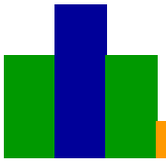


Lesson 8: Programming with shared memory

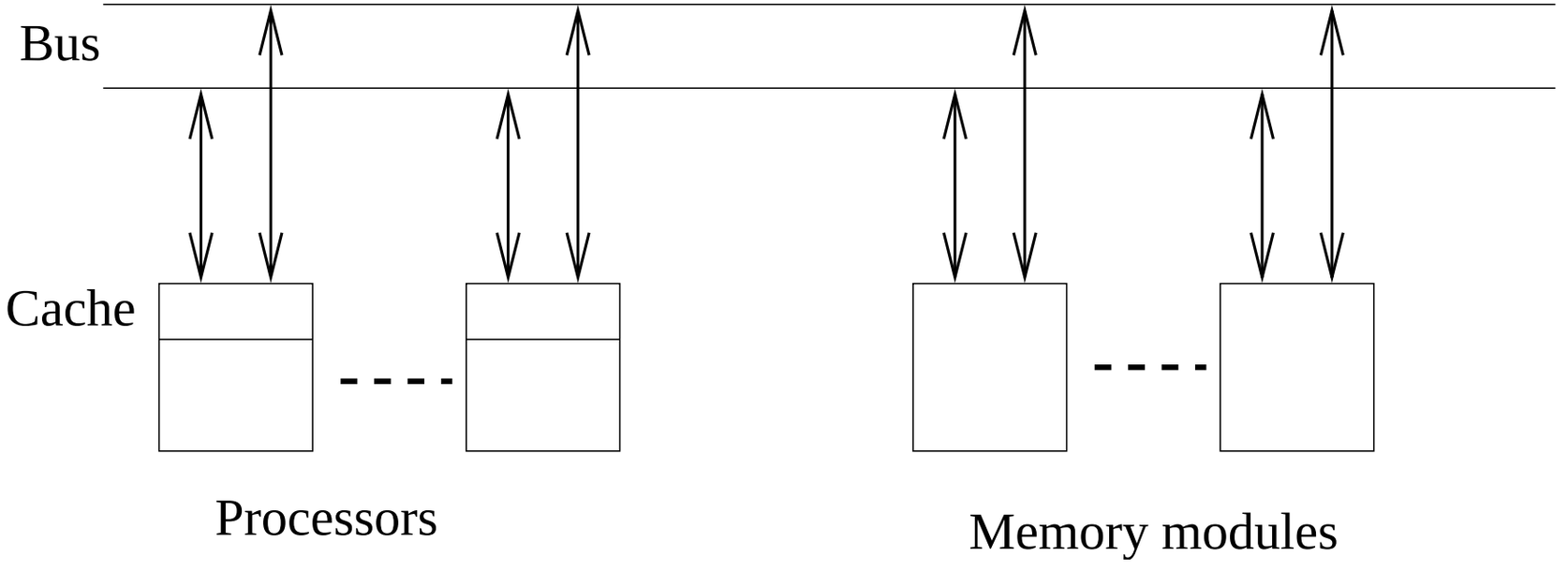
G Stefanescu — University of Bucharest

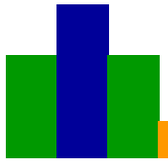
Parallel & Concurrent Programming
Fall, 2014

Shared memory model

Shared memory multiprocessor system:

- *any* memory location is *directly accessible* by any processor (not via message passing)
- there is a *single address space*
- example: a common architecture is the *single bus architecture* (good for a small number of processors, e.g. ≤ 8)





