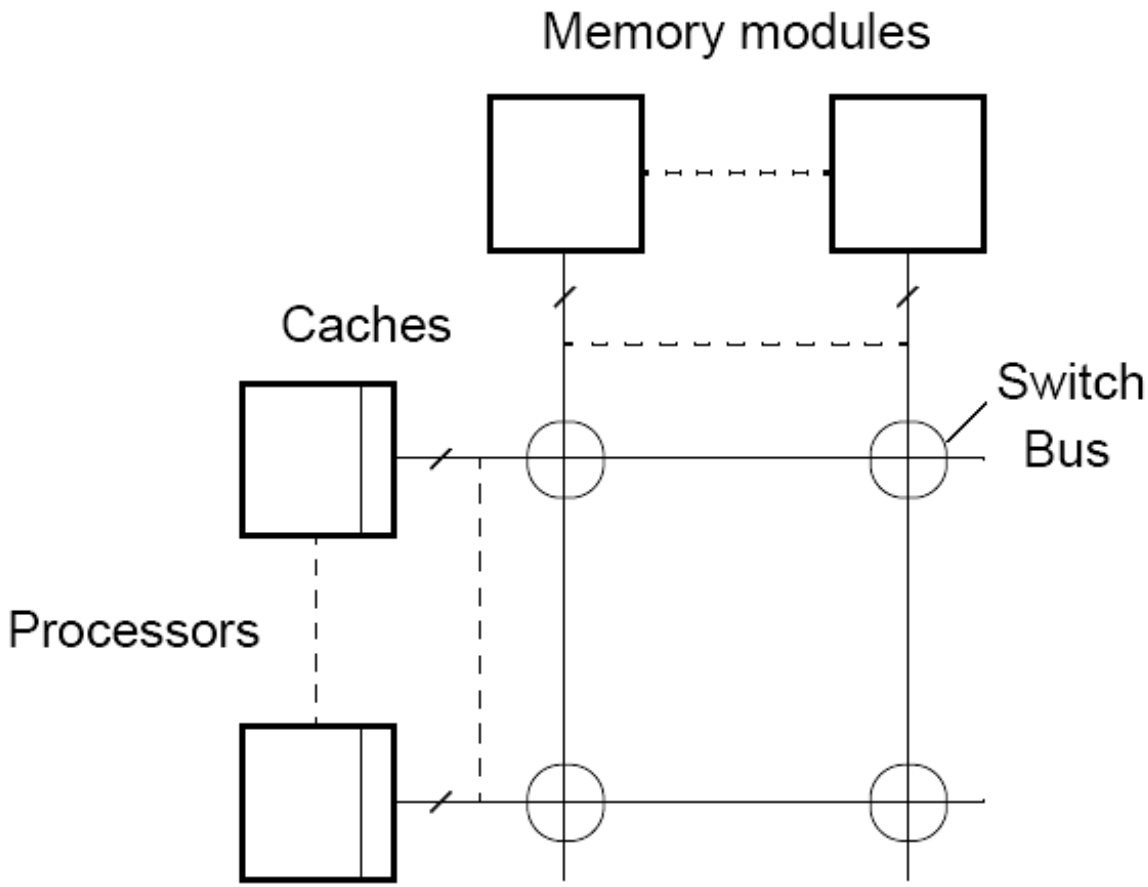
A *single address space* exists, meaning that each memory location is given a unique address within a single range of addresses.

Generally, shared memory programming more convenient although it does require access to shared data to be controlled by the programmer (using critical sections etc.)

# Shared memory multiprocessor using a single bus



# Shared memory multiprocessor using a crossbar switch



# Alternatives for Programming Shared Memory Multiprocessors:

- Using heavy weight processes.
- Using threads. Example Pthreads
- Using a completely new programming language for parallel programming - not popular. Example Ada.
- Using library routines with an existing sequential programming language.
