Programação com memória partilhada

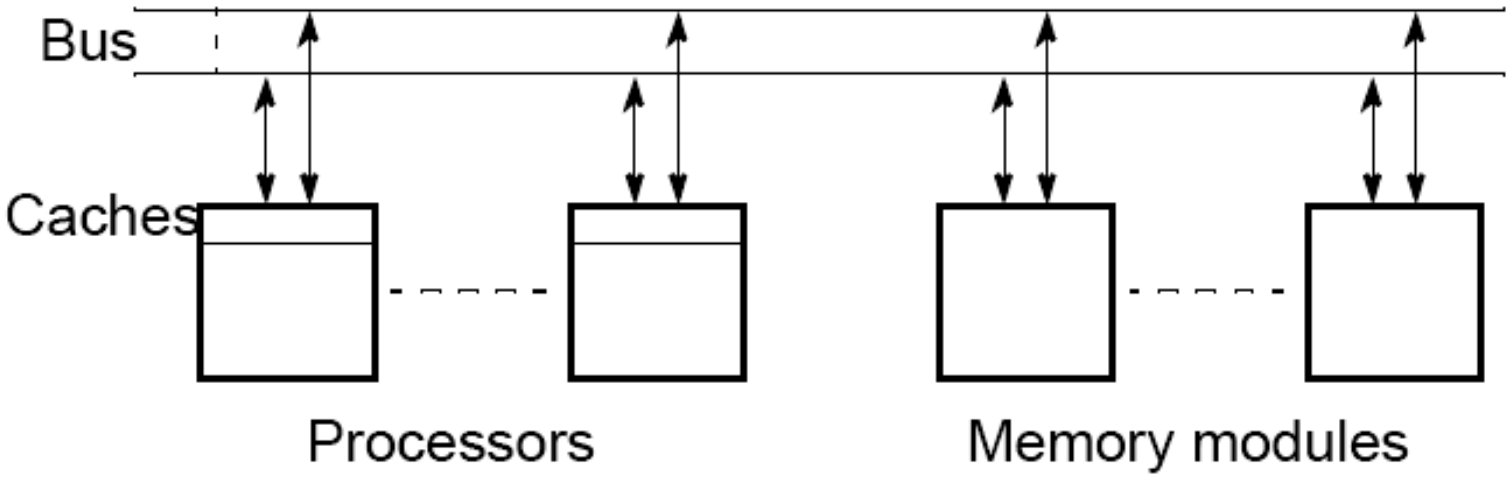
Multiprocessadores com memória partilhada

Qualquer localização de memória pode ser acedida por qualquer processo.

Só existe um espaço de endereçamento.

Exige o controlo de acesso a dados partilhados.

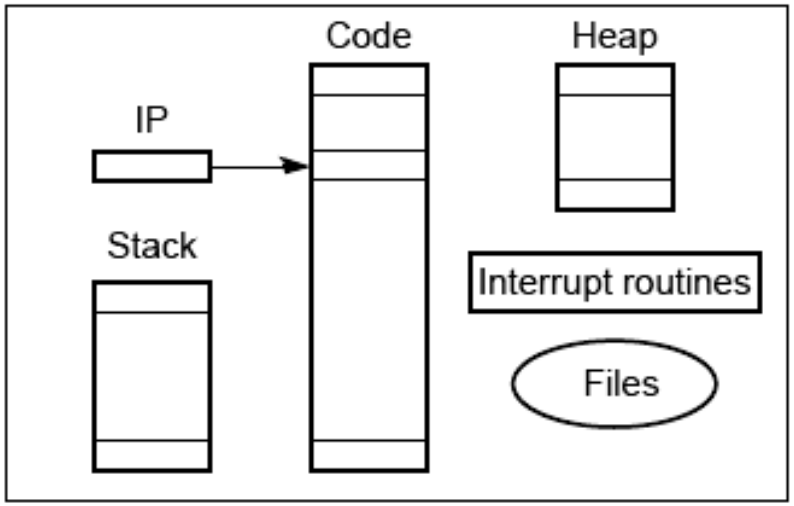
Shared memory multiprocessor using a single bus



Diferenças entre processos e threads

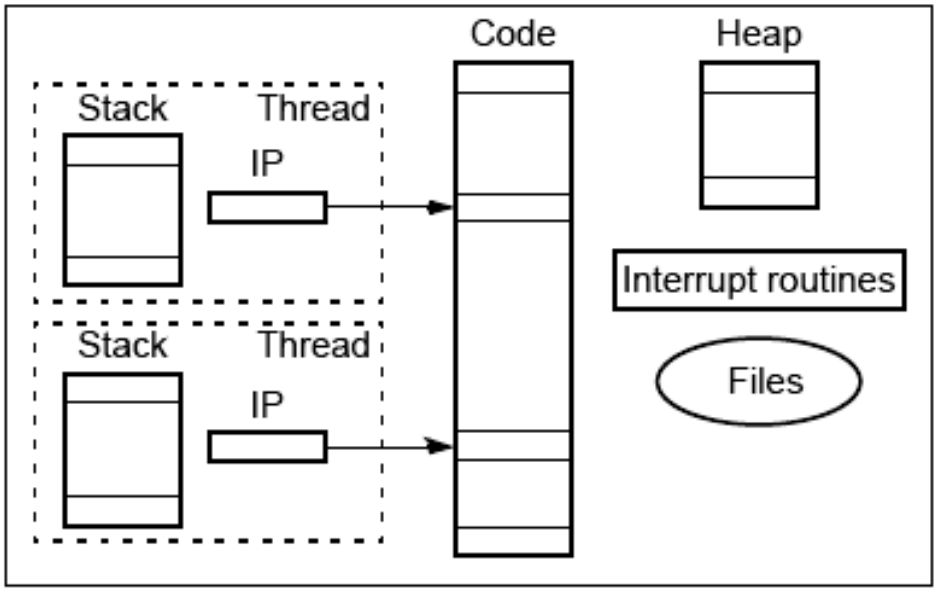
"heavyweight" process - completely separate program with its own variables, stack, and memory allocation.