Parallel programming

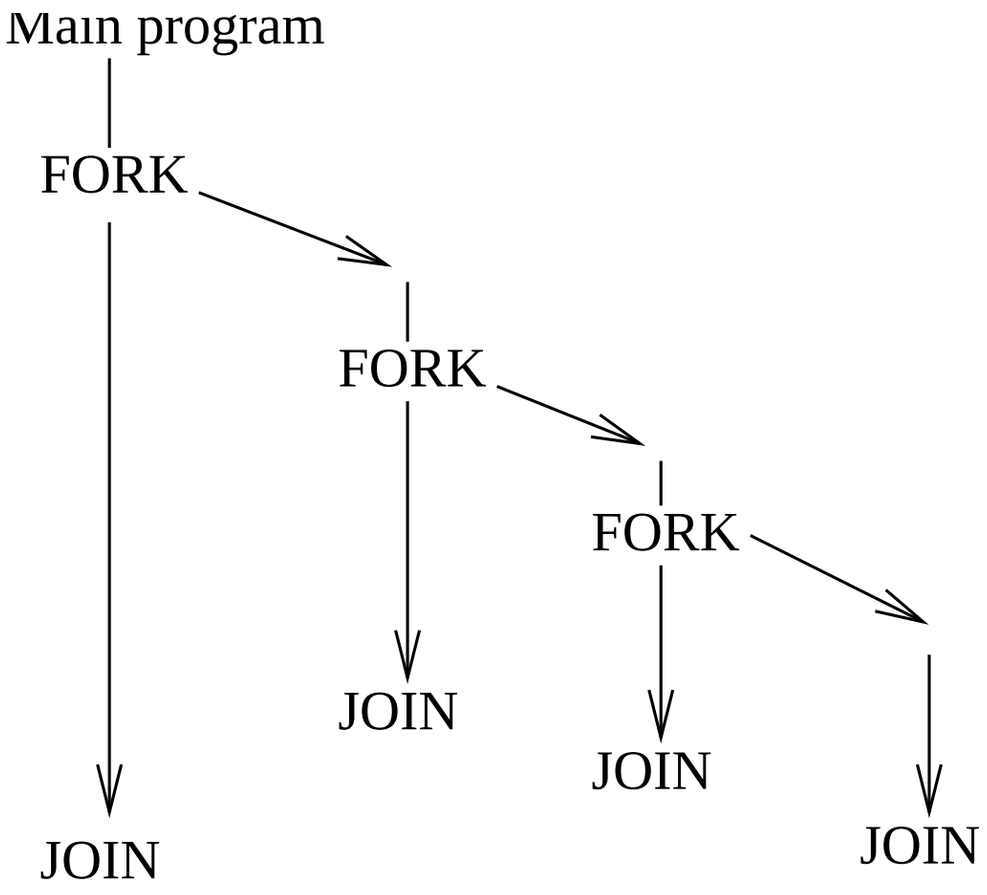
To our initial list of parallel programming alternatives

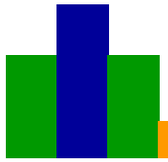
- use *new* programming languages
- *modify existing* sequential ones
- use *library* routines
- use sequential programs with *parallelizing compilers*

one may add

- *Unix processes* and
- *Threads* (Pthreads, Java, ...)

Fork-Join construct

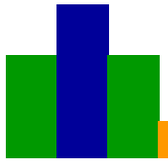




Unix (heavyweight) processes

A *fork* in Unix is obtained using system call `fork()`:

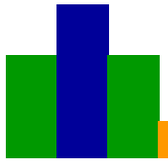
- the new process is an *exact copy* of the parent, except that it has its own process ID
- it has the same variables, whose initial values are the (current) values of the parent process
- the child process start execution from the point of the fork
- if successful,
  —`fork()` returns 0 to the child and child ID to the parent;
  if not,
  —return -1 to parent and no child process is created
- processes are joined using `wait()` and `exit()`
  —`wait(statusp)` - delay the caller
  —`exit(status)` - terminates a process



Fork in Unix: example

Simple example of using Unix `fork`:

```
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);
```



Threads

Threads vs. (heavyweight) processes:

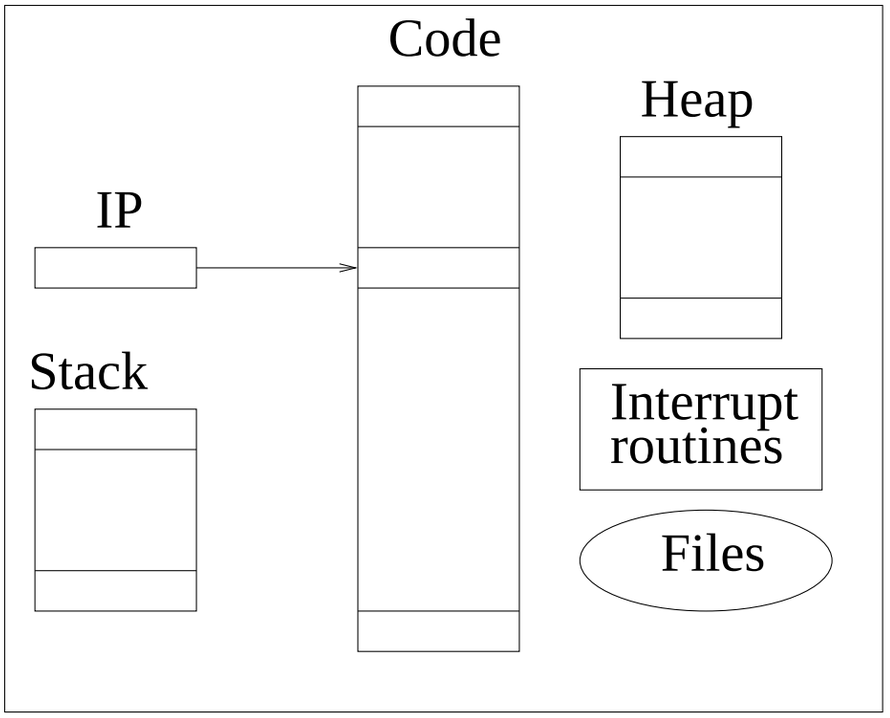
- heavyweight *processes* are completely *separate* programs (with their own variables, stack, memory allocation)
- *threads share* the same memory space and global variables (but they have their own stack)

