Synchronization: Basics

15-213: Introduction to Computer Systems
24th Lecture, July 26, 2018

Instructor:
Brian Railing
Today

- Threads review
- Sharing
- Mutual exclusion
- Semaphores
Traditional View of a Process

- Process = process context + code, data, and stack

**Process context**

- Program context:
  - Data registers
  - Condition codes
  - Stack pointer (SP)
  - Program counter (PC)

- Kernel context:
  - VM structures
  - Descriptor table
  - brk pointer

**Code, data, and stack**

- Stack
- Shared libraries
- Run-time heap
- Read/write data
- Read-only code/data

SP → Stack
brk → Shared libraries
PC → Run-time heap
0 → Read/write data

Alternate View of a Process

Process = thread + code, data, and kernel context

Thread (main thread)

- Stack
- Thread context:
  - Data registers
  - Condition codes
  - Stack pointer (SP)
  - Program counter (PC)

Code, data, and kernel context

- Shared libraries
- Run-time heap
- Read/write data
- Read-only code/data

Kernel context:
- VM structures
- Descriptor table
- brk pointer
A Process With Multiple Threads

- Multiple threads can be associated with a process
  - Each thread has its own logical control flow
  - Each thread shares the same code, data, and kernel context
  - Each thread has its own stack for local variables
    - but not protected from other threads
  - Each thread has its own thread id (TID)

Thread 1 (main thread)  Thread 2 (peer thread)

<table>
<thead>
<tr>
<th>Thread 1 context:</th>
<th>Thread 2 context:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data registers</td>
<td>Data registers</td>
</tr>
<tr>
<td>Condition codes</td>
<td>Condition codes</td>
</tr>
<tr>
<td>$SP_1$</td>
<td>$SP_2$</td>
</tr>
<tr>
<td>$PC_1$</td>
<td>$PC_2$</td>
</tr>
</tbody>
</table>

| stack 1                  | stack 2                  |

Shared code and data

- shared libraries
- run-time heap
- read/write data
- read-only code/data
- Kernel context:
  - VM structures
  - Descriptor table
  - brk pointer
Shared Variables in Threaded C Programs

Question: Which variables in a threaded C program are shared?

- The answer is not as simple as “global variables are shared” and “stack variables are private”

Def: A variable $x$ is shared if and only if multiple threads reference some instance of $x$.

Requires answers to the following questions:

- What is the memory model for threads?
- How are instances of variables mapped to memory?
- How many threads might reference each of these instances?
Threads Memory Model

**Conceptual model:**
- Multiple threads run within the context of a single process
- Each thread has its own separate thread context
  - Thread ID, stack, stack pointer, PC, condition codes, and GP registers
- All threads share the remaining process context
  - Code, data, heap, and shared library segments of the process virtual address space
  - Open files and installed handlers

**Operationally, this model is not strictly enforced:**
- Register values are truly separate and protected, but…
- Any thread can read and write the stack of any other thread

*The mismatch between the conceptual and operation model is a source of confusion and errors*
Example Program to Illustrate Sharing

```c
char **ptr; /* global var */

int main(int argc, char *argv[]) {
    long i;
    pthread_t tid;
    char *msgs[2] = {
        "Hello from foo",
        "Hello from bar"
    };
    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid,
                        NULL,
                        thread,
                        (void *)i);
    Pthread_exit(NULL);
}

void *thread(void *vargp) {
    long myid = (long)vargp;
    static int cnt = 0;

    printf("[%ld]:  %s (cnt=%d)\n",
           myid, ptr[myid], ++cnt);
    return NULL;
}
```

Peer threads reference main thread’s stack indirectly through global ptr variable
Mapping Variable Instances to Memory

- **Global variables**
  - *Def*: Variable declared outside of a function
  - Virtual memory contains exactly one instance of any global variable

- **Local variables**
  - *Def*: Variable declared inside function without `static` attribute
  - Each thread stack contains one instance of each local variable

- **Local static variables**
  - *Def*: Variable declared inside function with the `static` attribute
  - Virtual memory contains exactly one instance of any local static variable.
Mapping Variable Instances to Memory