(a) Process



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

(b) Threads



Ordem de execução das instruções

Exemplo

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

Várias ordens possíveis, por exemplo:

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

Assumindo que as instruções já não são divisíveis.

Otimizações do compilador / processador

Para otimizar o desempenho de um programa o compilador pode trocar a ordem das instruções.

Exemplo

As instruções:

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

Podem ser executadas pela ordem inversa:

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

A lógica do programa não é alterada.

Threads em python:

class threading.Thread (***group***=*None*,

target=*None*, ***name***=*None*, ***args***=*()*, ***kwargs***={}, ***daemon***=*None*)

group - deve ser None; reservado para futura implementação de uma classe ThreadGroup

target - "callable object" (função) que será invocado no método run.

Name – nome da thread, por omissão: "Thread-*N*" com n inteiro

args - argumentos para a função "target"

Threads em python:

class threading.Thread (***group***=*None*,

target=*None*, ***name***=*None*, ***args***=*()*, ***kwargs***={}, ***daemon***=*None*)

…

kwargs - dicionário de arguments para a função “target”

daemon - explicita se a thread é daemon

Se uma subclasse de Thread sobrepõe o construtor, tem de invocar o construtor da classe Thread (Thread.__init__()) antes de qualquer outra instrução.

Threads em python:

```
#Hello_world.py
# Importar Thread e sleep
from threading import Thread
from time import sleep

# A classe exemplo irá ser subclasse de Thread
class Exemplo1(Thread):
    def __init__(self):
    Thread.__init__(self)
     self.message = "Hello Parallel example!\n"

#Método que imprime a mensagem
    def print_message(self):
        print (self.message)
```

Threads em python:

```
##O método run vai ser executado pela thread
    def run(self):
        print ("Thread Starting\n")
        x=0
        while (x < 10):
            self.print_message()
            sleep(2)
            x += 1
        print ("Thread Ended\n")
```

Threads em python:

```
# Início do processo principal
print ("Process Started")

# Criar uma instância da classe Exemplo1

hello_Python = Exemplo1()

# Iniciar a thread
hello_Python.start()


#Fim do processo principal
print ("Process Ended")
```

Output:

Process Started
Thread Starting

Hello Parallel example!

Process Ended

Hello Parallel example!
Hello Parallel example!
Hello Parallel example!
Hello Parallel example!
Hello Parallel example!
Hello Parallel example!
Hello Parallel example!
Hello Parallel example!
Hello Parallel example!

Thread Ended

Threads em python:

```
# Exemplo2 – criar thread usando diretamente a classe #Thread
import threading
def function(i):
    print ("function called by thread %i\n" %i)
    return

threads = []
for i in range(5):
    t = threading.Thread (target=function , args=(i,))
    threads.append(t)
    t.start()
    t.join()   # processo principal espera que a t termine
```

Função a ser invocada no run da Thread

Threads em python:

Exemplo 2 – cria 5 threads

… Podemos substituir:

```
threads = []
for i in range(5):
    t = threading.Thread (target=function , args=(i,))
    threads.append(t)
    t.start()
    t.join()
```

Por:

```
threads = [ threading.Thread (target=function , args=(i,)) for i in range (5) ]
[t.start() for t in threads]
[t.join() for t in threads]
```

Output:

function called by thread 0

function called by thread 1

function called by thread 2

function called by thread 3

function called by thread 4

Threads em python:

```
# Obter o nome da thread

import threading
import time

def first_function():
    print (threading.currentThread().getName()+\
    str(' is Starting \n'))
    time.sleep(2)
    print (threading.currentThread().getName()+\
    str( ' is Exiting \n'))
    return
```

Threads em python:

```
def second_function():
    …
def third_function():
    …

if __name__ == "__main__":
    t1 = threading.Thread\
    (name='first_function', target=first_function)
    …
    t1.start()
    …
    t1.join()
    …
```

Threads em python:

Output:

first_function is Starting

second_function is Starting

third_function is Starting

first_function is Exiting

second_function is Exiting

third_function is Exiting

Threads em python:

Exemplo3

```
import threading
import time


class myThread (threading.Thread):
    def __init__(self, threadID, name, counter):
        threading.Thread.__init__(self)
        self.threadID = threadID
        self.name = name
        self.counter = counter
```

Subclasse de Thread

Sobrepor init

Threads em python:

Exemplo3

…

```
def run(self):
    print ("Starting " + self.name + "\n")
    print_time(self.name, self.counter, 5)
    print ("Exiting " + self.name + "\n")

def print_time(threadName, delay, counter):
     while counter:
         time.sleep(delay)
         print ("%s: %s" % (threadName,  time.ctime(time.time())))
         counter -= 1
```

Sobrepor run

Threads em python:

Exemplo3

...

```
#Main
if __name__ == '__main__':
# Criar duas threads
    thread1 = myThread(1, "Thread-1", 1)
    thread2 = myThread(2, "Thread-2", 2)
# Iniciar as threads criadas
    thread1.start()
    thread2.start()
# Esperar que as threads terminem a execução
    thread1.join()
    thread2.join()
    print ("Exiting Main Thread")
```

Threads em python:

- Starting Thread-1

  Starting Thread-2

  Thread-1: Sun Mar 4 22:00:28 2018
  Thread-2: Sun Mar 4 22:00:29 2018
  Thread-1: Sun Mar 4 22:00:29 2018
  Thread-1: Sun Mar 4 22:00:30 2018
  Thread-2: Sun Mar 4 22:00:31 2018
  Thread-1: Sun Mar 4 22:00:31 2018
  Thread-1: Sun Mar 4 22:00:32 2018
  Exiting Thread-1

  Thread-2: Sun Mar 4 22:00:33 2018
  Thread-2: Sun Mar 4 22:00:35 2018
  Thread-2: Sun Mar 4 22:00:37 2018
  Exiting Thread-2