Threads' advantages:

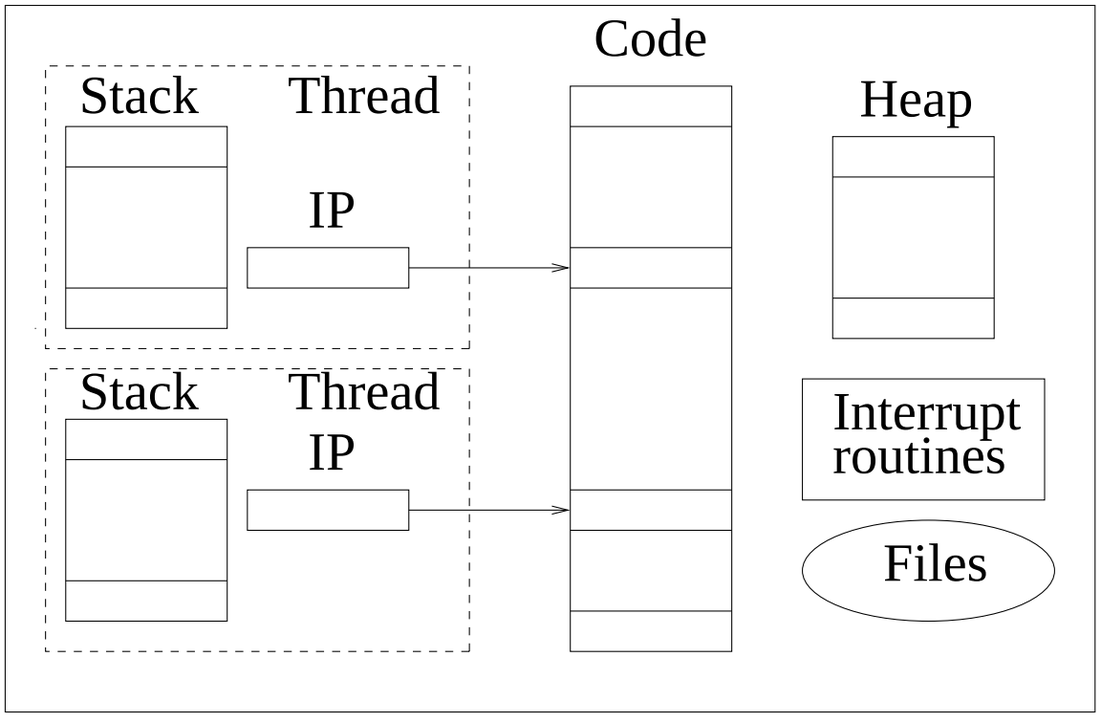
- thread *creation* can take three orders of magnitude less time than process creation
- *communication* (via shared memory) and *synchronization* are also faster than in the case of processes

Sometimes there is no need to join a child thread with its parent; such threads are called *detached threads*.

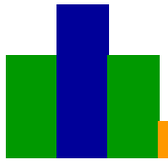
Threads



(a) Process



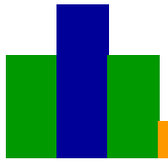
(b) Threads



Pthreads

Pthreads is an IEEE standard

```
Main program                                thread1
   ⋮
pthreads_create(&thread1,                   proc1(&arg){
    NULL, proc1, &args);
   ⋮                                           ⋮
pthreads_join(thread1, *status);            return(*status);
   ⋮                                        }
```

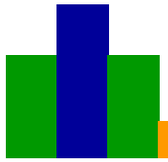


Thread barrier

A *barrier* may be defined in Pthreads as follows

```
for (i=0; i<n; i++)
  pthread_create(&thread[i], NULL,
        (void *) slave, (void *) &arg);
for (i=0; i<n; i++)
  pthread_join(thread[i], NULL);
```

Notice that `pthreads_join()` *waits for one specific thread to terminate. The current threads disappear after the barrier - if you want to reuse them after the barrier, you shoud recreate them.*



Statement execution order

Execution order:

- on a single processor system the statements of parallel processes are *interleaved*
- on a multiprocessor system it is a chance the have real parallel processes, but usually the number of processes is larger than that of processors, so *some interleaving* appear here, as well

Interleaving: Given two sequences

$$abc \text{ and } ABCD$$

any shuffle preserving the relative order of the letters from each string is possible, e.g.,

$$aABbCDc, AabcBCD, aAbBcCD, \ldots$$

..Execution order

Examples:

- *Printing*: when more processes print messages, their individual characters may be interleaved
- *Compiler optimization*: the order of some statements may be irrelevant; e.g.,

```
a := b*3;
x := sin(y);
```

and

```
x := sin(y);
a := b*3;
```

are equivalent; most of modern compilers/processors use such transformations to increase execution speed

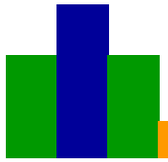
Thread-safe routines

Thread-safe routines are system calls or library routines that may be called from multiple threads simultaneously and always produce correct results

Examples

- standard I/O is print safe (no interleaving of characters)
- accessing shared/static data may be thread-safe, but some care is needed
- system routines that return time may be not thread safe

A general solution to thread-safeness is to *enclose* such routines into *critical section* (see forthcoming slides), but this is very inefficient



Accessing sharing data

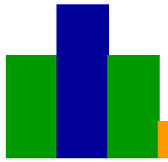
Suppose `x` is a shared variable initially equal to 0. What is the result of running in parallel

```
Process 1:          Process 2:
x = x+1             x = x+1
```

?

