Operating Systems

Isfahan University of Technology
Electrical and Computer Engineering Department

Zeinab Zali

Synchronization

Background

- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
- Concurrent access to **shared data** may result in data **inconsistency**
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

Illustration of the problem: Calculating summation of numbers with some threads

- We want to calculate the summation of some numbers with more than one thread to speed up the operation
- Consider a global variable sum
- We use data parallelism, divide numbers in to some sets, create a thread for each set to add numbers to sum
- Execute the code in next page and see the result

Illustration of the problem:
Calculating summation of numbers with some threads

```
/********************************************************
Concurrency problem
********************************************************/
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>


struct thread_input{
int b;
int e;
};

int sum = 0;

void *summation_thread(void *input)
{
   struct thread_input *arg;
   arg = (struct thread_input*)input;
   for (int i=arg->b; i<=arg->e; i++){
  sum += i;
}
   pthread_exit(NULL);
}
```

```
int main(int argc, char *argv[])
{
  int n = 10000000;
  pthread_t threads[NUM_THREADS];
  int rc, t;
  struct thread_input beg_end[NUM_THREADS];
  int d = n / NUM_THREADS ;
  for(t=0;t<NUM_THREADS;t++){
       beg_end[t].b = t * d + 1;
        beg_end[t].e = beg_end[t].b + d - 1;
        rc = pthread_create(&threads[t], NULL, summation_thread, (void
        *)&beg_end[t]);
        if (rc){
                printf("ERROR; return code from pthread_create() is %d\
        n", rc);
                exit(-1);
         }
  }
  for (t=0;t<NUM_THREADS;t++){
         pthread_join(threads[t], NULL);
  }
  printf("sum= %d\n", sum);
  /* Last thing that main() should do */
  pthread_exit(NULL);
}
```

Illustration of the problem: updating a linked list

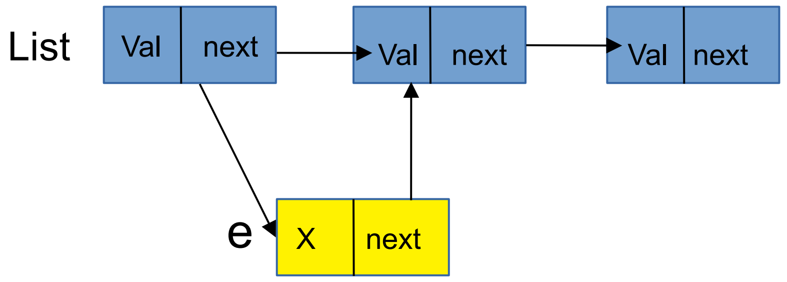
Create new list element e

Set e.value = X

Read list and list.next

Set e.next=list.next

Set list.next=e



What happens if two threads try to add an element concurrently to the same list??

Illustration of the problem: producer-consumer problem

- Suppose that we wanted to provide a solution to the consumer-producer problem that fills ***all*** the buffers.
- We can do so by having an integer **`counter`** that keeps track of the number of full buffers.
- Initially, **`counter`** is set to 0.
- **counter** is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Producer

```
while (true) {
    /* produce an item in next produced */

    while (counter == BUFFER_SIZE) ;
        /* do nothing */
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

Consumer

```
while (true) {
    while (counter == 0)
        ; /* do nothing */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
        counter--;
    /* consume the item in next consumed */
}
```

Race Condition

- `counter++` could be implemented as