- Modifying the syntax of an existing sequential programming language to create a parallel programing language. Example UPC
- Using an existing sequential programming language supplemented with compiler directives for specifying parallelism. Example OpenMP

# Using Heavyweight Processes

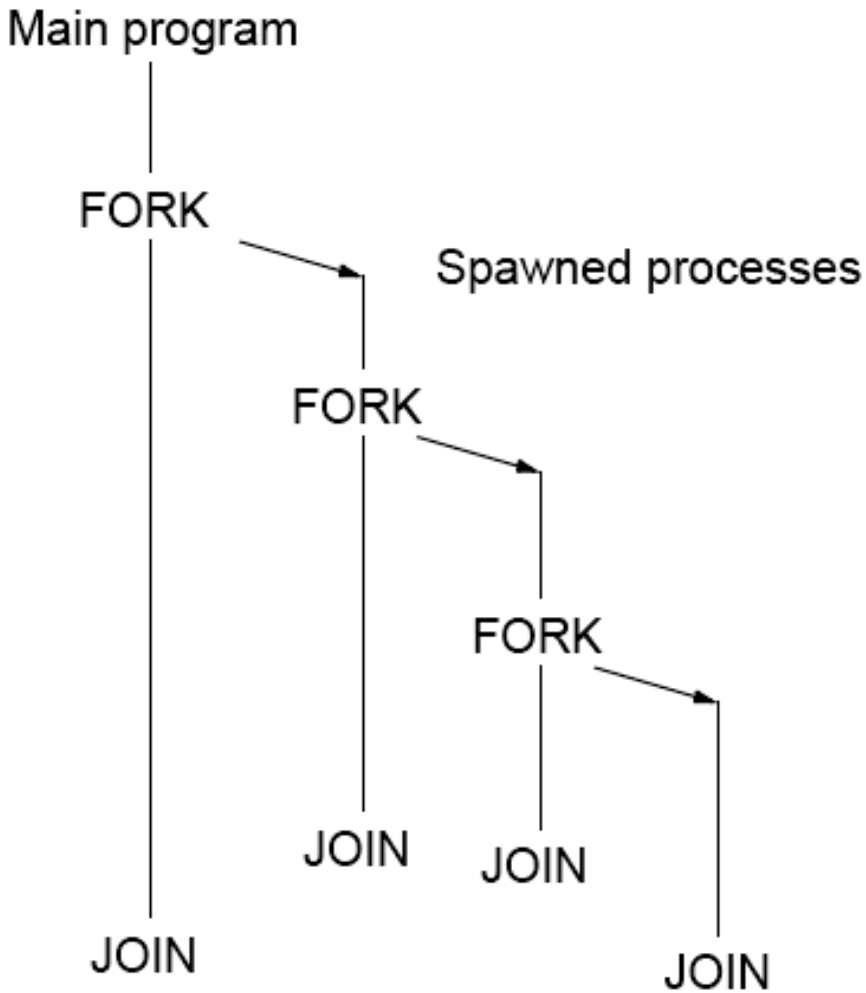
Operating systems often based upon notion of a process.

Processor time shares between processes, switching from one process to another. Might occur at regular intervals or when an active process becomes delayed.

Offers opportunity to deschedule processes blocked from proceeding for some reason, e.g. waiting for an I/O operation to complete.