The answer is: It depends...

- if the assignment statement is *atomic*, then the result is the expected one 2



..Accessing sharing data

- but if the assignment `x = x+1` is *not atomic*, e.g., it is replaced by `read x; compute x+1; write x`

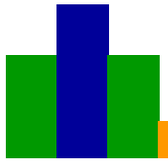
then the program becomes

```
Process 1:              Process 2:
read x;                 read x;
compute x+1;            compute x+1;
write x;                write x;
```

with unexpected results, i.e.,

$r_1 c_1 w_1 r_2 c_2 w_2 \rightarrow 2$ (ok), but
$r_1 r_2 c_1 c_2 w_1 w_2 \rightarrow 1$!

where r_1, c_1, w_1 (resp. r_2, c_2, w_2) means read/compute/write in process 1 (resp. 2)



Critical section

We need a mechanism to ensure that only one process access a particular resource at any time; the section of code dealing with the access to such a resource is called *critical section*.

The mechanism has to obey a few conditions:

- a first process reaching a critical section for a particular resource *enters and executes* the critical section
- at the same time, it *prevents all other processes* from entering to access the some resource in their critical sections
- when the process has finished its critical section, *another* process is allowed to enter its critical section for the resource

This mechanism is called *mutual exclusion*.

Locks

Locks provide the simplest mechanism for ensuring mutual exclusion of critical sections:

- a lock is a 0-1 variable that is
  —1 to indicate that *a process* has *entered* the critical section and
  —0 to indicate that *no process* is *in* the critical section

- it operates much like a door lock:
  —a process *coming to the door* of a critical resource and finding the *door open* may *enter* the critical section, *locking* the door behind to prevent other processes from entering;
  —when the process has *finished* the critical section, it *unlocks the door* and leaves

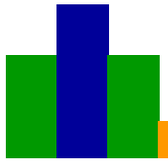
..Locks

Spin lock:

- a process trying to enter the critical section may *check continuously* the lock *doing nothing*; such a lock is called *spin lock* and the waiting mechanism *busy waiting*
- quite *inefficient* as no useful work is done

Atomic checks of the looks:

- it is important to have an atomic lock checking: e.g.,
  —no two processes check the door and enter simultaneously;
  or
  —if a process find the door open no other process is able to check the door before the current process is able to close it



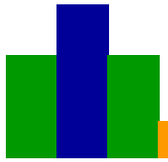
Pthreads locks

In Pthreads mutual exclusion is implemented using *mutex* variables; they are to be *declared* and initialized in the main program

```
pthread_mutex_t myMutex;
   :
pthread_mutex_intit(&myMutex, NULL);
```

and used as follows

```
pthread_mutex_lock(myMutex)
  critical section
pthread_mutex_unlock(myMutex);
```



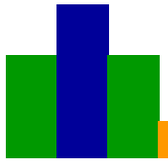
Deadlock

Deadlock may appear when processes need to access resources hold by other processes and *no process can progress*.

Example:

P1	holds	R1	and needs	R2
P2	holds	R2	and needs	R3
⋮				
P(n-1)	holds	R(n-1)	and needs	Rn
Pn	holds	Rn	and needs	R1

A Pthread solution is `pthread_mutex_trylock()`: it *tests* the lock *without blocking* the thread. (It locks an unlock mutex variable and return 0 or return EBUSY if the variable is locked.)



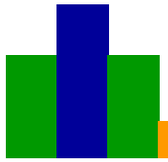
Semaphores

Dijskstra (1968) has introduced *semaphores*, as a way to control the access of critical sections. A semaphore is a positive integer s (initialized to 1) together with two operations

- $P(s)$ - waits until s is grater than 0 and then decrements s by one (wait to pass)
- $V(s)$ - increments s by one and release one of the waiting processes (release)

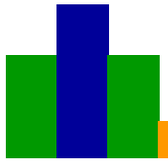
Each process has a sequence

$$\ldots \text{noncritical} \rightarrow P(s) \rightarrow \text{critical} \rightarrow V(s) \rightarrow \text{noncritical} \ldots$$



..Semaphores

- *binary semaphores* behave as locks, but P/V operations also provide a scheduling mechanism for process selection
- *general semaphores* (using arbitrary positive numbers) may be used to solve producer/consumer problems, keeping track on the number of free resources
- semaphores do *not* exists in *Pthreads*, but may be *found* in *Unix*
- semaphores are *powerful* enough to solve most of critical section problems, but they provides a very low-level mechanism which may be difficult to handle



Monitor

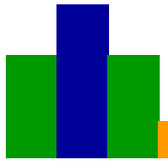
Hoare (1974) has introduced the notion of *monitor* which is

- a *higher-level* technique to control the access to critical sections, and
- encapsulates *data* and *access operations* in one structure

A monitor may be implemented using semaphores as follows

```
monitor_proc1{
  P(monitor_semaphore);
    monitor body
  V(monitor_semaphore);
  return;
}
```

Java has such "monitors".