**Global var**: 1 instance (ptr [data])

```c
char **ptr; /* global var */
```

**Local vars**: 1 instance (i.m, msgs.m, tid.m)

```c
int main(int main, char *argv[])
{
    long i;
    pthread_t tid;
    char *msgs[2] = {
        "Hello from foo",
        "Hello from bar"
    };

    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid,
                        NULL,
                        thread,
                        (void *)i);
    Pthread_exit(NULL);
}
```

**Local var**: 2 instances (myid.p0 [peer thread 0’s stack], myid.p1 [peer thread 1’s stack])

```c
void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;

    printf("[%ld]: %s (cnt=%d)\n",
           myid, ptr[myid], ++cnt);
    return NULL;
}
```

**Local static var**: 1 instance (cnt [data])

```c
sharing.c
```
Shared Variable Analysis

Which variables are shared?

<table>
<thead>
<tr>
<th>Variable instance</th>
<th>Referenced by main thread?</th>
<th>Referenced by peer thread 0?</th>
<th>Referenced by peer thread 1?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptr</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>cnt</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>i.m</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>msgs.m</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>myid.p0</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>myid.p1</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

```c
char **ptr; /* global */
int main(int argc, char *argv[]) {
    int i;
    pthread_t tid;
    char *msgs[2] = {"Hello from foo", "Hello from bar"};
    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid,..., (void *)i);

    pthread_exit(NULL);
}

/* thread routine */
void *thread(void *vargp)
{
    int myid = (int)vargp;
    static int cnt = 0;
    printf("[%d]: %s (cnt=%d)\n", myid, ptr[myid], ++cnt);
    return NULL;
}
```
Shared Variable Analysis

Which variables are shared?

<table>
<thead>
<tr>
<th>Variable instance</th>
<th>Referenced by main thread?</th>
<th>Referenced by peer thread 0?</th>
<th>Referenced by peer thread 1?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptr</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>cnt</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>i.m</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>msgs.m</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>myid.p0</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>myid.p1</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

Answer: A variable \( x \) is shared iff multiple threads reference at least one instance of \( x \). Thus:

- \( \text{ptr, cnt, and msgs are shared} \)
- \( \text{i and myid are not shared} \)
Synchronizing Threads

- Shared variables are handy...

- …but introduce the possibility of nasty *synchronization* errors.
badcnt.c: Improper Synchronization

```c
/* Global shared variable */
volatile long cnt = 0; /* Counter */

int main(int argc, char **argv)
{
    long niters;
    pthread_t tid1, tid2;

    niters = atoi(argv[1]);
    Pthread_create(&tid1, NULL, thread, &niters);
    Pthread_create(&tid2, NULL, thread, &niters);
    Pthread_join(tid1, NULL);
    Pthread_join(tid2, NULL);

    /* Check result */
    if (cnt != (2 * niters))
        printf("BOOM! cnt=%ld\n", cnt);
    else
        printf("OK cnt=%ld\n", cnt);
    exit(0);
}
```

```c
/* Thread routine */
void *thread(void *vargp)
{
    long i, niters = *(long *)vargp;

    for (i = 0; i < niters; i++)
        cnt++;

    return NULL;
}
```

```bash
linux> ./badcnt 10000
OK cnt=20000
linux> ./badcnt 10000
BOOM! cnt=13051
linux>
```

cnt should equal 20,000.

What went wrong?
Assembly Code for Counter Loop

C code for counter loop in thread i

```c
for (i = 0; i < niters; i++)
    cnt++;
```

```
movq  (%rdi), %rcx
testq %rcx,%rcx
jle   .L2
movl  $0, %eax
.L3:
movq  cnt(%rip),%rdx
addq  $1, %rdx
movq  %rdx, cnt(%rip)
addq  $1, %rax
cmpq  %rcx, %rax
jne   .L3
.L2:
```

\[ H_i : \text{Head} \]
\[ L_i : \text{Load cnt} \]
\[ U_i : \text{Update cnt} \]
\[ S_i : \text{Store cnt} \]
\[ T_i : \text{Tail} \]
## Concurrent Execution