Concept could be used for parallel programming. Not much used
because of overhead but fork/join concepts used elsewhere.

# 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 */
    .
    .
    .
```

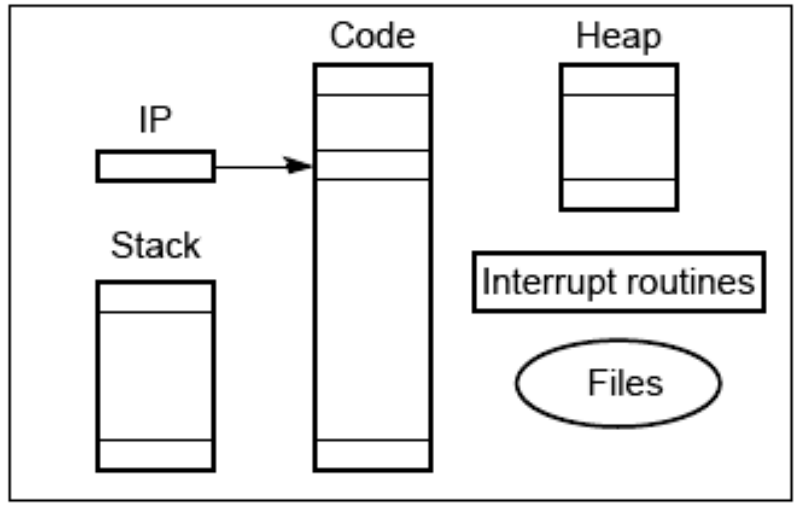
# UNIX System Calls

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

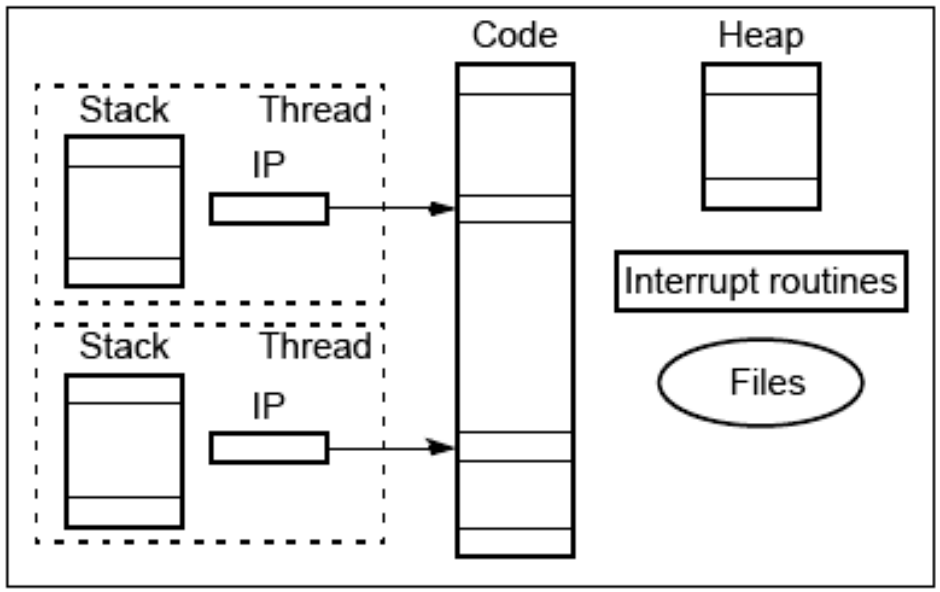
# Differences between a process and threads

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




(a) Process

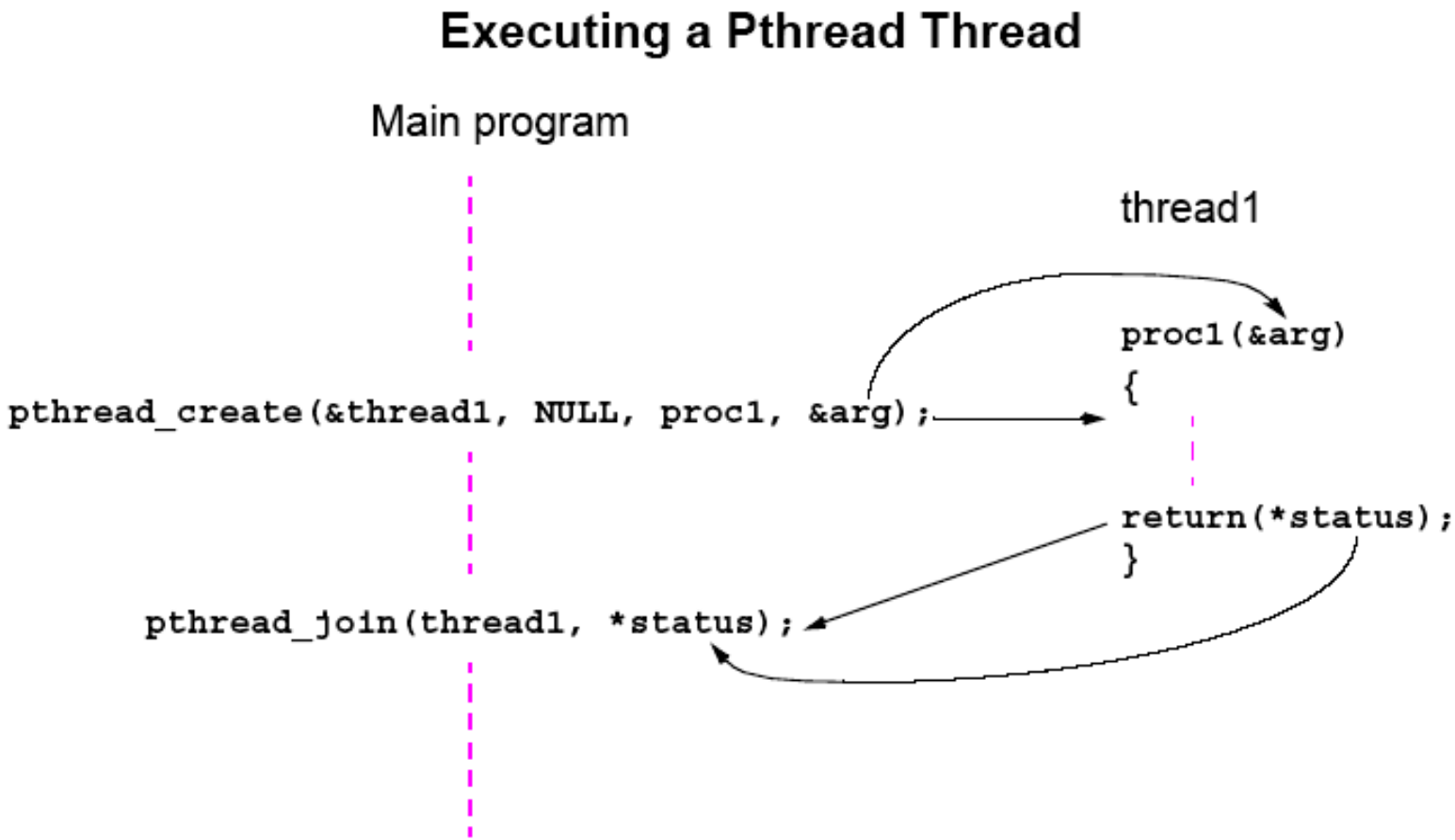
Threads - shares the same memory space and global variables between routines.