  Exiting Main Thread

Thread-Safe Routines

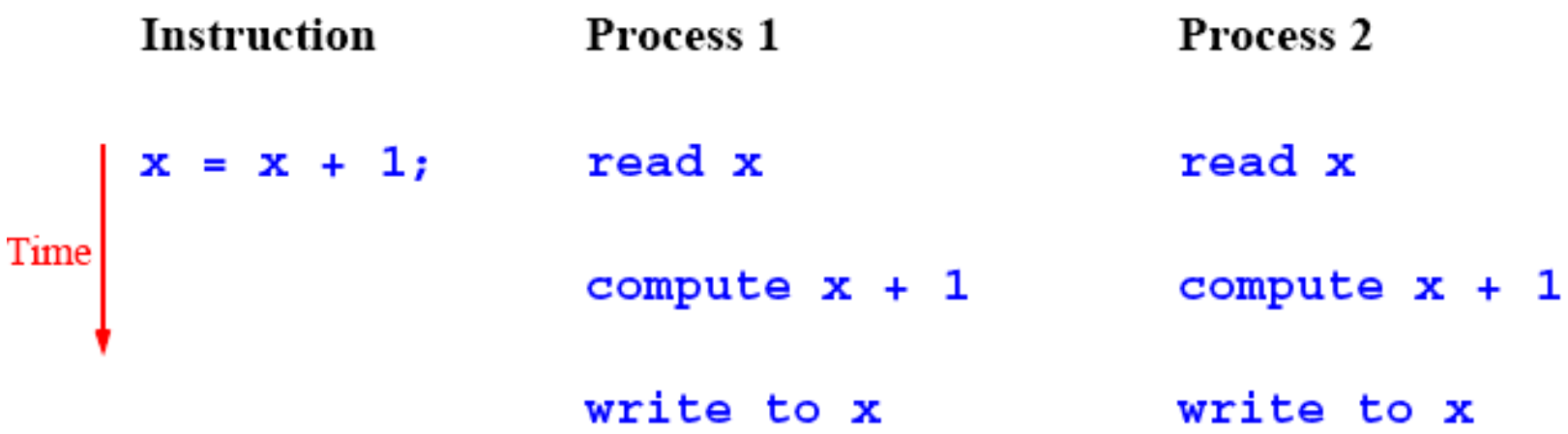
Uma função diz-se "Thread safe" se pode ser chamada simultaneamente por várias threads e produzir sempre resultados corretos.

Standard I/O é thread safe (as mensagens são impressas sem intercalação de carateres.)

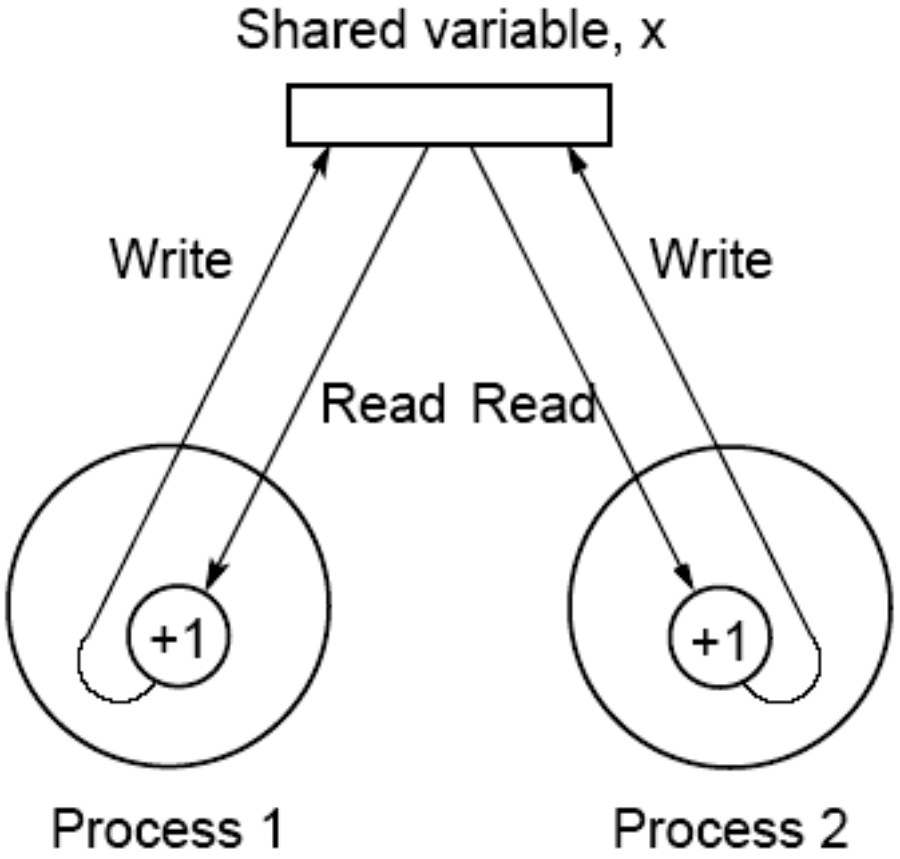
Funções que acedem a dados partilhados devem ser thread safe.

Acesso a dados partilhados:

Sejam dois processos, cada um vai adicionar uma unidade à variável partilhada x: É necessário ler a variável x, adicionar 1 e escrever o resultado novamente em x.



Conflito no acesso à variável partilhada



Secção crítica e exclusão mútua

É necessário um mecanismo que assegure que apenas um processo de cada vez acede às secções de código que envolvem variáveis partilhadas (**secções críticas**), (sto é, evitar “race conditions”.

Isto é, queremos assegurar que cada secção crítica (S.C.) é executada em exclusão mútua.