  ```
  register1 = counter
  register1 = register1 + 1
  counter = register1
  ```

- `counter--` could be implemented as

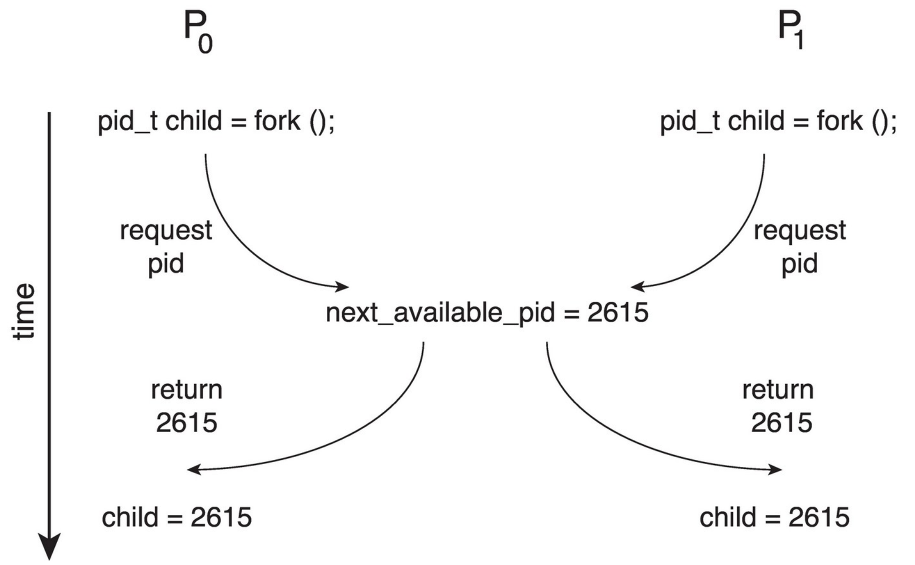
  ```
  register2 = counter
  register2 = register2 - 1
  counter = register2
  ```

- Consider this execution interleaving with "count = 5" initially:

S0: producer execute	`register1 = counter`	{register1 = 5}
S1: producer execute	`register1 = register1 + 1`	{register1 = 6}
S2: consumer execute	`register2 = counter`	{register2 = 5}
S3: consumer execute	`register2 = register2 - 1`	{register2 = 4}
S4: producer execute	`counter = register1`	{counter = 6 }
S5: consumer execute	`counter = register2`	{counter = 4}

Race Condition

- Processes P_0 and P_1 are creating child processes using the `fork()` system call
- Race condition on kernel variable `next_available_pid` which represents the next available process identifier (pid)



- Unless there is a mechanism to prevent P_0 and P_1 from accessing the variable `next_available_pid` the same pid could be assigned to two different processes!

Critical Section Problem

- Consider system of ***n*** processes {$p_0, p_1, \ldots p_{n-1}$}
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc.
  - When one process in critical section, no other may be in its critical section
- ***Critical section problem*** is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

Critical Section

- General structure of process P_i

```
while (true) {

        entry section

            critical section

        exit section

            remainder section

}
```

Critical-Section Problem (Cont.)

Requirements for solution to critical-section problem

1. **Mutual Exclusion** - If process ***P_i*** is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the process that will enter the critical section next cannot be postponed indefinitely
3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
   - Assume that each process executes at a nonzero speed
   - No assumption concerning **relative speed** of the ***n*** processes

Definitions

- **Critical resource (منبع بحرانی)** : a shared resource between more than one threads (processes) that can not be used or updated concurrently by them.
- **Critical section (ناحیه بحرانی):** a section of code for updating a critical resource. Ex: write to printer buffer, updating a table in database, writing to a file
- **Race condition (شرایط رقابتی):** a situation where several processes access and manipulate the same data (a critical resource) concurrently.
- **Synchronization (همگام‌سازی):** Orderly execution of cooperating processes that share a critical resource
- **Deadlock (بن‌بست):** two or more processes are blocked with each other to enter the critical section and progress execution.
- **Starvation (گرسنگی):** a process wait indefinitely for entering the critical section

Critical-Section Handling in OS

Two approaches depending on if kernel is preemptive or non-preemptive

- **Preemptive** قبــضه شــدنی یــا غیرانحصاری – allows preemption of process when running in kernel mode
- **Non-preemptive** قبــضه نــشدنی یــا انــحصاری– runs until exits kernel mode, blocks, or voluntarily yields CPU
  - Essentially free of race conditions in kernel mode
- preemptive kernels are difficult to design especially in SMP (Symmetric Multi-Processing), but they are more responsive and suitable for real-time

Interrupt-based Solution

- Entry section: disable interrupts
- Exit section: enable interrupts
- Will this solve the problem?
  - What if the critical section is code that runs for an hour?
  - Can some processes starve – never enter their critical section.
  - What if there are two CPUs?

Software Solution 1

- Two process solution
- Assume that the **`load`** and **`store`** machine-language instructions are atomic; that is, cannot be interrupted
- The two processes share one variable:
  - **`int turn;`**
- The variable **`turn`** indicates whose turn it is to enter the critical section
- initially, the value of **`turn`** is set to *i*

Algorithm for Process P_i

```
while (true){

    while (turn = = j);

    /* critical section */

    turn = j;

    /* remainder section */

}
```

Correctness of the Software Solution

- Mutual exclusion is preserved

  P_i enters critical section only if:

  `turn = i`

  and **`turn`** cannot be both 0 and 1 at the same time
- What about the Progress requirement?
- What about the Bounded-waiting requirement?

Peterson’s Solution

- Two process solution
- Assume that the **`load`** and **`store`** machine-language instructions are atomic; that is, cannot be interrupted
- The two processes share two variables:
  - **`int turn;`**
  - **`boolean flag[2]`**

- The variable **`turn`** indicates whose turn it is to enter the critical section
- The **`flag`** array is used to indicate if a process is ready to enter the critical section.
  - **`flag[i] = `*`true`*** implies that process $\mathbf{P}_{\mathtt{i}}$ is ready!

Algorithm for Process P_i and P_j

P_i