(b) Threads

# Pthreads

IEEE Portable Operating System Interface, POSIX, sec. 1003.1 standard



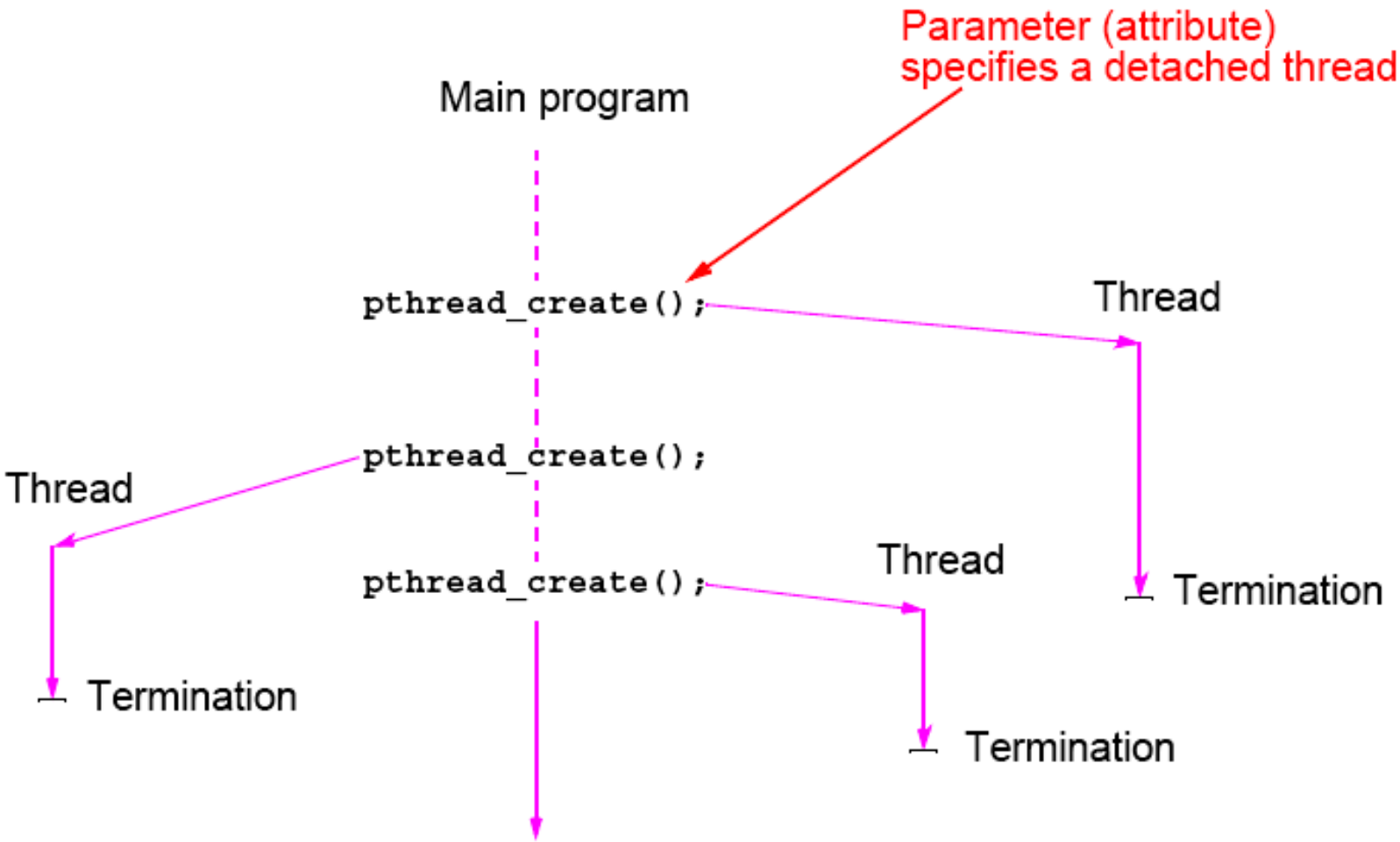
# Detached Threads

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 |

Several possible orderings, including

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.

If two processes were to print messages, for example, the messages could appear in different orders depending upon the scheduling of processes calling the print routine.

Worse, the individual characters of each message could be interleaved if the machine instructions of instances of the print routine could be interleaved.

# Compiler/Processor Optimizations

Compiler and processor reorder instructions for optimization.

**Example**

The statements

```
a = b + 5;
x = y + 4;
```

could be compiled to execute in reverse order:

```
x = y + 4;
a = b + 5;
```

and still be logically correct.

May be advantageous to delay statement a = b + 5 because a previous instruction currently being executed in processor needs more time to produce the value for b. Very common for processors to execute machines instructions out of order for increased speed .

# 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).

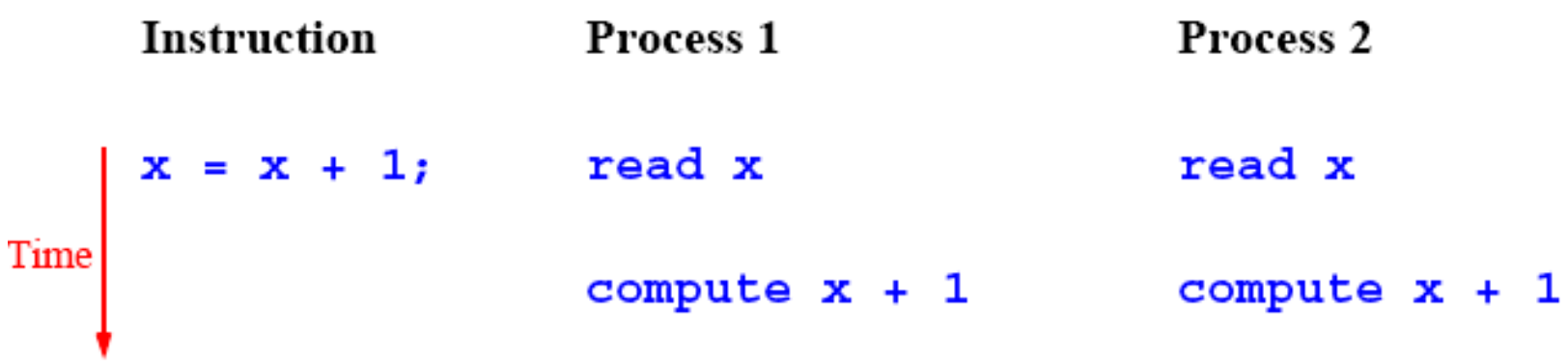
System routines that return time may not be thread safe.