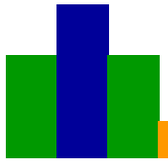
Condition variables

Problem:

- *entering* into a *critical* section may require *frequent checking* of certain (access) conditions
- *checking* them *continously* may be very *inefficient*

A proposed solution is to use *condition variables* (used, e.g., in the case of monitors):

- such a condition variable is accompanied with three operations
  —`Wait(condVar)` - *wait* for the condition to occur
  —`Signal(condVar)` - *signal* that the condition has occured
  —`Status(condVar)` - return the *number* of processes *waiting* for the condition to occur

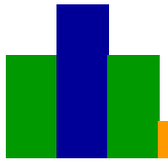


..Condition variables

Example:

```
action(){                          counter(){
    ⋮                                  ⋮
  lock();                            lock();
  while(x != 0)                      x--;
    wait(s);                           if (x == 0) signal(s);
  unlock(s);                         unlock();
  doAction;                            ⋮
    ⋮                              }
}
```

`wait(s)` unlocks the lock and waits for signal `s`; the loop `while ...` is used as a double check for the condition; also notice that signals have no memory [if `counter` reaches its critical section first, the signal `s` is lost]



Pthread condition variables

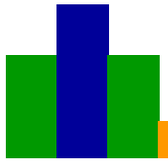
Pthread has *condition* variables *associated* to *mutex* variables;
Signal and wait mechanisms have the following format:

—in `action` part

```
    :
pthread_mutex_lock(&mutex1);
while (c <> 0)
  pthread_mutex_wait(cond1,mutex1);
pthread_mutex_unlock(&mutex1);
```

—in `counter` part

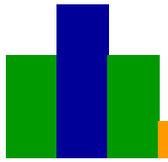
```
    :
pthread_mutex_lock(&mutex1);
c--;
  if (c == 0) pthread_cond_signal(cond1);
pthread_mutex_unlock(&mutex1);
```



Language constructs for parallelism

A few *specific* language constructs for shared memory parallel programs are:

- *Shared data* may be specified as
  ```
  shared int;
  ```
- *Different concurrent* statements may be introduced using the `par` statement
  ```
  par{S1;S2; ...,Sn;}
  ```
- *similar concurrent statements* may be introduced using the `forall` statement
  ```
  forall (i=0; i<n; i++){body}
  ```
  which generates *n* concurrent blocks {`body1; body2; ...; bodyn;`}, one instance of the `body` for each `i`



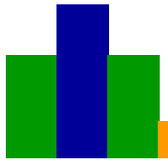
Dependency analysis

Dependency analysis is an *automatic* technique used by *parallelizing compilers* to detect which processes/statements may be executed in *parallel*

Examples (1st - obvious; 2nd - easy, but not so obvious)

```
forall(i=0; i<5; i++)
  a[i] = 0
```

```
forall(i=2; i<6; i++){
  x = i - 2*i + i*i;
  a[i] = a[x];
}
```



Bernstein's conditions

Bernstein's conditions: A set of *sufficient* conditions for two processes to be executed simultaneously is the following *concurrent-read/exclusive-write* situation:

- define two sets of memory locations

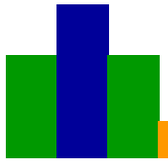
  I_i - the set of memory locations read by process P_i

  O_i - the set of memory locations where process P_i writes

- two processes $P1, P_2$ may be *executed simultaneously* if

$$O_1 \cap O_2 = \emptyset \quad \& \quad I_1 \cap O_2 = \emptyset \quad \& \quad I_2 \cap O_1 = \emptyset$$

  (in words: the writing locations are disjoint and no process reads from a location where the other process writes.)