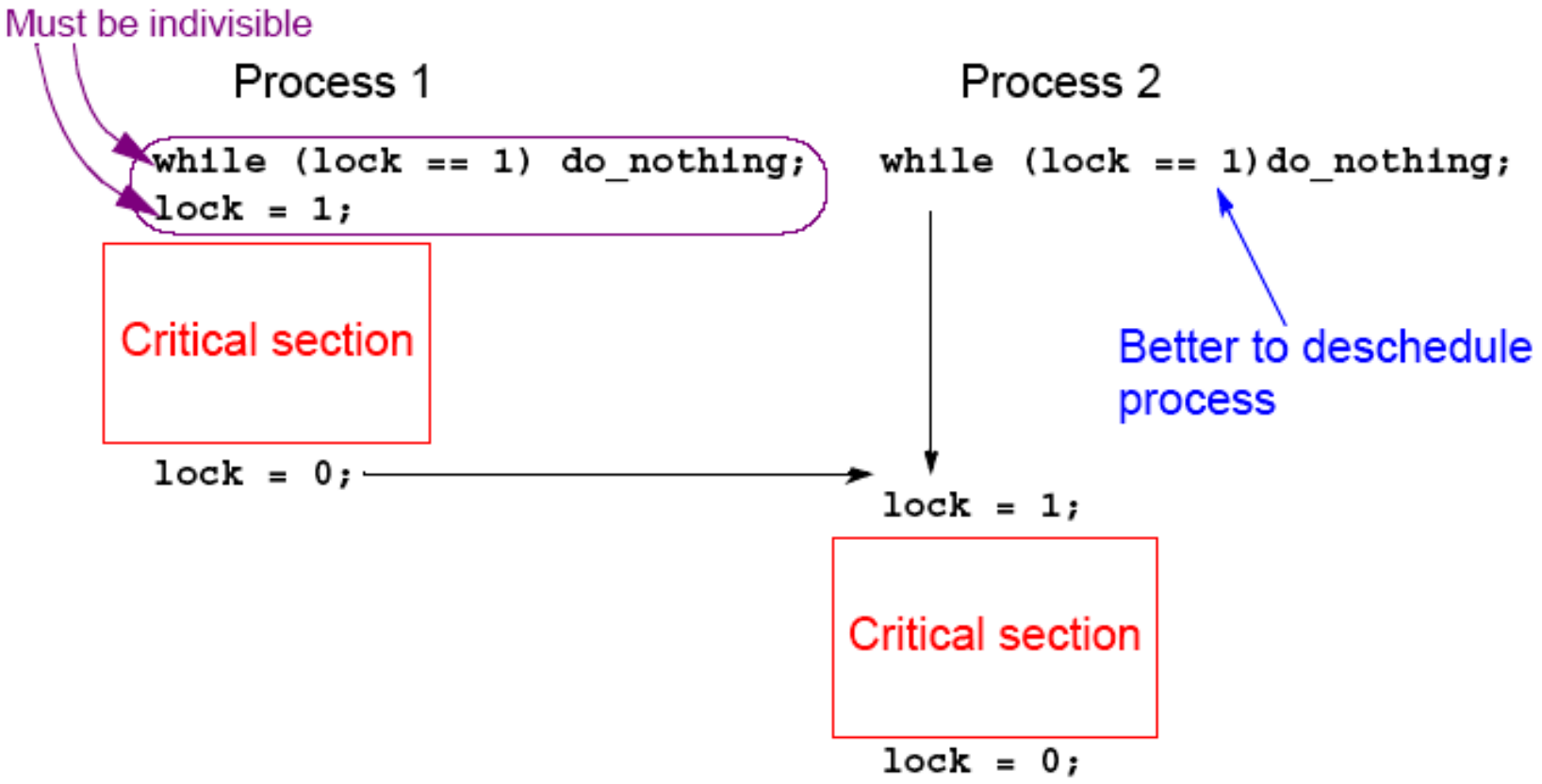
Locks

Mecanismo mais simples, para assegurar exclusão mútua.

Um lock é uma variável que contém o valor 1 se um processo entrou na S.C e tem o valor 0 se nenhum processo está na S.C..

Um processo que tenta entrar na S.C e vê que o lock está 0, avança, fechando o lock, isto é, colocando lock a 1. Quando sai da S.C volta a abrir o lock (isto é, coloca-o a 0).

Controlo de S.C. através de espera ativa



Locks em Python

GIL – Global Interpreter Lock

Em python, existe um lock global , GIL, que qualquer thread tem que deter para conseguir executar.

O interpretador vai escalonar uma nova thread após um número de instruções pré-determinado.

O GIL é também libertado quando ocorre uma operação de I/O.

Locks em Python

my_lock= threading.Lock()

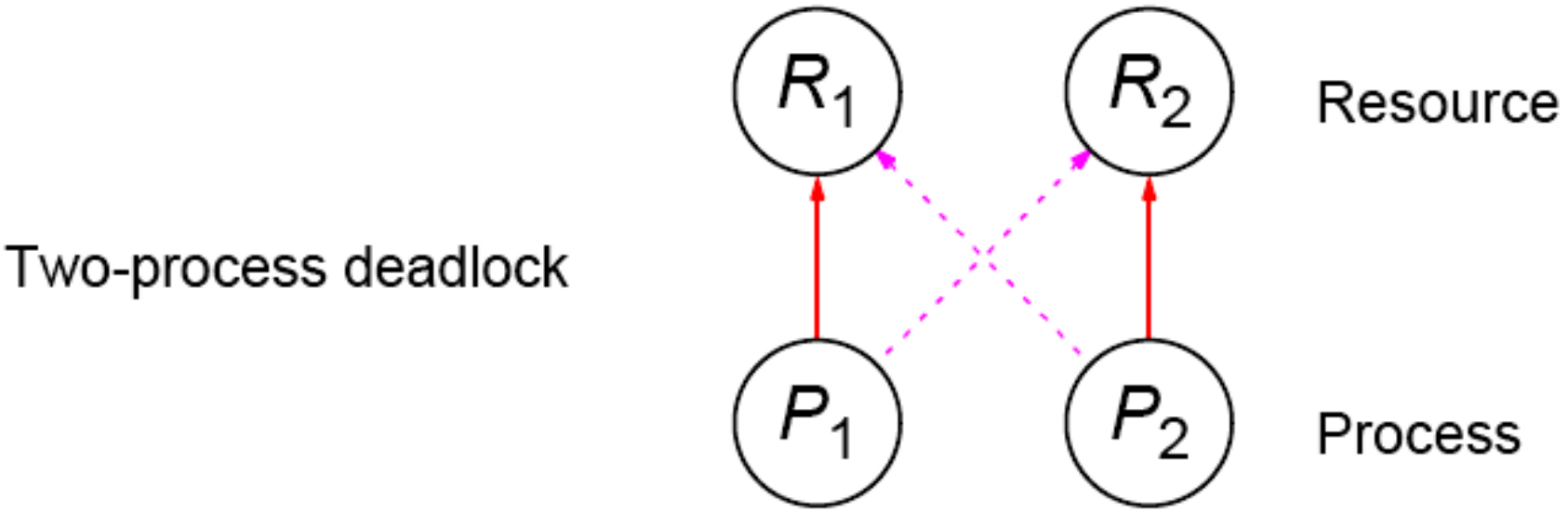
my_lock.acquire()