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

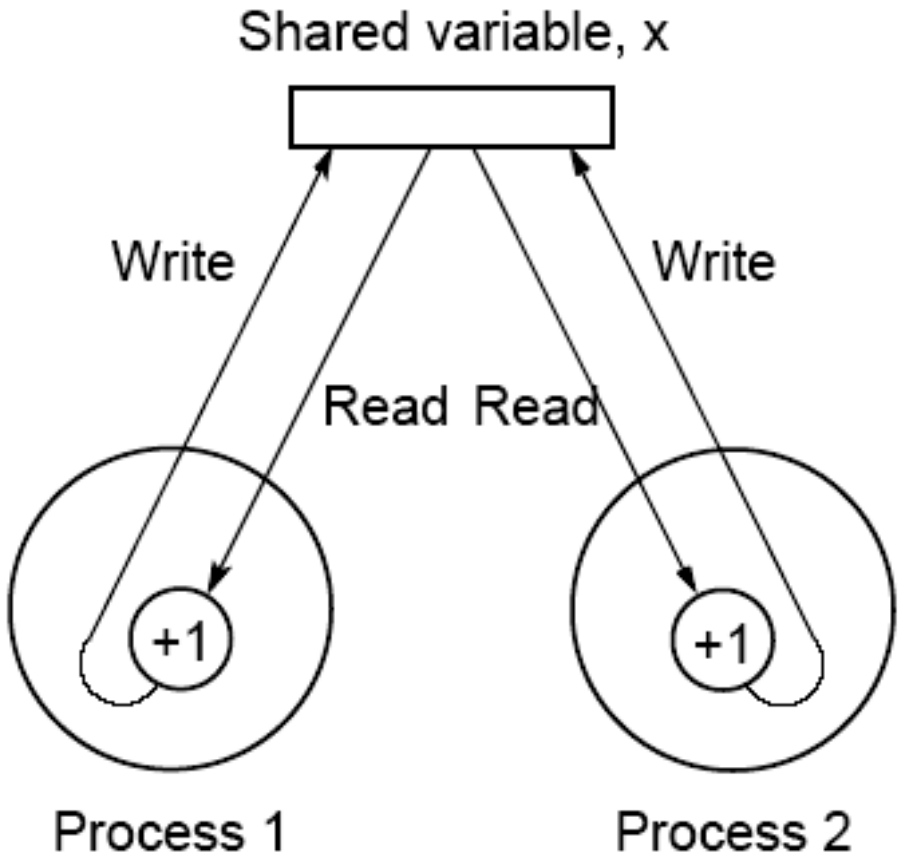
# Accessing Shared Data

Accessing shared data needs careful control.

Consider two processes each of which is to add one to a shared data item, x. Necessary for the contents of the location x to be read, x + 1 computed, and the result written back to the location:



# Conflict in accessing shared variable



# Critical Section

A mechanism for ensuring that only one process accesses a particular resource at a time is to establish sections of code involving the resource as so-called *critical sections* and arrange that only one such critical section is executed at a time

This mechanism is known as *mutual exclusion*.

This concept also appears in an operating systems.

# Locks

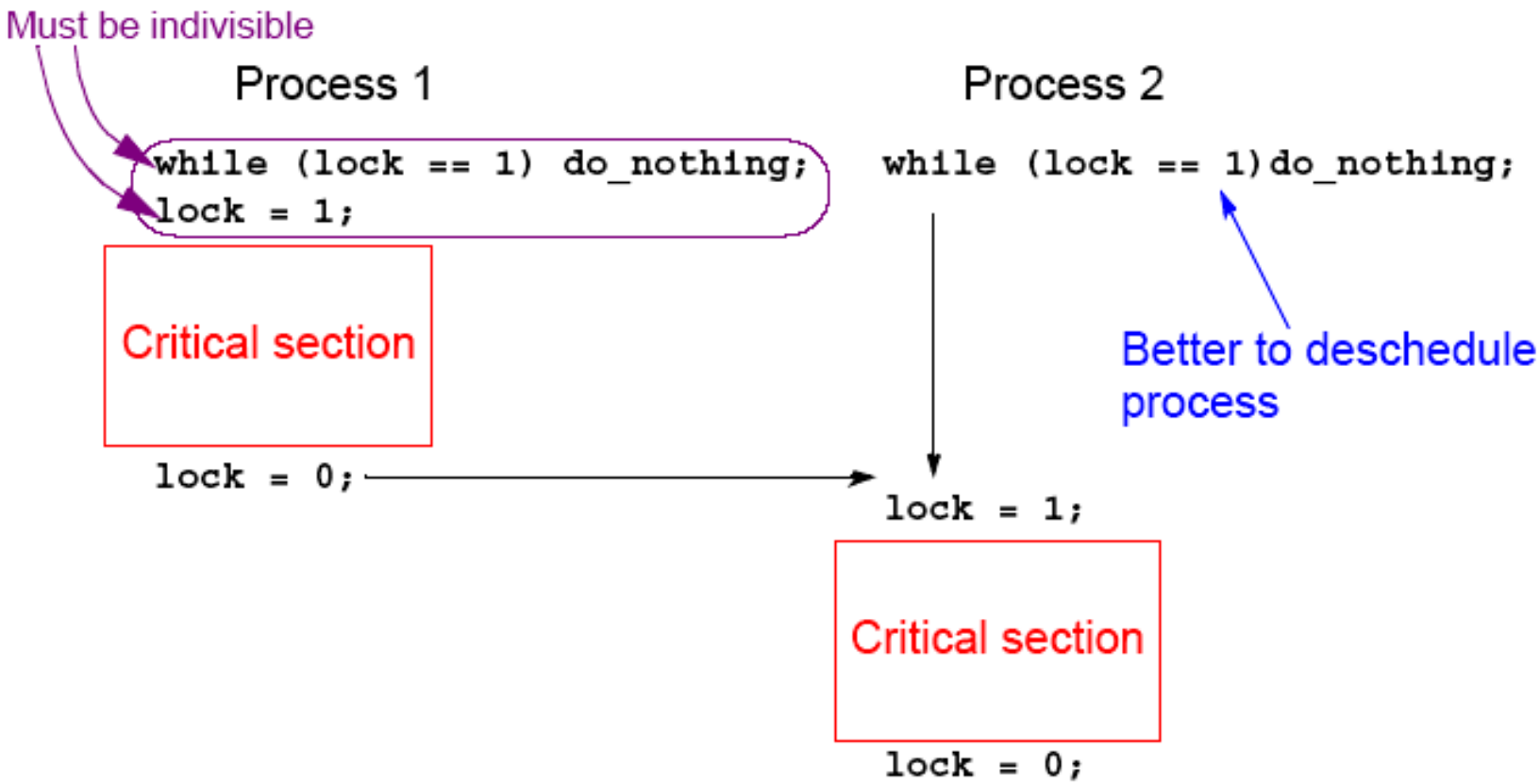
Simplest mechanism for ensuring mutual exclusion of critical sections.