```
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
     critical section
   flag[i] = false;
     remainder section
} while (true);
```

P_j

```
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
     critical section
   flag[j] = false;
     remainder section
} while (true);
```

Peterson’s Solution (Cont.)

- Provable that the three CS requirement are met:
  1. Mutual exclusion is preserved

     `P`$_i$ enters CS only if:

     either **`flag[j] = false`** or **`turn = i`**
  2. Progress requirement is satisfied

     after critical section flag[i] set to be false
  3. Bounded-waiting requirement is met

     it is achieved through setting turn correctly
- Peterson’s solution is not guaranteed to work on modern computer architectures
  - Because of reordering read and write operations that have no dependencies

Peterson’s Solution and Modern Architecture

- Although useful for demonstrating an algorithm, Peterson’s Solution is not guaranteed to work on modern architectures.
  - To improve performance, processors and/or compilers may reorder operations that have no dependencies
- Understanding why it will not work is useful for better understanding race conditions.
- For single-threaded this is ok as the result will always be the same.
- For multithreaded the reordering may produce inconsistent or unexpected results!

Modern Architecture Example

- Two threads share the data:

```
boolean flag = false;
int x = 0;
```

- Thread 1 performs

```
while (!flag)
 ;
print x
```

- Thread 2 performs

```
x = 100;
flag = true
```

- What is the expected output?

    100

Modern Architecture Example (Cont.)

- However, since the variables `flag` and `x` are independent of each other, the instructions:

```
flag = true;
 x = 100;
```

  for Thread 2 may be reordered
- If this occurs, the output may be 0!

Reordering problem for peterson

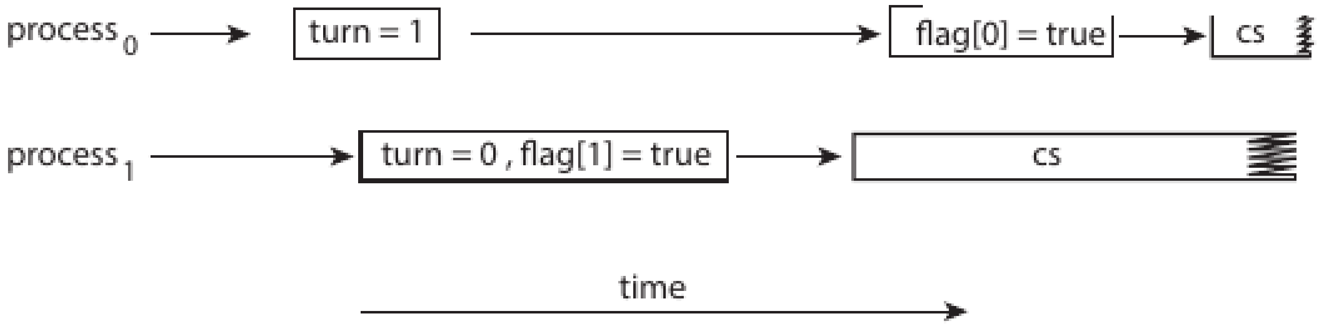
```
boolean flag = false;
int x = 0;
```

Thread 1

```
while (!flag)
   ;
print x;
```

Thread 2

```
x = 100;
flag = true;
```



This allows both processes to be in their critical section at the same time! To ensure that Peterson's solution will work correctly on modern computer architecture we must use **Memory Barrier**.

Memory Barrier

How a computer architecture determines what memory guarantees it will provide to an application program is known as its **memory model**

- Memory models may be either:
  - **Strongly ordered** – where a memory modification of one processor is immediately visible to all other processors.
  - **Weakly ordered** – where a memory modification of one processor may not be immediately visible to all other processors.
- A **memory barrier** is an instruction that forces any change in memory to be propagated (made visible) to all other processors.

Memory Barrier Instructions

- When a memory barrier instruction is performed, the system ensures that all loads and stores are completed before any subsequent load or store operations are performed.
- Therefore, even if instructions were reordered, the memory barrier ensures that the store operations are completed in memory and visible to other processors before future load or store operations are performed.

Memory Barrier Example

- Returning to the example of slides 6.24 - 6.25
- We could add a memory barrier to the following instructions to ensure Thread 1 outputs 100:
- Thread 1 now performs

