Concurrency and Synchronization

CMPSCI 230: Computer Systems Principles
int pthread_join (pthread_t thread, void **value_ptr) {
    int result;
    ptw32_thread_t * tp = (ptw32_thread_t *) thread.p;
    ...
    if (NULL == tp
        || thread.x != tp->ptHandle.x){
        result = ESRCH;
    }
    else if (PTHREAD_CREATE_DETACHED == tp->detachState){
        result = EINVAL;
    }
    else {
        result = 0;
    }
    if (result == 0) {
        result = pthreadCancelableWait (tp->threadH);
        ...
        *value_ptr = tp->exitStatus;
        result = pthread_detach (thread);
    }
    return (result);
}
int pthread_join (pthread_t thread, void **value_ptr) {
    int result;
    ptw32_thread_t * tp = (ptw32_thread_t *) thread.p;
    ...
    if (NULL == tp
        || thread.x != tp->ptHandle.x){
        result = ESRCH;
    }
    else if (PTHREAD_CREATE_DETACHED == tp->detachState){
        result = EINVAL;
    }
    else {
        result = 0;
    }
    if (result == 0) {
        result = pthreadCancelableWait (tp->threadH);
        ...
        *value_ptr = tp->exitStatus;
        result = pthread_detach (thread);
    }
    return (result);
}
Typical way a function return two values

```c
int pthread_join (pthread_t thread, void **value_ptr) {
  int result;
  ptw32_thread_t * tp = (ptw32_thread_t *) thread.p;
  ...
  if (NULL == tp || thread.x != tp->ptHandle.x)
    result = ESRCH;
  else if (PTHREAD_CREATE_DETACHED == tp->detachState)
    result = EINVAL;
  else {
    result = 0;
  }
  if (result == 0) {
    result = pthreadCancelableWait (tp->threadH);
    ...
    *value_ptr = tp->exitStatus;
    result = pthread_detach (thread);
  }
  return (result);
}
```
To change a pointer even outside of a function, you need to use double pointer.
Concurrency and Synchronization

CMPSCI 230: Computer Systems Principles
Today

- Sharing
- Mutual exclusion
- Semaphores
Shared Variables in Threaded C Programs

- **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
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 */

int main()
{
    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);
}

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

    printf("[%d]: %s (svar=%d)\n",
           myid, ptr[myid], ++cnt);
}
```
Example Program to Illustrate Sharing

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

int main()
{
    long i;
    pthread_t tid;
    char *msgs[2] = {
        "Hello from foo",
        "Hello from bar"
    }; //D0
    ptr = msgs;
    for (i = 0; i < 2; i++)
        pthread_create(&tid, NULL, thread, (void *)(i));
    pthread_exit(NULL);
}

/* thread routine */
void *thread(void *vargp)
{
    long myid = (long) (vargp);
    static int cnt = 0;
    printf("[%d]: %s (svar=%d)\n", myid, ptr[myid], ++cnt);
}

pthread_exit
vs
pthread_join:
Do you need further processing in main thread?
```
Example Program to Illustrate Sharing

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

int main()
{
    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);
}

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

    printf("[%d]: %s (svar=%d)\n",
           myid, ptr[myid], ++cnt);
}
```
Example Program to Illustrate Sharing

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

int main()
{
    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);
}

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

static acts to extend the lifetime of a variable to the lifetime of the process
Example Program to Illustrate Sharing

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

int main()
{
    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);
}

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

    printf("[%d]: %s (svar=%d)\n", myid, ptr[myid], ++cnt);
}
```
Example Program to Illustrate Sharing

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

int main()
{
    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);
}

/* thread routine */
void *thread(void *vargp)
{
    long myid = (long)(vargp);
    static int cnt = 0;
    
    printf("[%d]: %s (svar=%d)\n", myid, ptr[myid], ++cnt);
}

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 */

int main() {
    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,
                        NULL,
                        thread,
                        (void *)i);
    Pthread_exit(NULL);
}

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

sharing.c
Mapping Variable Instances to Memory

char **ptr; /* global */

int main() {
    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, NULL, thread, (void *)i);
    Pthread_exit(NULL);
}

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

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

sharing.c
Mapping Variable Instances to Memory

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

int main()
{
    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,
    NULL,
    thread,
    (void *)i);
Pthread_exit(NULL);
}

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

Local var: 2 instances (myid.p0 [peer thread 0’s stack], myid.p1 [peer thread 1’s stack])
Mapping Variable Instances to Memory

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

int main() {
    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, NULL, thread, (void *)i);
    Pthread_exit(NULL);
}

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

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

What if declare myid static?
```

/* sharing.c*/
Mapping Variable Instances to Memory

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

int main()
{
    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, NULL, thread, (void *)i);
    Pthread_exit(NULL);
}

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

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

sharing.c
Mapping Variable Instances to Memory

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

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

int main()
{
    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,
                        NULL,
                        thread,
                        (void *)i);
    Pthread_exit(NULL);
}
```

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

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

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

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

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>

■ A variable $x$ is shared iff multiple threads reference at least one instance of $x$. Thus:

■ $ptr$, $cnt$, and $msgs$ are shared

■ $i$ and $myid$ are *not* shared
Synchronizing Threads

- Shared variables are handy...