**Key idea:** In general, any sequentially consistent interleaving is possible, but some give an unexpected result!

- $l_i$ denotes that thread $i$ executes instruction $l$
- $\%rdx_i$ is the content of $\%rdx$ in thread $i$’s context

<table>
<thead>
<tr>
<th>i (thread)</th>
<th>instr&lt;sub&gt;i&lt;/sub&gt;</th>
<th>$%rdx_1$</th>
<th>$%rdx_2$</th>
<th>cnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$H_1$</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>$L_1$</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>$U_1$</td>
<td>1</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>$S_1$</td>
<td>1</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>$H_2$</td>
<td>-</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>$L_2$</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>$U_2$</td>
<td>-</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>$S_2$</td>
<td>-</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>$T_2$</td>
<td>-</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>$T_1$</td>
<td>1</td>
<td>-</td>
<td>2</td>
</tr>
</tbody>
</table>

*OK*
Concurrent Execution

**Key idea:** In general, any sequentially consistent interleaving is possible, but some give an unexpected result!

- $i_j$ denotes that thread $i$ executes instruction $j$
- $%rdx_i$ is the content of $%rdx$ in thread $i$’s context

<table>
<thead>
<tr>
<th>$i$ (thread)</th>
<th>instr$_i$</th>
<th>$%rdx_1$</th>
<th>$%rdx_2$</th>
<th>cnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$H_1$</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>$L_1$</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>$U_1$</td>
<td>1</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>$S_1$</td>
<td>1</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>$H_2$</td>
<td>-</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>$L_2$</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>$U_2$</td>
<td>-</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>$S_2$</td>
<td>-</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>$T_2$</td>
<td>-</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>$T_1$</td>
<td>1</td>
<td>-</td>
<td>2</td>
</tr>
</tbody>
</table>

Thread 1
- critical section

Thread 2
- critical section

OK
Incorrect ordering: two threads increment the counter, but the result is 1 instead of 2

<table>
<thead>
<tr>
<th>i (thread)</th>
<th>instr _i</th>
<th>%rdx_1</th>
<th>%rdx_2</th>
<th>cnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>H_1</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>L_1</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>U_1</td>
<td>1</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>H_2</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>L_2</td>
<td>-</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>S_1</td>
<td>1</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>T_1</td>
<td>1</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>U_2</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>S_2</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>T_2</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Oops!
Concurrent Execution (cont)

- How about this ordering?

<table>
<thead>
<tr>
<th>i (thread)</th>
<th>instr&lt;sub&gt;i&lt;/sub&gt;</th>
<th>%rdx&lt;sub&gt;1&lt;/sub&gt;</th>
<th>%rdx&lt;sub&gt;2&lt;/sub&gt;</th>
<th>cnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>H&lt;sub&gt;1&lt;/sub&gt;</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>L&lt;sub&gt;1&lt;/sub&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>H&lt;sub&gt;2&lt;/sub&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>L&lt;sub&gt;2&lt;/sub&gt;</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>U&lt;sub&gt;2&lt;/sub&gt;</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>S&lt;sub&gt;2&lt;/sub&gt;</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>U&lt;sub&gt;1&lt;/sub&gt;</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>S&lt;sub&gt;1&lt;/sub&gt;</td>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>T&lt;sub&gt;1&lt;/sub&gt;</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>T&lt;sub&gt;2&lt;/sub&gt;</td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

Oops!

- We can analyze the behavior using a progress graph.
A *progress graph* depicts the discrete *execution state space* of concurrent threads.

Each axis corresponds to the sequential order of instructions in a thread.

Each point corresponds to a possible *execution state* \((\text{Inst}_1, \text{Inst}_2)\).

E.g., \((L_1, S_2)\) denotes state where thread 1 has completed \(L_1\) and thread 2 has completed \(S_2\).
**Trajectories in Progress Graphs**

A *trajectory* is a sequence of legal state transitions that describes one possible concurrent execution of the threads.