```
  while (!flag)
   memory_barrier();
 print x
```

- Thread 2 now performs

```
  x = 100;
 memory_barrier();
 flag = true
```

- For Thread 1 we are guaranteed that that the value of `flag` is loaded before the value of `x`.
- For Thread 2 we ensure that the assignment to `x` occurs before the assignment `flag`.

Check Critical section requirements

```
P1:
while ( true )  {
        flag [1] = true ;
        turn = 1 ;
        while ( flag [2 ] and turn=2);
        <Critical - Section >
       flag [1] = false ;
       < remainder >
}
```

```
P2:
 while ( true ) {
      flag [2 ] = true ;
      turn = 2 ;
      while ( flag [1] and turn=1);
              < Critical - Section >
       flag [2 ] = false ;
      < remainder >
   }
```

Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- We will look at three forms of hardware support:
  1) Memory barriers
  2) Hardware instructions
  3) Atomic variables

Hardware Instructions

- Special hardware instructions that allow us to either *test-and-modify* the content of a word, or to *swap* the contents of two words atomically (uninterruptedly.)
  - **Test-and-Set** instruction: Either **test** memory word and **set** value
  - **Compare-and-Swap** instruction: Or **swap contents** of two memory words

The test_and_set Instruction

- Definition

```
boolean test_and_set (boolean *target)
  {
           boolean rv = *target;
           *target = true;
           return rv:
  }
```

- Properties
  - Executed atomically
  - Returns the original value of passed parameter
  - Set the new value of passed parameter to **`true`**

Solution Using test_and_set()

- Shared boolean variable **`lock`**, initialized to **`false`**
- Solution:

```
    do {
            while (test_and_set(&lock))
             ; /* do nothing */

                 /* critical section */

         lock = false;
                 /* remainder section */
     } while (true);
```

While(lock==true);
lock=true;

- Does it solve the critical-section problem?

This solution does not satisfy bounded waiting: starvation

The compare_and_swap Instruction

- Definition

```
int compare_and_swap(int *value, int expected, int new_value)
 {
   int temp = *value;
   if (*value == expected)
       *value = new_value;
    return temp;
  }
```

- Properties
  - Executed atomically
  - Returns the original value of passed parameter **`value`**
  - Set the variable **`value`** the value of the passed parameter **`new_value`** but only if **`*value == expected`** is true. That is, the swap takes place only under this condition.

Intel x86 instruction
cmpxchg <destination operand>, <source operand>

Solution using compare_and_swap

- Shared integer **`lock`** initialized to 0;
- Solution:

```
while (true){
        while (compare_and_swap(&lock, 0, 1) != 0)
        ; /* do nothing */

        /* critical section */

        lock = 0;

    /* remainder section */
}
```

- Does it solve the critical-section problem?

Bounded-waiting with compare-and-swap

```
while (true) {
   waiting[i] = true;
   key = 1;
   while (waiting[i] && key == 1)
      key = compare_and_swap(&lock,0,1);
   waiting[i] = false;
   /* critical section */
   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;
   if (j == i)
      lock = 0;
   else
      waiting[j] = false;
   /* remainder section */
}
```

Atomic Variables

- Typically, instructions such as compare-and-swap are used as building blocks for other synchronization tools.
- One tool is an **atomic variable** that provides *atomic* (uninterruptible) updates on basic data types such as integers and booleans.
- For example:
  - Let **`sequence`** be an atomic variable
  - Let **`increment()`** be operation on the atomic variable **`sequence`**
  - The Command:

```
increment(&sequence);
```

ensures **`sequence`** is incremented without interruption:

Atomic Variables

- The **increment()** function can be implemented as follows:

```
void increment(atomic_int *v)
{
    int temp;
    do {
        temp = *v;
    }
    while (temp != (compare_and_swap(v,temp,temp+1));
}
```

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
- Protect a critical section by first **`acquire()`** a lock then **`release()`** the lock
  - Boolean variable indicating if lock is available or not
- Calls to **`acquire()`** and **`release()`** must be **atomic**
  - Usually implemented via hardware atomic instructions such as compare-and-swap.