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

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>

/* 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;
}

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

badcnt.c
Carnegie Mellon

badcnt.c: Improper Synchronization

/* 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);
}

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

This guarantees the read/write actually happens

badcnt.c

This guarantees that the read/write actually happens
badcnt.c: Improper Synchronization

/* Global shared variable */
volatile long volatile long volatile long 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);
}

void *thread(void *vargp)
{
    long i, niters = *((long *)vargp);
    for (i = 0; i < niters; i++)
        cnt++;
    return NULL;
}
badcnt.c: Improper Synchronization

```c
/* 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);
}
```

race.c

```c
int main (int argc, char **argv[]) {
    pthread_t tid;
    for(int i=0; i<10; i++) {
        pthread_create(&tid, NULL, thread, &i);
    }
    pthread_exit(NULL);
    return 0;
}
```
/* 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);
}
badcnt.c: Improper Synchronization

```c
/* Global shared variable */
volatile long volatile long volatile long 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 < niter; i++)
        cnt++;

    return NULL;
}
```

badcnt.c
badcnt.c: Improper Synchronization

/* 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);
}

#include "badcnt.c"

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

/* 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;
}
badcnt.c: Improper Synchronization

/* 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;
}

linux> ./badcnt 1000000 OK cnt=2000000
linux> ./badcnt 1000000 BOOM! cnt=1332062

cnt should equal 2,000,000.
What went wrong?
Assembly Code for Counter Loop

C code for counter loop in thread i

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

Asm code for thread i

```assembly
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:                
```
Assembly Code for Counter Loop

C code for counter loop in thread i

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

*Asm code for thread i*

```assembly
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:
```
Assembly Code for Counter Loop

C code for counter loop in thread i

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

**Asm code for thread i**

```
.Hi: Head

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:
```
Assembly Code for Counter Loop

C code for counter loop in thread i

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

**Asm code for thread i**

```
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:
```

$H_i$: Head
$L_i$: Load cnt
$U_i$: Update cnt
$S_i$: Store cnt
Assembly Code for Counter Loop

C code for counter loop in thread i

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

Asm code for thread i

```
.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:
```

\[ H_i : \text{Head} \]

\[ L_i : \text{Load cnt} \]

\[ U_i : \text{Update cnt} \]

\[ S_i : \text{Store cnt} \]

\[ T_i : \text{Tail} \]
iClicker question

Suppose that cnt starts with value 0, and that two threads each execute the code below once. What are the possible values for cnt afterward?

A) Only 2

B) 1 or 2

C) 0 or 1 or 2

D) None of the above

```
movq   cnt(%rip),%rdx
addq   $1,  %rdx
movq   %rdx,  cnt(%rip)
```
Suppose that cnt starts with value 0, and that two threads each execute the code below once. What are the possible values for cnt afterward?

A) Only 2

B) 1 or 2

C) 0 or 1 or 2

D) None of the above

```assembly
movq cnt(%rip),%rdx
addq $1, %rdx
movq %rdx, cnt(%rip)
```
Concurrent Execution

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

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

- $I_i$ denotes that thread $i$ executes instruction $I$
- $%rdx_i$ 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></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## 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 $I$
- $\%rdx_i$ is the content of $\%rdx$ in thread $i$’s context

<table>
<thead>
<tr>
<th>$i$ (thread)</th>
<th>$\text{instr}_i$</th>
<th>$%rdx_1$</th>
<th>$%rdx_2$</th>
<th>$\text{cnt}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$H_1$</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
</tbody>
</table>
Concurrent Execution

- **Key idea:** In general, any sequentially consistent interleaving is possible, but some give an unexpected result!
  - $I_i$ denotes that thread $i$ executes instruction $I$
  - $%rdx_i$ 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>
</tbody>
</table>
Concurrent Execution

- **Key idea:** In general, any sequentially consistent interleaving is possible, but some give an unexpected result!
  - $I_i$ denotes that thread $i$ executes instruction $I$
  - $\%rdx_i$ 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>
</tbody>
</table>