A lock is 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



# Pthread Lock Routines

Locks are implemented in Pthreads with *mutually exclusive lock* variables, or "mutex" variables:

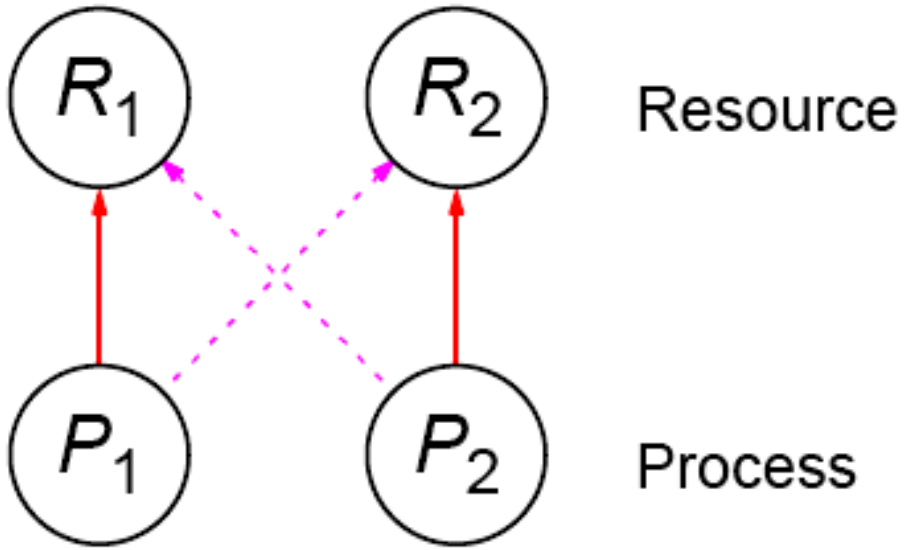
```
.
pthread_mutex_lock(&mutex1);
    critical section
pthread_mutex_unlock(&mutex1);
.
```

If a thread reaches a mutex lock and finds it locked, it will wait for the lock to open. If more than one thread is waiting for the lock to open when it opens, the system will select one thread to be allowed to proceed. Only the thread that locks a mutex can unlock it.

# Deadlock

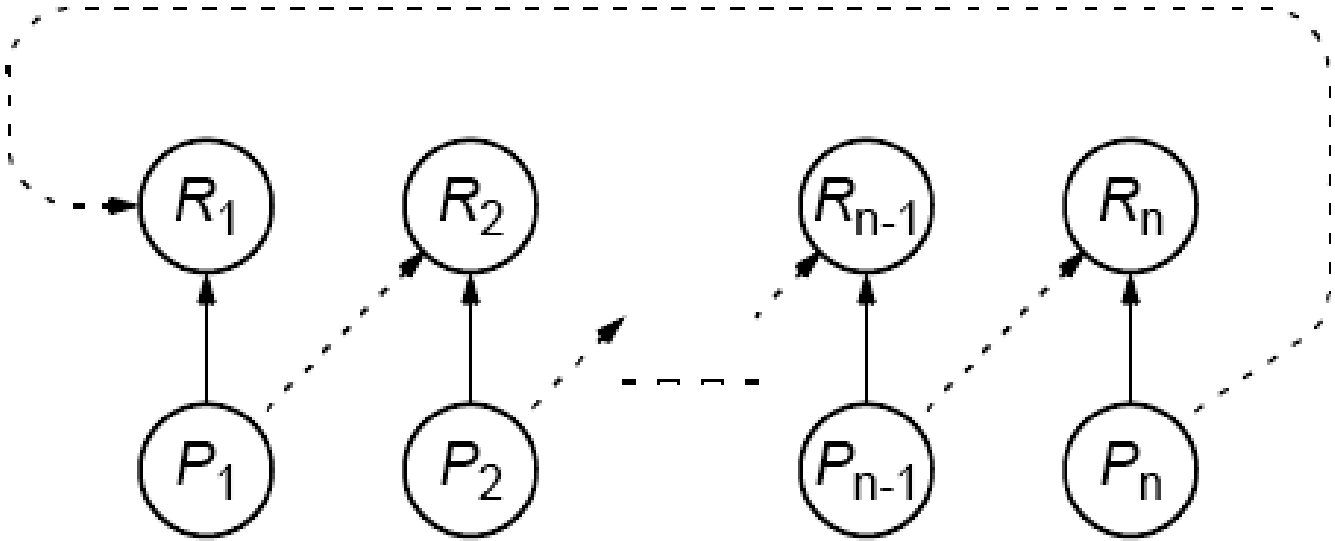
Can occur with two processes when one requires a resource held by the other, and this process requires a resource held by the first process.