```
do {
    acquire lock
        critical section
    release lock
        remainder section
 } while (true);
```

acquire() and release()

```
   acquire() {
       while (!available) ; /* busy wait */
       available = false;
    }
   release() {
       available = true;
   }
```

-
  ```
     do {
   acquire lock
       critical section
   release lock
      remainder section
   } while (true);
  ```

Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for processes to synchronize their activities.
- Semaphore ***S*** – integer variable
- Can only be accessed via two indivisible (atomic) operations
  - **`wait()`** and **`signal()`**
    - Originally called **`P()`** and **`V()`**
- Definition of the **`wait() operation`**

```
wait(S) {
    while (S <= 0)
       ; // busy wait
    S--;
}
```

- Definition of the **`signal() operation`**

```
signal(S) {
    S++;
}
```

Semaphore (Cont.)

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a **mutex lock**
- Can implement a counting semaphore ***S*** as a binary semaphore
- With semaphores we can solve various synchronization problems

Semaphore Usage Example

- Solution to the CS Problem
  - Create a semaphore "**mutex**" initialized to 1

```
wait(mutex);
    CS
signal(mutex);
```

- Consider P_1 and P_2 that with two statements S_1 and S_2 and the requirement that S_1 to happen before S_2
  - Create a semaphore "**synch**" initialized to 0

```
P1:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```

Example: Mistakes using semaphores

- Let ***S*** and ***Q*** be two semaphores initialized to 1, What happens executing P0 and P1 ?

P_0

```
wait(S);
wait(Q);
...
signal(S);
signal(Q);
```

P_1

```
wait(Q);
wait(S);
...
signal(Q);
signal(S);
```

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes

Example: Bounded-Buffer Problem

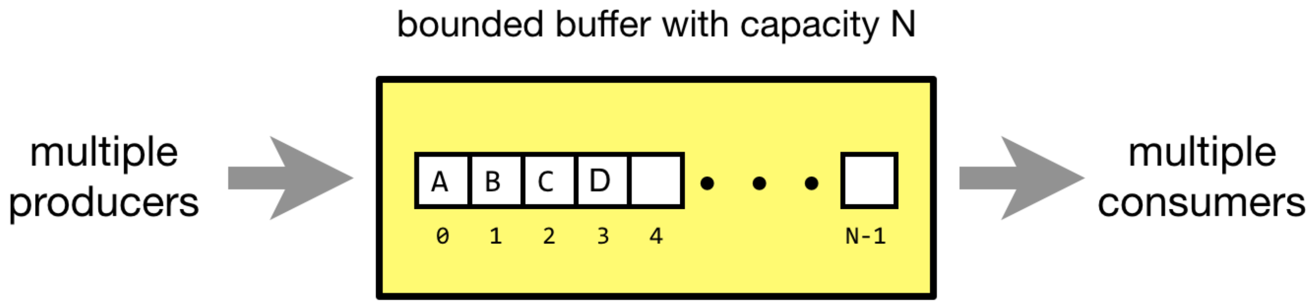
- ***n*** buffers, each can hold one item
- Semaphore **`mutex`** initialized to the value 1
- Semaphore **`full`** initialized to the value 0
- Semaphore **`empty`** initialized to the value n

Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem

Bounded-Buffer Problem

- ***n*** buffers, each can hold one item
- Semaphore **`mutex`** initialized to the value 1
- Semaphore **`full`** initialized to the value 0
- Semaphore **`empty`** initialized to the value n



Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
while (true) {
      ...
      /* produce an item in next_produced */




       ...
      /* add next produced to the buffer */
       ...



}
```

Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
while (true) {
     ...
     /* produce an item in next_produced */
     ...
   wait(empty);
   wait(mutex);
      ...
     /* add next produced to the buffer */
      ...
   signal(mutex);
   signal(full);
}
```

Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```
while (true) {



        ...
     /* remove an item from buffer to next_consumed */
        ...



       /* consume the item in next consumed */
        ...
    }
```

Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```
while (true) {
    wait(full);
    wait(mutex);
        ...
     /* remove an item from buffer to next_consumed */
        ...
    signal(mutex);
    signal(empty);
        ...
      /* consume the item in next consumed */
        ...
   }
```

Semaphore Implementation

- Must guarantee that no two processes can execute the **`wait()`** and **`signal()`** on the same semaphore at the same time
- Thus, the implementation becomes the critical section problem where the **`wait`** and **`signal`** code are placed in the critical section
- Could now have **busy waiting** in critical section implementation
  - But implementation code is short
  - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
  - Value (of type integer)
  - Pointer to next record in the list
- Two operations:
  - **block** – place the process invoking the operation on the appropriate waiting queue
  - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue

Implementation with no Busy waiting (Cont.)

- Waiting queue

```
typedef struct {
  int value;
  struct process *list;
} semaphore;
```

Implementation with no Busy waiting (Cont.)

```
wait(semaphore *S) {
   S->value--;
   if (S->value < 0) {
      add this process to S->list;
      block();
   }
}

signal(semaphore *S) {
   S->value++;
   if (S->value <= 0) {
      remove a process P from S->list;
      wakeup(P);
   }
}
```