Sharing data in systems with caches

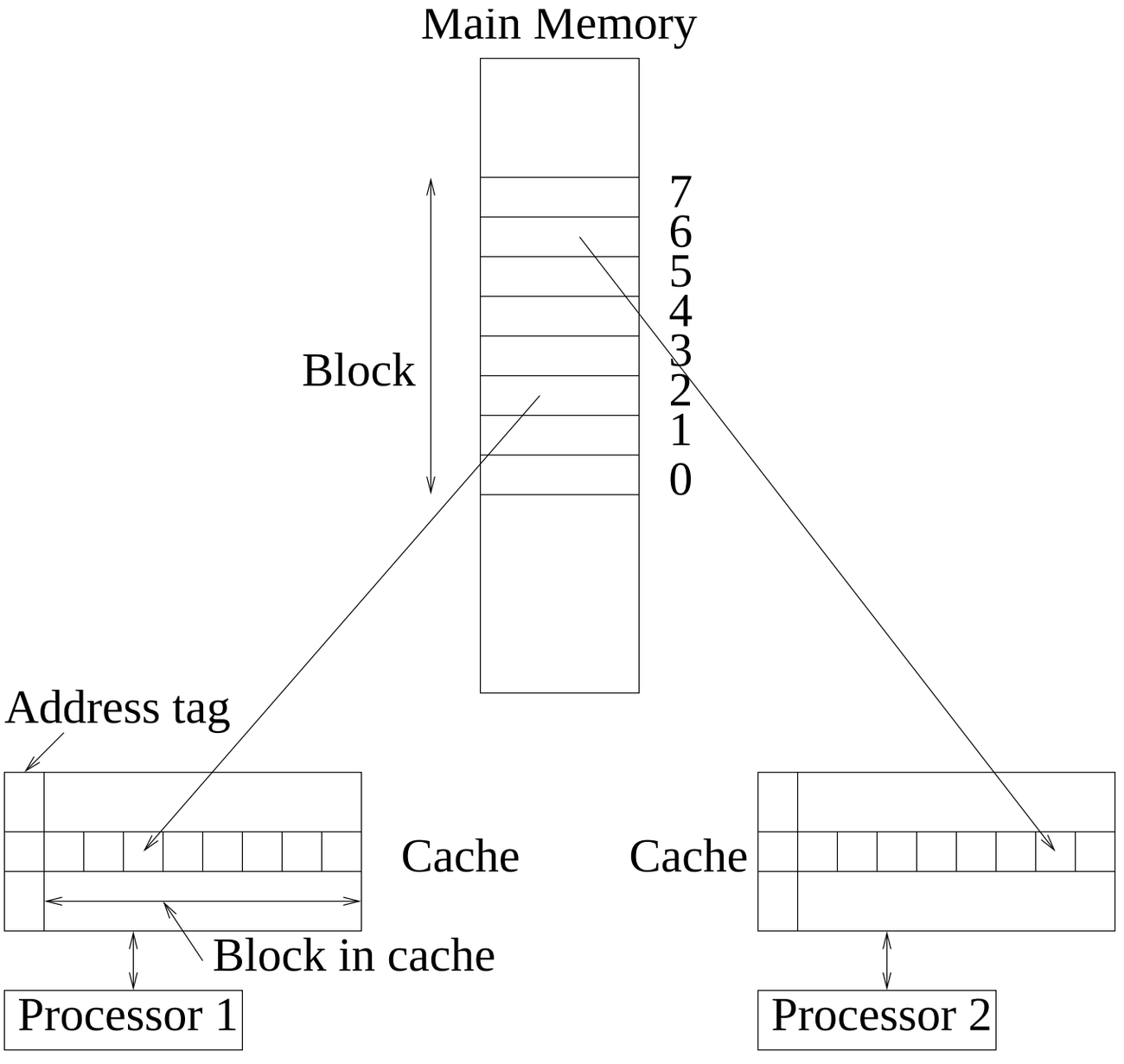
- *cache memory* is a high-speed memory closely attached to a processor to hold the recent used data and code

- *cache coherence protocols*: need to assure processes have coherent caches; two solutions:
  —*update policy* - copies in all caches are updated anytime when one copy is modified
  —*invalidate policy* - when a datum in one copy is modified, the same datum in the other caches is invalidated (resetting a bit); the new data is updated only if the processor needs it

- a *false sharing* may appear when different processors alter different data within the same block (solution - memory rearrangement)

Threads

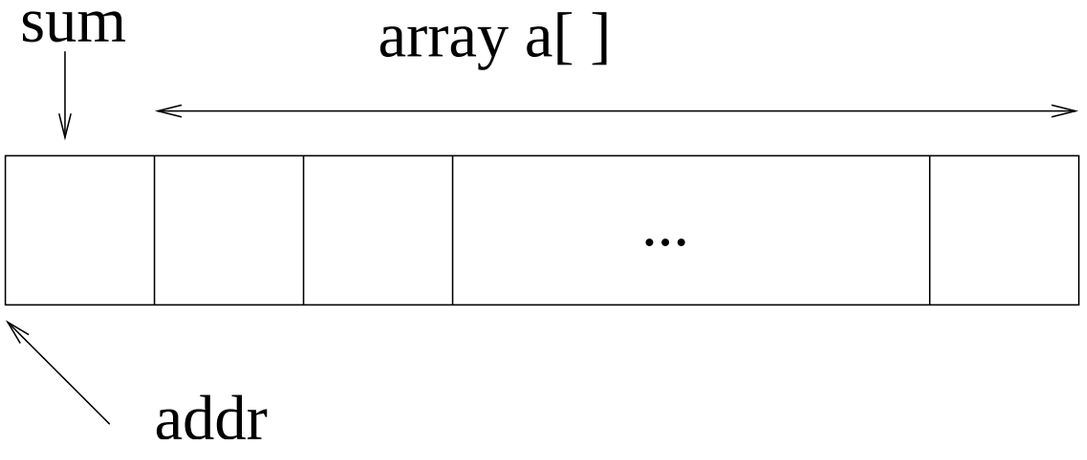
False sharing:



Examples (add 1000 numbers): 1 Unix

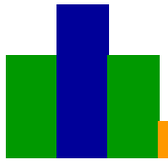
Adding numbers: The aim is to add 1000 numbers using two processes, namelly `P1` will add the numbers with an even index in array `a[ ]`, while `P2` those with odd index in `a[ ]`.

The following memory structure is used



..Examples / Unix

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <stdio.h>
#include <errno.h>
#define array_size 1000
extern char *shmat();
void P(int *s);
void V(int *s);
int main()
{
   int shmid, s, pid;
   char *shm;
   int *a, *addr, *sum;
   int partial_sum;
   int i;
```

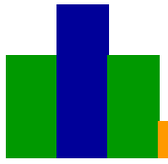


..Examples / Unix

```
int init_sem_value = 1;
s = semget(IPC_PRIVATE, 1, (0600 | IPC_CREAT));
if (s == -1){
   perror("semget");
   exit(1);
}
if (semctl(s,0,SETVAL, init_sem_value) < 0){
   perror("semctl");
   exit(1);
}

shmid = shmget(IPC_PRIVATE, (array_size*sizeof(int)+1),
      (IPC_CREAT|0600));
if (shmid == -1){
   perror("semget");
   exit(1);
}
```

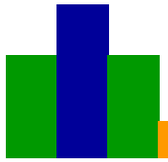
..Examples / Unix

```
shm = shmat(shmid, NULL, 0);

if (shm == (char*)-1){
   perror("shmat");
   exit(1);
}

addr = (int*)shm;
sum = addr;
addr++;
a = addr;

*sum = 0;
for (i = 0; i < array_size; i++)
   *(a+i) = i+1;
```



..Examples / Unix

```
pid = fork();
if (pid == 0){
   partial_sum = 0;
   for (i = 0; i < array_size; i = i + 2)
      partial_sum += *(a + i);
} else {
   partial_sum = 0;
   for (i = 1; i < array_size; i = i + 2)
      partial_sum += *(a + i);
}
P(&s);
*sum += partial_sum;
V(&s);

printf("\nprocess pid = %d, partial sum = %d\n",
      pid, partial_sum);
if (pid == 0) exit(0); else wait (0);
printf("\nThe sum of 1 to %i is %d\n",
      array_size, *sum);
```

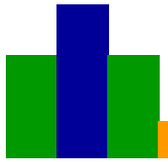


..Examples / Unix

```
    if (semctl(s, 0, IPC_RMID, 1) == -1) {
       perror("semctl");
       exit(1);
    }

    if (shmctl(shmid, IPC_RMID, NULL) == -1) {
       perror("shmctl");
       exit(1);
    }

    exit(0);
}
```

..Examples / Unix

```
void P(int *s)
   {
   struct sembuf sembuffer, *sops;
   sops = &sembuffer;
   sops->sem_num = 0;
   sops->sem_op = -1;
   sops->sem_flg = 0;
   if (semop(*s, sops, 1) < 0) {
      perror("semop");
      exit(1);
   }
   return;
}
```



..Examples / Unix

```
void V(int *s)
{
   struct sembuf sembuffer, *sops;
   sops = &sembuffer;
   sops->sem_num = 0;
   sops->sem_op = 1;
   sops->sem_flg = 0;
   if (semop(*s, sops, 1) < 0) {
      perror("semop");
      exit(1);
   }
   return;
}
```

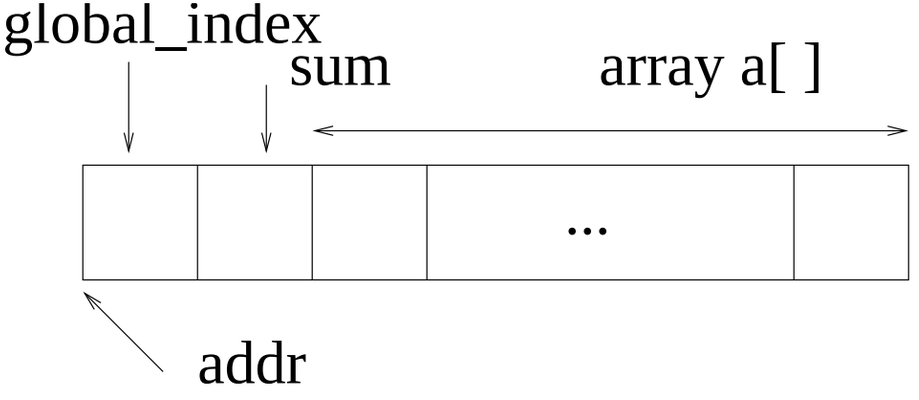
```
process pid = 0, partial sum = 250000
process pid = 2073, partial sum = 250500