s.c.

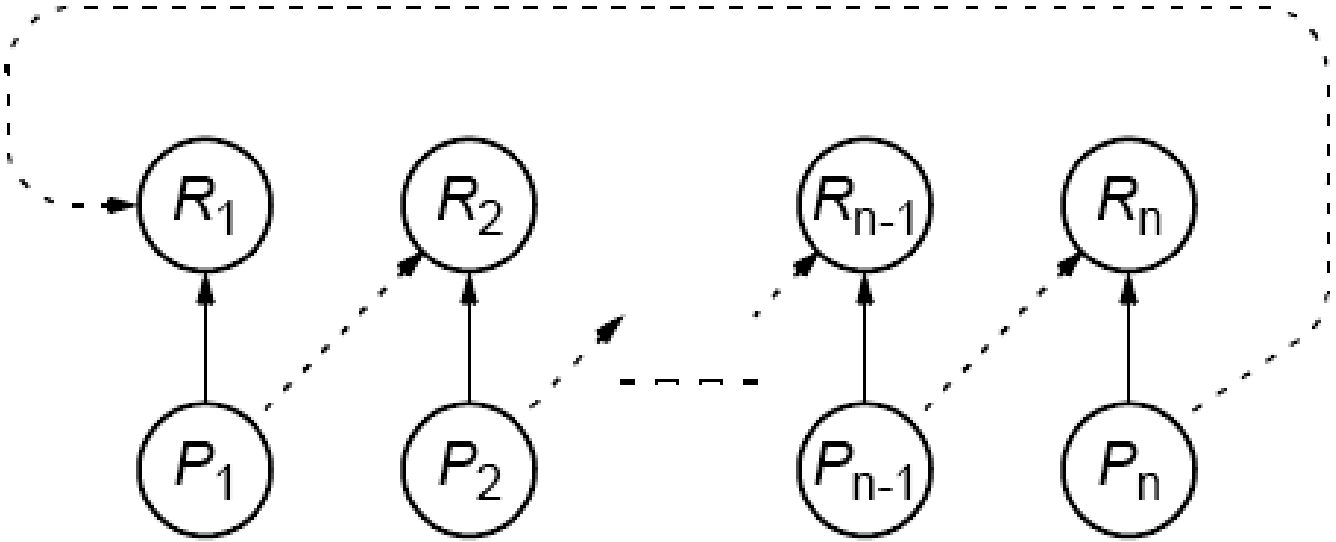
my_lock. release()

Deadlock

Pode ocorrer entre dois processos, quando P1 necessita de um recurso detido por P2, e P2 necessita de um recurso detido por P1.



Deadlock (deadly embrace)



Locks em Python

[Variáveis partilhadas em python:

```
xpto = 10

def f1():
    print (xpto)

def f2():
    print(xpto)

if __name__== "__main__":
    f1()
    f2()
    print(xpto)
```

output: … ?

Locks em Python

```
xpto = 10

def f1():
    xpto = xpto +1
    print (xpto)

def f2():
    print(xpto)

if __name__== "__main__":
    f1()
    f2()
    print(xpto)
```

Nova variável local a f1()

```
Erro: ****
f1()
File "D:\_PPD_17_18-mestrado\pratica\gblobais1.py", line 4, in f1
xpto = xpto +1
```
UnboundLocalError: local variable 'xpto' referenced before assignment

Locks em Python

```
xpto = 10

def f1():
    global xpto
    xpto += 2
    print (xpto)

def f2():
    global xpto
    xpto -= 1
    print(xpto)

if __name__== "__main__":
     f1()
     f2()
     print (xpto)
```

Output: ??

➔ **Estudar: - passagem de parâmetros em python**

Locks em Python

```
class c:
    def __init__(self, att=None ):
        self.att = att

    def set_att(self, x):
        self.att = x

    def get_att (self):
        return self.att

    def str (self):
        return "Conteúdo do objeto: " + str(self.att)

if __name__== "__main__":
    c1 = c()
    c1.set_att("ABC")
    c2 = c("XPTO")
    print (c1.str())
    print (c2.str())
```

Output:
Conteúdo do objeto: ABC
Conteúdo do objeto: XPTO

Locks em Python

```
c1 = c()

def f1(c):
    c.set_att("F1")
    print (c1.str())

def f2(c):
    c.set_att("F2")
    print (c.str())

if __name__== "__main__":
    print (c1.str())
    f1(c1)
    f2(c1)
    print (c1.str())
    c2 = c("XPTO")
    print (c2.str())
```

Output: ???