Two-process deadlock



# Deadlock (deadly embrace)

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



# Pthreads

Offers one routine that can test whether a lock is actually closed without blocking the thread:

**pthread_mutex_trylock()**

Will lock an unlocked mutex and return 0 or will return with **EBUSY** if the mutex is already locked – might find a use in overcoming deadlock.

# 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++.

The compiler directives can specify such things as the par and forall operations described previously.

Several OpenMP compilers available.

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 (a statement or statements) that follows the directive and then the directive and structured block form a “construct”.

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.

# 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.

# 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.

Note that these constructs do not start a new team of threads. That done by an enclosing parallel construct.

# 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.

# 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 the for loop be divided into sizes specified by chunk_size and allocated to threads in a round robin fashion.

# Single

The directive

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

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

# 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.

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.)

# 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.

# 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).

# Flush

A synchronization point which causes thread to have a "consistent" view of certain or all shared variables in memory. All current read and write operations on the variables allowed to complete and values written back to memory but any memory operations in the code after flush are not started, thereby creating a "memory fence". Format:

```
#pragma omp flush (variable_list)
```

Only applied to thread executing flush, not to all threads in the team.

Flush occurs automatically at the entry and exit of parallel and critical directives (and combined parallel for and parallel sections directives), and at the exit of for, sections, and single (if a no-wait clause is not present).

# Ordered

Used in conjunction with for and parallel for directives to cause an iteration to be executed in the order that it would have occurred if written as a sequential loop.

See Appendix C of textbook for further details.

# Shared Memory Programming Performance Issues

# Shared Data in Systems with Caches

All modern computer systems have cache memory, high-speed memory closely attached to each processor for holding recently referenced data and code.

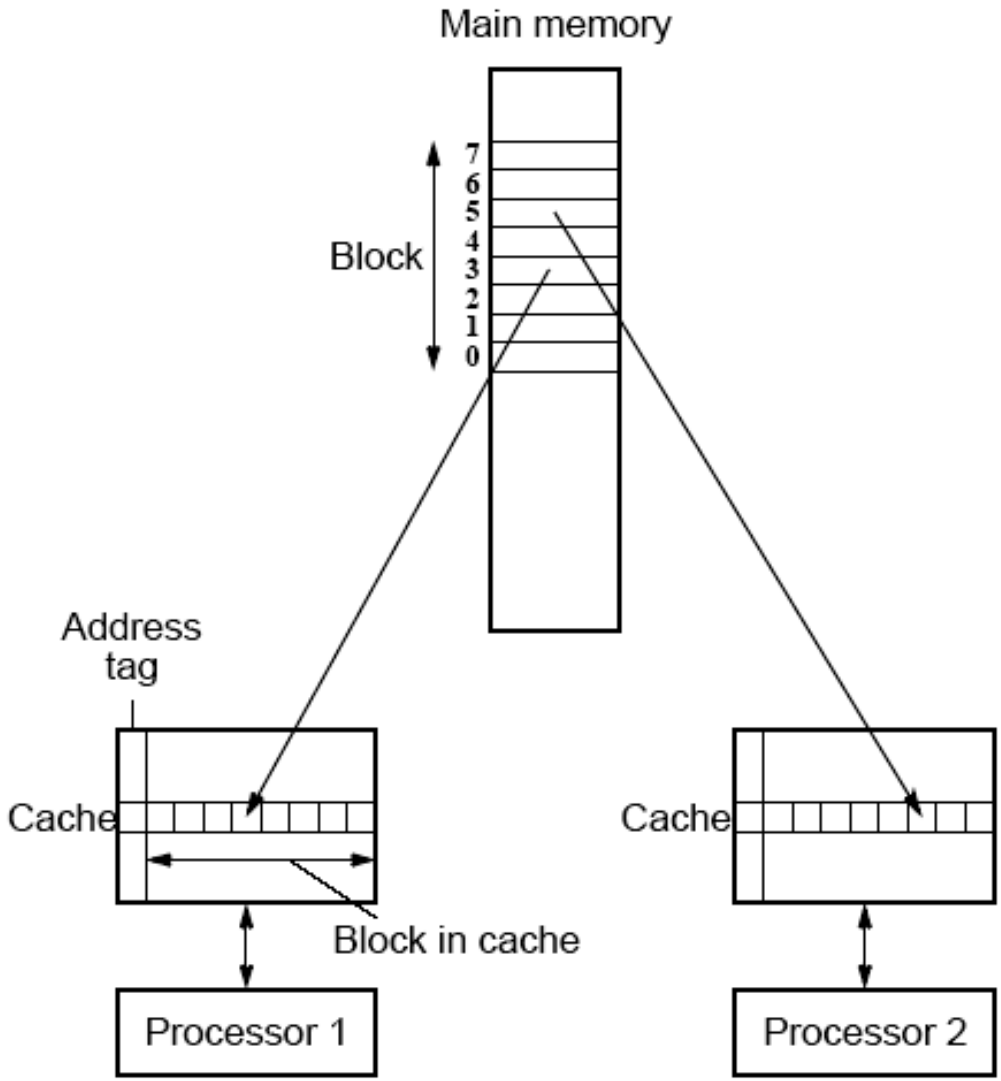
## Cache coherence protocols

*Update policy* - copies of data in all caches are updated at the time one copy is altered.