The sum of 1 to 1000 is 500500
```

Examples (add 1000 numbers): 2 Pthreads

Adding numbers: The aim is again to add 1000 numbers, now using ten threads:

- each thread reads in a `global_index` for scanning array `a[]` and adds the corresponding element to its local `partial_sum`
- then, each thread will add its `partial_sum` to a global `sum` variable
- both, the access/update of `global_index` and of `sum` variables are critical sections implemented using mutex variables

The following memory structure is used

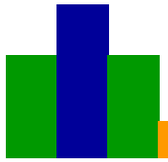


..Examples / Pthreads

```
#include <stdio.h>
#include <pthread.h>
#define array_size 1000
#define no_threads 10

int a[array_size];
int global_index = 0;
int sum = 0;
pthread_mutex_t mutex1;
```

..Examples / Pthreads

```
void *slave(void *ignored)
{
   int local_index, partial_sum = 0;
   do {
      pthread_mutex_lock(&mutex1);
         local_index = global_index;
         global_index++;
      pthread_mutex_unlock(&mutex1);

      if (local_index < array_size)
         partial_sum += *(a + local_index);
   } while (local_index < array_size);
   pthread_mutex_lock(&mutex1);
      sum += partial_sum;
   pthread_mutex_unlock(&mutex1);
   return();
}
```



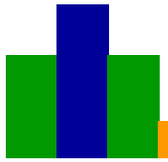
..Examples / Pthreads

```
main()
{
   int i;
   pthread_t thread[10];
   pthread_mutex_init(&mutex1, NULL);

   for (i = 0; i < array_size; i++)
      a[i] = i+1;


   for (i = 0; i < no_threads; i++)
      if (pthread_create(&thread[i], NULL, slave, NULL) != 0)
         perror("Pthread_create fails");

   for (i = 0; i < no_threads; i++)
      if (pthread_join(thread[i], NULL) != 0)
         perror("Pthread_join fails");

   printf("The sum of 1 to %i is %d\n", array_size, sum);
}
```

Examples (add 1000 numbers): 2 Java

Adding numbers: The aim is again to add 1000 numbers using ten threads, but now using Java programming.

The key mechanism to implement the access to the critical sections is by using *synchronized* methods.

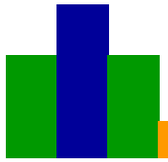


..Examples / Java

```
public class Adder
{
   public int[] array;
   private int sum = 0;
   private int index = 0;
   private int number_of_threads = 10;
   private int threads_quit;

   public Adder()
   {
      threads_quit = 0;
      array = new int[1000];
      initializeArray();
      startThreads();
   }

   public synchronized int getNextIndex()
   {
      if (index < 1000) return (index++); else return (-1);
   }
```

..Examples / Java

```
public synchronized void addPartialSum(int partial_sum)
{
   sum = sum + partial_sum;
   if (++threads_quit == number_of_threads)
      System.out.println("The sum of the numbers is " + sum);
}

private void initializeArray()
{
   int i;
   for (i = 0; i < 1000; i++)
      array[i] = i;
}
```

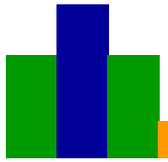


..Examples / Java

```
    public void startThreads()
    {
       int i = 0;
       for (i = 0; i < 10; i++)
          {
             AdderThread at = new AdderThread(this,i);
          at.start();
          }
    }

    public static void main(String args[])
    {
       Adder a = new Adder();
    }
}
```

..Examples / Java

```
class AdderThread extends Thread
{
   int partial_sum = 0;
   Adder parent;
   int number;

   public AdderThread(Adder parent, int number)
   {
      this.parent = parent;
      this.number = number;
   }
```



..Examples / Java

```
    public void run()
    {
       int index = 0;

       while(index != -1) {
          partial_sum = partial_sum + parent.array[index];
          index = parent.getNextIndex();
       }

       System.out.println("Partial sum from thread "
            + number + " is " + partial_sum);
       parent.addPartialSum(partial_sum);
    }
}
```

..Examples / Java

```
Partial sum from thread 8 is 0
Partial sum from thread 7 is 910
Partial sum from thread 1 is 165931
Partial sum from thread 2 is 51696
Partial sum from thread 9 is 10670
Partial sum from thread 6 is 54351
Partial sum from thread 0 is 98749
Partial sum from thread 3 is 62057
Partial sum from thread 5 is 45450
Partial sum from thread 4 is 9686
The sum of the numbers is 499500
```