Locks em Python

Output:

Conteúdo do objeto: None
Conteúdo do objeto: F1
Conteúdo do objeto: F2
Conteúdo do objeto: F2
Conteúdo do objeto: XPTO

Locks em Python

Exemplo:

```
import threading

shared_resource_with_lock = 0
COUNT = 100000
shared_resource_lock = threading.Lock()

def increment_with_lock():
    global shared_resource_with_lock
    for i in range(COUNT):
        shared_resource_lock.acquire()
        shared_resource_with_lock += 1
        shared_resource_lock.release()
```

S.C. executada em exclusão mútua

Locks em Python

Exemplo:

…

```
def decrement_with_lock():
    global shared_resource_with_lock
    for i in range(COUNT):
        shared_resource_lock.acquire()
        shared_resource_with_lock -= 1
        shared_resource_lock.release()
```

S.C. executada em exclusão mútua

```
if __name__ == "__main__":
     t1 = threading.Thread(target = increment_with_lock)
     t2 = threading.Thread(target = decrement_with_lock)
t1.start()
t2.start()
t1.join()
t2.join()
print ("the value of shared variable with lock management is %s"\
%shared_resource_with_lock)
```

RLocks em Python

threading.Rlock – **Lock reentrante ou recursivo**

rlock = threading.RLock()

rlock.acquire()

S.C.

rlock.release()

Exemplo ➔

RLocks em Python

```
class Box:
    lock = threading.RLock()   # variável de classe

    def __init__(self):
        self.total_items = 0

     def execute(self,n):
         Box.lock.acquire()
         self.total_items += n
         Box.lock.release()

     def add(self):
         Box.lock.acquire()
         self.execute(1)
         Box.lock.release()

     def remove(self):
         Box.lock.acquire()
         self.execute(-1)
         Box.lock.release()
```

RLocks em Python

```
def adder(box,items):
    while items > 0:
        print ("adding 1 item in the box\n")
        box.add()
        time.sleep(5)
        items -= 1


def remover(box,items):
    while items > 0:
        print ("removing 1 item in the box")
        box.remove()
        time.sleep(5)
        items -= 1
```

RLocks em Python

```
if __name__ == "__main__":
    items = 5
    print ("putting %s items in the box " % items)
    box = Box()   # ojecto partilhado

    t1 = threading.Thread(target=adder,args=(box,items))
    t2 = threading.Thread(target=remover,args=(box,items))

    t1.start()
    t2.start()

    t1.join()
    t2.join()
    print ("%s items still remain in the box " % box.total_items)
```

RLocks em Python

Output:

putting 5 items in the box

adding 1 item in the box
removing 1 item in the box
removing 1 item in the box
adding 1 item in the box
removing 1 item in the box
adding 1 item in the box
adding 1 item in the box
removing 1 item in the box
adding 1 item in the box
removing 1 item in the box

0 items still remain in the box

Semáforos

Tipo abstrato de dados representado por um valor inteiro e manipulado por duas operações:

Operação P num semáforo s

- O processo espera até que **s** seja maior que 0, após o que decrementa **s** de 1 e continua a execução

Operação V num semáforo *s*

- O processo incrementa **s** de 1 e liberta um dos processos em espera (se algum)

- P e V são operações atómicas.

- O mecanismo para ativar os processos em espera é implícito nas operações P e V. É esperado que o algoritmo seja justo.

- Processos suspensos por **P**(s) permanecem em espera até serem libertados por uma operação **V**(s) no mesmo semáforo.

- Proposto por Dijkstra in 1968. Letra **P** vem da palavra alemã ***passeren***, que significa “passar” a letra **V** vem da palavra alemã ***vrijgeven***, que significa “libertar”.

Exclusão mútua entre secções criticas pode ser obtida com um semáforo que pode ter os valores 0 ou 1 (semáforo binário) que atua como um **lock.**

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

Semáforo não binário

Pode tomar valores positivos diferentes de 0 e 1.

Permite por exemplo quantificar o número de processos que pode aceder simultaneamente a um recurso.

Semáforos em python

class threading.Semaphore(*value=1*)

Representa um contador atómico que contém o número de invocações ao método release() menos o número de invocações ao método acquire() mais o valor inicial do semáforo (valor de omissão: 1).

Se é atribuído um valor negativo à variável do semáforo, é gerada a exceção (ValueError)

```
sem = threading.Semaphore(1)
sem.acquire()
s.c.
sem.release()
```

Semáforos em python

Exemplo (produtor/consumidor)*:*

```
import threading
import time
import random

semaphore = threading.Semaphore(0)

def consumer():
    global item
    print ("consumer is waiting.")
    semaphore.acquire()
    print ("Consumer notify : consumed item nº %s " %item)
```

Semáforos em python

```
…
def producer():
    global item
    time.sleep(10)
    item = random.randint(0,1000)
    print ("producer notify : producted item number %s" %item)
    semaphore.release()

if __name__ == '__main__':
   for i in range (0,5) :
       t1 = threading.Thread(target=producer)
       t2 = threading.Thread(target=consumer)
       t1.start()
       t2.start()
       t1.join()
       t2.join()
   print ("program terminated.")
```

Semáforos em python

Um output:

consumer is waiting.
producer notify : producted item number 349
Consumer notify : consumed item number 349

consumer is waiting.
producer notify : producted item number 37
Consumer notify : consumed item number 37

consumer is waiting.
producer notify : producted item number 355
Consumer notify : consumed item number 355
consumer is waiting.
producer notify : producted item number 876
Consumer notify : consumed item number 876
consumer is waiting.
producer notify : producted item number 400
Consumer notify : consumed item number 400
program terminated.

Monitores

Conjunto de procedimentos que fornecem o único acesso a um recurso partilhado. Apenas um processo de cada vez pode executar um procedimento do monitor.

Pode ser implementado usando um semáforo ou um lock para proteger a execução do método:

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

Condition Variables

Muitas vezes uma S.C. deve ser executada se uma determinada condição ocorre, por exemplo se uma variável atinge um determinado valor.

Com locks, a condição tem de estar sistematicamente a ser testada (pooled) dentro da S.C.

Tarefa que consome tempo e pouco produtiva e que pode ser evitada com *condition variables*.

Condition Variables (c.v.)

Em python:

```
cond = Condition() # Uma c.v. tem um lock associado

cond.acquire()    #  A tread adquire o lock


cond.release() # A thread liberta o lock
```

Condition Variables (c.v.)

cond.wait()
# o lock é libertado e a thread é suspensa até que uma outra execute um notify na mesma c.v.

cond.notify()
# uma das threads suspensas na c.v. é acordada e pode readquirir o lock, após este ser libertado pela thread que executa o notify.

cond.notify_All()
# todas as threds suspensas no lock são acordadas
# e podem competir pelo lock. …

Condition Variables (c.v.)