Concurrent Execution

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

- $i_i$ denotes that thread $i$ executes instruction $I$
- $%rdx_i$ 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>
</tbody>
</table>
Concurrent Execution

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

- $I_i$ denotes that thread $i$ executes instruction $I$
- $%rdx_i$ 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&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>$H_1$</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>
</tbody>
</table>
Concurrent Execution

- **Key idea:** In general, any sequentially consistent interleaving is possible, but some give an unexpected result!
  - $I_i$ denotes that thread $i$ executes instruction $I$
  - $\%rdx_i$ is the content of $\%rdx$ in thread $i$’s context

<table>
<thead>
<tr>
<th>$i$ (thread)</th>
<th>$\text{instr}_i$</th>
<th>$%rdx_1$</th>
<th>$%rdx_2$</th>
<th>$\text{cnt}$</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>
</tbody>
</table>
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 \( l \)
- \( %r_{dx_i} \) is the content of \( %r_{dx} \) in thread \( i \)'s context

<table>
<thead>
<tr>
<th>( i ) (thread)</th>
<th>instr(_i)</th>
<th>( %r_{dx_1} )</th>
<th>( %r_{dx_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>
</tbody>
</table>
**Concurrent Execution**

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

- $i_i$ denotes that thread $i$ executes instruction $I$
- $\%rdx_i$ 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>
</tbody>
</table>
Concurrent Execution

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

- $I_i$ denotes that thread $i$ executes instruction $I$
- $%rdx_i$ 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>
</tbody>
</table>
Concurrent Execution

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

- $I_i$ denotes that thread $i$ executes instruction $I$
- $%rdx_i$ 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>
Concurrent Execution

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

- $I_i$ denotes that thread $i$ executes instruction $I$
- $\%rdx_i$ 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>

**OK**
Concurrent Execution

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

- $I_i$ denotes that thread $i$ executes instruction $I$
- $\%rdx_i$ 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
Concurrent Execution (cont)

- 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></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Concurrent Execution (cont)

- 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>
</tbody>
</table>
Concurrent Execution (cont)

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

<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>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>L&lt;sub&gt;1&lt;/sub&gt;</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
</tbody>
</table>
Concurrent Execution (cont)

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

<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>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>L&lt;sub&gt;1&lt;/sub&gt;</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>U&lt;sub&gt;1&lt;/sub&gt;</td>
<td>1</td>
<td>-</td>
<td>0</td>
</tr>
</tbody>
</table>
Concurrent Execution (cont)

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

<table>
<thead>
<tr>
<th>i (thread)</th>
<th>instrᵢ</th>
<th>%rdx₁</th>
<th>%rdx₂</th>
<th>cnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>H₁</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>L₁</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>U₁</td>
<td>1</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>H₂</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
</tbody>
</table>
Concurrent Execution (cont)

- 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>
</tbody>
</table>
Concurrent Execution (cont)

- 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>
</tbody>
</table>
Concurrent Execution (cont)

- 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>
</tbody>
</table>
Concurrent Execution (cont)

- 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>
</tbody>
</table>
Concurrent Execution (cont)

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

<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>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>L&lt;sub&gt;1&lt;/sub&gt;</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>U&lt;sub&gt;1&lt;/sub&gt;</td>
<td>1</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>H&lt;sub&gt;2&lt;/sub&gt;</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>L&lt;sub&gt;2&lt;/sub&gt;</td>
<td>-</td>
<td>0</td>
<td>0</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>1</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>U&lt;sub&gt;2&lt;/sub&gt;</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>S&lt;sub&gt;2&lt;/sub&gt;</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Concurrent Execution (cont)

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

<table>
<thead>
<tr>
<th>i (thread)</th>
<th>instr,</th>
<th>%rdx₁</th>
<th>%rdx₂</th>
<th>cnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>H₁</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>L₁</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>U₁</td>
<td>1</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>H₂</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>L₂</td>
<td>-</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>S₁</td>
<td>1</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>T₁</td>
<td>1</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>U₂</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>S₂</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>T₂</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Oops!
Concurrent Execution (cont)

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

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

Thread 1 critical section
Thread 2 critical section

Oops!
Concurrent Execution

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

- $I_i$ denotes that thread $i$ executes instruction $I$
- %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
Concurrent Execution (cont)

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

<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>H₁</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>L₁</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>U₁</td>
<td>1</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>H₂</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>L₂</td>
<td>-</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>S₁</td>
<td>1</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>T₁</td>
<td>1</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>U₂</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>S₂</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>T₂</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Oops!