I*nvalidate policy* - when one copy of data is altered, the same data in any other cache is invalidated (by resetting a valid bit in the cache). These copies are only updated when the associated processor makes reference for it.

# False Sharing

Different parts of block required by different processors but not same bytes. If one processor writes to one part of the block, copies of the complete block in other caches must be updated or invalidated though the actual data is not shared.



# Solution for False Sharing

Compiler to alter the layout of the data stored in the main memory, separating data only altered by one processor into different blocks.

# Critical Sections Serializing Code

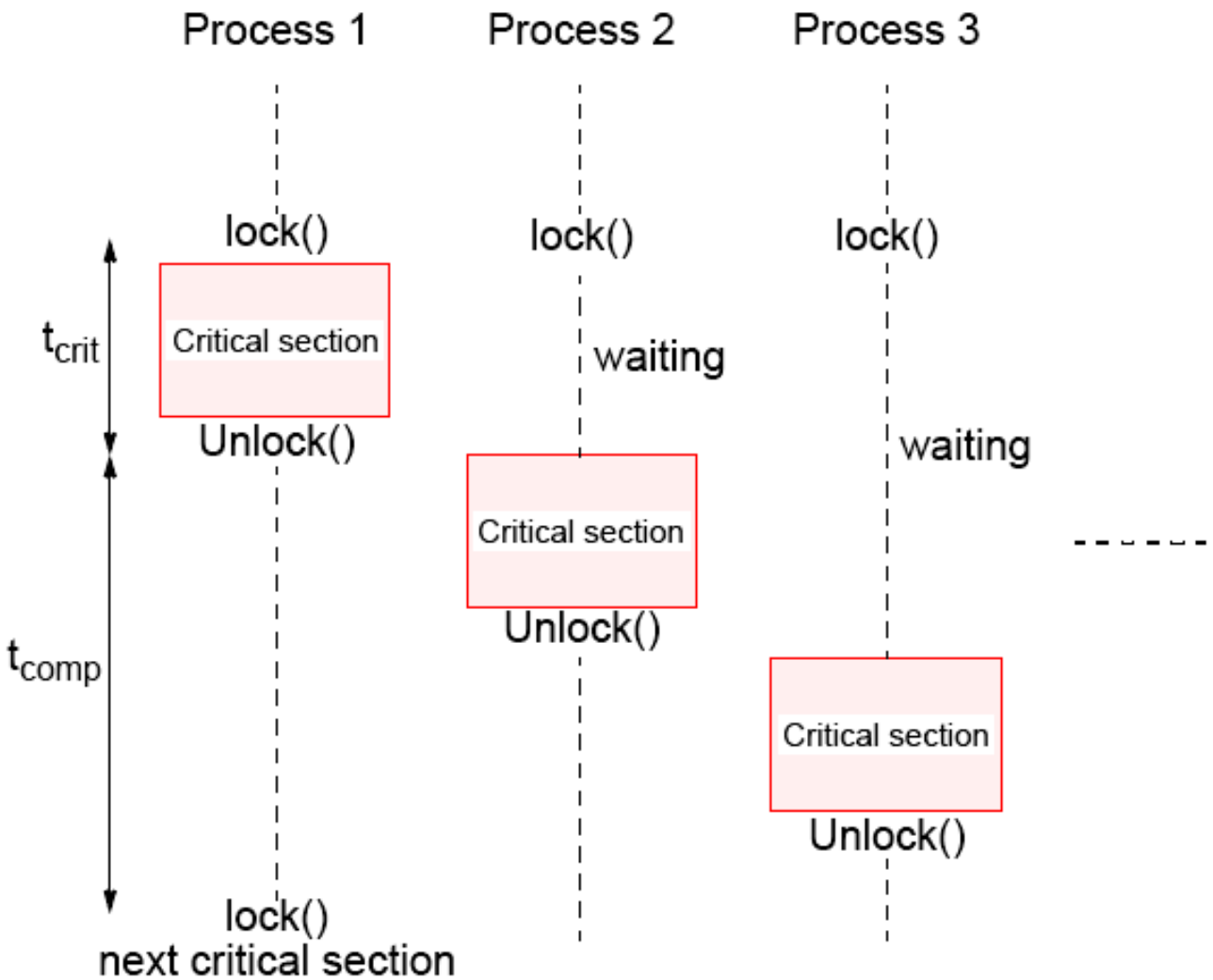
High performance programs should have as few as possible critical sections as their use can serialize the code.

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