Exemplo:

```
from threading import Thread, Condition
import time

items = []
condition = Condition()

class consumer(Thread):
    def __init__(self):
        Thread.__init__(self)
```

Condition Variables (c.v.)

```
def consume (self):
    global condition
    global items
    condition.acquire()

    if len(items) == 0:
        print("Consumer notify : no item to consume")
        condition.wait()

    items.pop()
    print("Consumer notify : consumed 1 item")
    print("Consumer notify : items to consume are " + str(len(items)))

   condition.notify()
   condition.release()
```

Condition Variables (c.v.)

```
def run(self):                    # thread Consumer
    for i in range(0,20):
        time.sleep(10)
        self.consume()


class producer(Thread):
    def __init__(self):
        Thread.__init__(self)
```

Condition Variables (c.v.)

```
def produce(self):
    global condition
    global items

    condition.acquire()
    if len(items) == 10:
        condition.wait()
        print("Producer notify : items produced are " + str(len(items)))
        print("Producer notify : stop the production!!")
     items.append(1)
     print("Producer notify : total items produced " + str(len(items)))
     condition.notify()
     condition.release()
```

Condition Variables (c.v.)

```
def run(self):
    for i in range(0,20):
        time.sleep(5)
        self.produce()


if __name__ == "__main__":
   producer = producer()
   consumer = consumer()
   producer.start()
   consumer.start()
   producer.join()
   consumer.join()
```

Events

Um evento é um objeto que serve para comunicar entre Threads.

e = Event()

São baseados numa *flag* que suporta as operações:

e.set() # atribui o valor true

e.clear() # atribuiu o valor false

e.wait() # bloqueia a thread até que a flag seja true.

Events

Exemplo:

```
import time
from threading import Thread, Event
import random

items = []
event = Event()

class consumer(Thread):
    def __init__(self, items, event):
        Thread.__init__(self)
        self.items = items
        self.event = event
```

Events

Exemplo:

```
def run(self):
    while True:
        time.sleep(2)
        self.event.wait()
        item = self.items.pop()
        print ('Cons. notify : %d popped from list by %s' %(item, self.name))
        self.event.clear()
        print ('Produce notify : event cleared by %s \n' % self.name)
```

Events

Exemplo:

```
class producer(Thread):
    def __init__(self, integers, event):
        Thread.__init__(self)
    self.items = items
    self.event = event

    def run(self):
        for i in range(100):
            time.sleep(2)
            item = random.randint(0, 256)
            self.items.append(item)
            print ('Prod notify: item %d appended to list by %s' % (item,
self.name))
            print ('Producer notify : event set by %s' % self.name)
            self.event.set()
```

Events

Exemplo:

```
if __name__ == '__main__':
    t1 = producer(items, event)
    t2 = consumer(items, event)
    t1.start()
    t2.start()
    t1.join()
    t2.join()
```

Exemplos noutras linguagens:

Program

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];
```

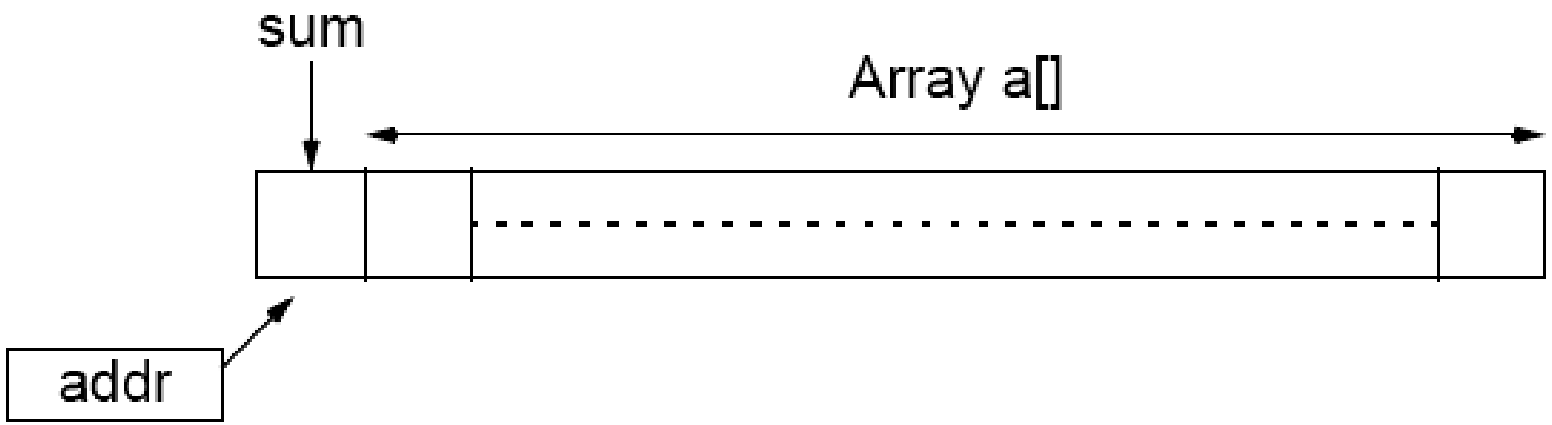
Each process will add its result (sum1 or sum2) to an accumulating result, sum :

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

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

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

Shared memory locations for UNIX program example



```
#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        /* no of elements in shared memory */
extern char *shmat();
void P(int *s);
void V(int *s);
int main()
{
int shmid, s, pid;             /* shared memory, semaphore, proc id */
char *shm;                     /*shared mem. addr returned by shmat()*/
int *a, *addr, *sum;           /* shared data variables*/
int partial_sum;               /* partial sum of each process */
int i;
                               /* initialize semaphore set */
int init_sem_value = 1;
s = semget(IPC_PRIVATE, 1, (0600 | IPC_CREAT))
if (s == -1) {                 /* if unsuccessful*/
   perror("semget");
   exit(1);
}
if (semctl(s, 0, SETVAL, init_sem_value) < 0) {
   perror("semctl");
   exit(1);
}
```