- Thread 1 critical section
- Thread 2 critical section
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></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>L&lt;sub&gt;1&lt;/sub&gt;</td>
<td>0</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>1</td>
</tr>
<tr>
<td>2</td>
<td>S&lt;sub&gt;2&lt;/sub&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>U&lt;sub&gt;1&lt;/sub&gt;</td>
<td>1</td>
<td></td>
<td>1</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!
Concurrent Execution (cont)

- How about this ordering?

<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></td>
</tr>
<tr>
<td>2</td>
<td>H_2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>L_2</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>U_2</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>S_2</td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>U_1</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>S_1</td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>T_1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>T_2</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.
A progress graph depicts the discrete execution state space of concurrent threads.

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

A progress graph depicts the discrete execution state space of concurrent threads.

Each axis corresponds to the sequential order of instructions in a thread.
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 (Inst₁, Inst₂).
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 (Inst₁, Inst₂).

E.g., (L₁, S₂) denotes state where thread 1 has completed L₁ and thread 2 has completed S₂.
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.
Critical Sections and Unsafe Regions

L, U, and S form a critical section with respect to the shared variable \( \text{cnt} \).

Instructions in critical sections (wrt some shared variable) should not be interleaved.

Sets of states where such interleaving occurs form unsafe regions.
Critical Sections and 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.
Critical Sections and Unsafe Regions

Def: A trajectory is safe iff it does not enter any unsafe region

Claim: A trajectory is correct (wrt \(\text{cnt}\)) iff it is safe
Using the program graph, classify the following trajectories as either safe or unsafe.

1) \( H_1 L_1 U_1 S_1 H_2 L_2 U_2 S_2 T_2 T_1 \)
2) \( H_2 L_2 H_1 L_1 U_1 S_1 T_1 U_2 S_2 T_2 \)
3) \( H_1 H_2 L_2 U_2 S_2 L_1 U_1 S_1 T_1 T_2 \)

A. 1) Safe 2) Safe 3) Safe
B. 1) Unsafe 2) Unsafe 3) Unsafe
C. 1) Safe 2) Unsafe 3) Safe
D. 1) Safe 2) Unsafe 3) Unsafe
Using the program graph, classify the following trajectories as either safe or unsafe.

1) $H_1 L_1 U_1 S_1 H_2 L_2 U_2 S_2 T_2 T_1$
2) $H_2 L_2 H_1 L_1 U_1 S_1 T_1 U_2 S_2 T_2$
3) $H_1 H_2 L_2 U_2 S_2 L_1 U_1 S_1 T_1 T_2$

A. 1) Safe 2) Safe 3) Safe
B. 1) Unsafe 2) Unsafe 3) Unsafe
C. 1) Safe 2) Unsafe 3) Safe
D. 1) Safe 2) Unsafe 3) Unsafe
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.
badcnt.c: Improper Synchronization

```c
/* 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;
}
```

linux> ./badcnt 1000000 OK cnt=2000000
linux> ./badcnt 1000000 BOOM! cnt=1332062

cnt should equal 2,000,000.
What went wrong?
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.
Semaphores

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

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

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

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

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

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

- **Semaphore**: non-negative global integer synchronization variable.
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 $s$. 
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**: $(s \geq 0)$
C Semaphore Operations

Pthreads functions:

```
#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) */
```
badcnt.c: Improper Synchronization

/* 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 \textit{mutex}, initially 1, with each shared variable (or related set of shared variables).
  - Surround corresponding critical sections with \textit{P(mutex)} and \textit{V(mutex)} operations.
Using Semaphores for Mutual Exclusion

■ Basic idea:
  ▪ Associate a unique semaphore \textit{mutex}, initially 1, with each shared variable (or related set of shared variables).
  ▪ Surround corresponding critical sections with \textit{P(mutex)} and \textit{V(mutex)} operations.

■ Terminology:
  ▪ \textit{Binary semaphore}: semaphore whose value is always 0 or 1
  ▪ \textit{Mutex}: binary semaphore used for mutual exclusion
    ▪ \textit{P} operation: “locking” the mutex
    ▪ \textit{V} operation: “unlocking” or “releasing” the mutex
    ▪ “\textit{Holding}” a mutex: locked and not yet unlocked.
  ▪ \textit{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++) {
    sem_wait(&mutex);
    cnt++;
    sem_post(&mutex);
}
```

```
linux> ./goodcnt 1000000 OK cnt=1000000
```

```
linux> ./goodcnt 1000000 OK cnt=1000000
```
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++) {
    sem_wait(&mutex);
    cnt++;
    sem_post(&mutex);
}
```

Warning: It’s orders of magnitude slower than `badcnt.c`.
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)
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)

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)

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 $s = 1$
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.