Example:

H1, L1, U1, H2, L2, S1, T1, U2, S2, T2
A *trajectory* is a sequence of legal state transitions that describes one possible concurrent execution of the threads.

Example:

H1, L1, U1, H2, L2, S1, T1, U2, S2, T2
Critical Sections and Unsafe Regions

L, U, and S form a critical section with respect to the shared variable cnt. Instructions in critical sections (wrt some shared variable) should not be interleaved. Sets of states where such interleaving occurs form unsafe regions.
**Def:** A trajectory is *safe* iff it does not enter any unsafe region

**Claim:** A trajectory is *correct (wrt cnt)* iff it is *safe*
badcnt.c: Improper Synchronization

```c
/* Global shared variable */
volatile long cnt = 0; /* Counter */

int main(int argc, char **argv)
{
    long niters;
    pthread_t tid1, tid2;

    niters = atoi(argv[1]);
    Pthread_create(&tid1, NULL,
                   thread, &niters);
    Pthread_create(&tid2, NULL,
                   thread, &niters);
    Pthread_join(tid1, NULL);
    Pthread_join(tid2, NULL);

    /* Check result */
    if (cnt != (2 * niters))
        printf("BOOM! cnt=%ld\n", cnt);
    else
        printf("OK cnt=%ld\n", cnt);
    exit(0);
}

/* Thread routine */
void *thread(void *vargp)
{
    long i, niters = *((long *)vargp);
    for (i = 0; i < niters; i++)
        cnt++;  
    return NULL;
}
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>main</th>
<th>thread1</th>
<th>thread2</th>
</tr>
</thead>
<tbody>
<tr>
<td>cnt</td>
<td>yes*</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>niters.m</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>tid1.m</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>i.1</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>i.2</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>niters.1</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>niters.2</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>
Enforcing Mutual Exclusion

- **Question:** How can we guarantee a safe trajectory?

- **Answer:** We must *synchronize* the execution of the threads so that they can never have an unsafe trajectory.
  - i.e., need to guarantee *mutually exclusive access* for each critical section.

- **Classic solution:**
  - Semaphores (Edsger Dijkstra)

- **Other approaches (out of our scope)**
  - Mutex and condition variables (Pthreads)
  - Monitors (Java)
Semaphores

- **Semaphore**: non-negative global integer synchronization variable. Manipulated by $P$ and $V$ operations.
- $P(s)$
  - If $s$ is nonzero, then decrement $s$ by 1 and return immediately.
    - Test and decrement operations occur atomically (indivisibly)
  - If $s$ is zero, then suspend thread until $s$ becomes nonzero and the thread is restarted by a $V$ operation.
  - After restarting, the $P$ operation decrements $s$ and returns control to the caller.
- $V(s)$:
  - Increment $s$ by 1.
    - Increment operation occurs atomically
  - If there are any threads blocked in a $P$ operation waiting for $s$ to become non-zero, then restart exactly one of those threads, which then completes its $P$ operation by decrementing $s$.
- **Semaphore invariant**: $(s \geq 0)$
Semaphores

- **Semaphore**: non-negative global integer synchronization variable

- Manipulated by \( P \) and \( V \) operations:
  - \( P(s) \): [ while (s == 0) wait(); s--; ]
    - Dutch for "Proberen" (test)
  - \( V(s) \): [ s++; ]
    - Dutch for "Verhogen" (increment)

- OS kernel guarantees that operations between brackets [ ] are executed indivisibly
  - Only one \( P \) or \( V \) operation at a time can modify \( s \).
  - When `while` loop in \( P \) terminates, only that \( P \) can decrement \( s \)

- **Semaphore invariant**: \( (s >= 0) \)
C Semaphore Operations

Pthreads functions:

```c
#include <semaphore.h>

int sem_init(sem_t *s, 0, unsigned int val);} /* s = val */

int sem_wait(sem_t *s); /* P(s) */
int sem_post(sem_t *s); /* V(s) */
```

CS:APP wrapper functions:

```c
#include "csapp.h"

void P(sem_t *s); /* Wrapper function for sem_wait */
void V(sem_t *s); /* Wrapper function for sem_post */
```
/* Global shared variable */
volatile long cnt = 0; /* Counter */

int main(int argc, char **argv)
{
    long niter;
    pthread_t tid1, tid2;
    
    niter = atoi(argv[1]);
    Pthread_create(&tid1, NULL,
            thread, &niter);
    Pthread_create(&tid2, NULL,
            thread, &niter);
    Pthread_join(tid1, NULL);    
    Pthread_join(tid2, NULL);
    
    /* Check result */
    if (cnt != (2 * niter))
        printf("BOOM! cnt=%ld\n", cnt);
    else
        printf("OK cnt=%ld\n", cnt);
    exit(0);
}

/* Thread routine */
void *thread(void *vargp)
{
    long i, niter = *
            ((long *)vargp);
    for (i = 0; i < niter; i++)
        cnt++;
    return NULL;
}

How can we fix this using semaphores?
Using Semaphores for Mutual Exclusion

- **Basic idea:**
  - Associate a unique semaphore *mutex*, initially 1, with each shared variable (or related set of shared variables).
  - Surround corresponding critical sections with *P(mutex)* and *V(mutex)* operations.

- **Terminology:**
  - *Binary semaphore*: semaphore whose value is always 0 or 1
  - *Mutex*: binary semaphore used for mutual exclusion
    - P operation: “locking” the mutex
    - V operation: “unlocking” or “releasing” the mutex
      - “Holding” a mutex: locked and not yet unlocked.
  - *Counting semaphore*: used as a counter for set of available resources.
goodcnt.c: Proper Synchronization

- Define and initialize a mutex for the shared variable `cnt`:

```c
volatile long cnt = 0; /* Counter */
sem_t mutex; /* Semaphore that protects cnt */
sem_init(&mutex, 0, 1); /* mutex = 1 */
```

- Surround critical section with `P` and `V`:

```c
for (i = 0; i < niters; i++) {
    P(&mutex);
    cnt++;
    V(&mutex);
}
```

Warning: It’s orders of magnitude slower than `badcnt.c`.
**goodcnt.c: Proper Synchronization**

- Define and initialize a mutex for the shared variable `cnt`:

```c
volatile long cnt = 0; /* Counter */
sem_t mutex; /* Semaphore that protects cnt */
sem_init(&mutex, 0, 1); /* mutex = 1 */
```

- Surround critical section with `P` and `V`:

```c
for (i = 0; i < niters; i++) {
P(&mutex);
cnt++;
V(&mutex);
}
```

```
linux> ./goodcnt 10000
OK cnt=20000

OK cnt=2000000
BOOM! cnt=1036525 Slowdown
```

```
real 0m0.138s 0m0.007s 20X
user 0m0.120s 0m0.008s 15X
sys 0m0.108s 0m0.000s NaN
```

And slower means much slower!
Why Mutexes Work

Provide mutually exclusive access to shared variable by surrounding critical section with $P$ and $V$ operations on semaphore $s$ (initially set to 1)

Initially $s = 1$
Why Mutexes Work

Provide mutually exclusive access to shared variable by surrounding critical section with $P$ and $V$ operations on semaphore $s$ (initially set to 1)

Semaphore invariant creates a forbidden region that encloses unsafe region and that cannot be entered by any trajectory.

Initially $s = 1$.
Why Mutexes Work

Provide mutually exclusive access to shared variable by surrounding critical section with $P$ and $V$ operations on semaphore $s$ (initially set to 1)

Semaphore invariant creates a forbidden region that encloses unsafe region and that cannot be entered by any trajectory.
Why Mutexes Work

Provide mutually exclusive access to shared variable by surrounding critical section with \( P \) and \( V \) operations on semaphore \( s \) (initially set to 1).

Semaphore invariant creates a forbidden region that encloses unsafe region and that cannot be entered by any trajectory.
Summary

- Programmers need a clear model of how variables are shared by threads.

- Variables shared by multiple threads must be protected to ensure mutually exclusive access.

- Semaphores are a fundamental mechanism for enforcing mutual exclusion.