```
                                        /* create segment*/
shmid = shmget(IPC_PRIVATE,(array_size*sizeof(int)+1),
   (IPC_CREAT|0600));
if (shmid == -1) {
   perror("shmget");
   exit(1);
}
                                       /* map segment to process data space */
shm = shmat(shmid, NULL, 0);
                                       /* returns address as a character*/
if (shm == (char*)-1) {
   perror("shmat");
   exit(1);
}
```

```
addr = (int*)shm;                    /* starting address */
sum = addr;                          /* accumulating sum */
addr++;
a = addr;                            /* array of numbers, a[] */

*sum = 0;
for (i = 0; i < array_size; i++) /* load array with numbers */
   *(a + i) = i+1;

pid = fork();                        /* create child process */
if (pid == 0)    {                   /* child does this */
   partial_sum = 0;
   for (i = 0; i < array_size; i = i + 2)
      partial_sum += *(a + i);
else {                               /* parent does this */
   partial_sum = 0;
   for (i = 1; i < array_size; i = i + 2)
      partial_sum += *(a + i);
}
P(&s);                               /* for each process, add partial sum */
   *sum += partial_sum;
V(&s);
```

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

                                            /* remove semaphore */
if (semctl(s, 0, IPC_RMID, 1) == -1) {
   perror("semctl");
   exit(1);
}
                                            /* remove shared memory */
if (shmctl(shmid, IPC_RMID, NULL) == -1) {
   perror("shmctl");
   exit(1);
}
}                                           /* end of main */
```

```
void P(int *s)                                  /* P(s) routine*/
{
   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;
}

void V(int *s)                                  /* V(s) routine */
{
   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;
}
```

SAMPLE OUTPUT

```
process pid = 0, partial sum = 250000
process pid = 26127, partial sum = 250500
The sum of 1 to 1000 is 500500
```

Pthreads Example

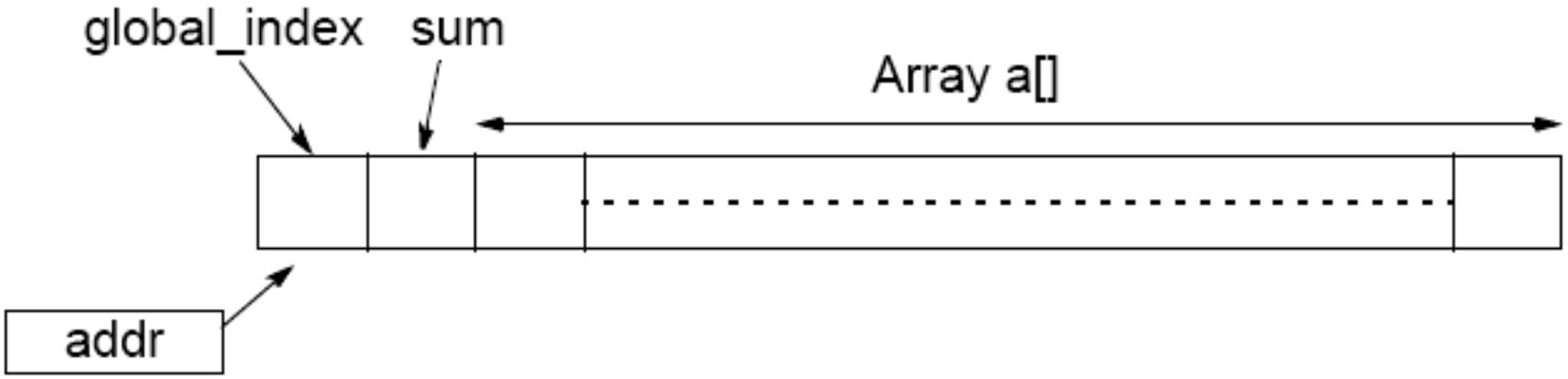
n threads created, each taking numbers from list to add to their sums. When all numbers taken, threads can add their partial results to a shared location sum.

The shared location global_index is used by each thread to select the next element of a[].

After index is read, it is incremented in preparation for the next element to be read.

The result location is sum, as before, and will also need to be shared and access protected by a lock.

Shared memory locations for Pthreads program example



```
#include <stdio.h>
#include <pthread.h>
#define array_size 1000
#define no_threads 10
                                /* shared data */
int a[array_size];              /* array of numbers to sum */
int global_index = 0;           /* global index */
int sum = 0;                    /* final result, also used by slaves */
pthread_mutex_t mutex1;         /* mutually exclusive lock variable */
void *slave(void *ignored)      /* Slave threads */
{
int local_index, partial_sum = 0;
do {
   pthread_mutex_lock(&mutex1);/* get next index into the array */
      local_index = global_index;/* read current index & save locally*/
      global_index++;           /* increment 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);  /* add partial sum to global sum */
   sum += partial_sum;
pthread_mutex_unlock(&mutex1);

return ();                      /* Thread exits */
}
```

```
   main () {
int i;
pthread_t thread[10];                    /* threads */
pthread_mutex_init(&mutex1,NULL);        /* initialize mutex */

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

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

for (i = 0; i < no_threads; i++)         /* join threads */
   if (pthread_join(thread[i], NULL) != 0)
      perror("Pthread_join fails");
printf("The sum of 1 to %i is %d\n", array_size, sum);
}                                        /* end of main */
```

SAMPLE OUTPUT

```
The sum of 1 to 1000 is 500500
```

Java Example

```
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);
   }
```

```
    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;
    }

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

}
```

```
   public static void main(String args[])

class AdderThread extends Thread
{
   int partial_sum = 0;
   Adder parent;
   int number;
   public AdderThread(Adder parent,int number)
   {
      this.parent = parent;
      this.number = number;
   }

   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);
   }
}
```