CSC 252: Computer Organization
Spring 2018: Lecture 25

Instructor: Yuhao Zhu

Department of Computer Science
University of Rochester

Action Items:
• Programming Assignment 5 grades are out
• Programming Assignment 6 is due soon
Announcement

• Programming assignment 6 is due on 11:59pm, Monday, April 30.
• Programming assignment 5 grades are out
Today

• From process to threads
  • Basic thread execution model
• Shared variables in multi-threaded programming
  • Mutual exclusion using semaphore
  • Deadlock
• Thread-level parallelism
  • Amdahl’s Law: performance model of parallel programs
• Hardware implementation implications of threads
  • Multi-core
  • Hyper-threading
  • Cache coherence
Traditional View of a Process

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

**Process context**

<table>
<thead>
<tr>
<th>Program context:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data registers</td>
</tr>
<tr>
<td>Condition codes</td>
</tr>
<tr>
<td>Stack pointer (SP)</td>
</tr>
<tr>
<td>Program counter (PC)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Kernel context:</th>
</tr>
</thead>
<tbody>
<tr>
<td>VM structures</td>
</tr>
<tr>
<td>Descriptor table</td>
</tr>
<tr>
<td>brk pointer</td>
</tr>
</tbody>
</table>

**Code, data, and stack**

- Stack
  - Shared libraries
  - Run-time heap
  - Read/write data
  - Read-only code/data
Alternate View of a Process

- Process = thread + code, data, and kernel context
- A thread runs in the context of a process
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>Stack 1</th>
<th>Stack 2</th>
</tr>
</thead>
</table>

Thread 1 context:
- Data registers
- Condition codes
- SP1
- PC1

Thread 2 context:
- Data registers
- Condition codes
- SP2
- PC2

Shared code and data
- Shared libraries
- Run-time heap
- Read/write data
- Read-only code/data
- Kernel context:
  - VM structures
  - Descriptor table
  - Brk pointer
Logical View of Threads

• Threads associated with process form a pool of peers
  • Unlike processes which form a tree hierarchy

Threads associated with process foo

Process hierarchy
Concurrent Threads

- Two threads are *concurrent* if their flows overlap in time.
- Otherwise, they are sequential.

**Examples:**
- Concurrent: A & B, A&C
- Sequential: B & C
Concurrent Thread Execution

- Single Core Processor
  - Simulate parallelism by time slicing

- Multi Core Processor
  - Threads can have true parallelisms

Run 3 threads on 2 cores
Threads vs. Processes

• How threads and processes are similar
  • Each has its own logical control flow
  • Each can run concurrently with others (possibly on different cores)
  • Each is context switched, controlled by kernel
Threads vs. Processes

- How threads and processes are similar
  - Each has its own logical control flow
  - Each can run concurrently with others (possibly on different cores)
  - Each is context switched, controlled by kernel

- How threads and processes are different
  - Threads share all code and data (except local stacks)
    - Processes (typically) do not
  - Threads are less expensive than processes
    - Process control (creating and reaping) twice as expensive as thread control
    - Typical Linux numbers:
      - ~20K cycles to create and reap a process
      - ~10K cycles (or less) to create and reap a thread
Posix Threads (Pthreads) Interface

- **Pthreads**: Standard interface for ~60 functions that manipulate threads from C programs
  - Creating and reaping threads
    - `pthread_create()`
    - `pthread_join()`
  - Determining your thread ID
    - `pthread_self()`
  - Terminating threads
    - `pthread_cancel()`
    - `pthread_exit()`
    - `exit()` [terminates all threads], `RET` [terminates current thread]
  - Synchronizing access to shared variables
    - `pthread_mutex_init`
    - `pthread_mutex_[un]lock`
The Pthreads "hello, world" Program

/*
 * hello.c - Pthreads "hello, world" program
 */
#include "csapp.h"

void *thread(void *vargp);

int main()
{
    pthread_t tid;
    Pthread_create(&tid, NULL, thread, NULL);
    Pthread_join(tid, NULL);
    exit(0);
} /* hello.c */

void *thread(void *vargp) /* thread routine */
{
    printf("Hello, world!\n");
    return NULL;
} /* hello.c */
The Pthreads "hello, world" Program

/*
 * hello.c - Pthreads "hello, world" program
 */
#include "csapp.h"

void *thread(void *vargp);

int main()
{
    pthread_t tid;
    Pthread_create(&tid, NULL, thread, NULL);
    Pthread_join(tid, NULL);
    exit(0);
}

void *thread(void *vargp) /* thread routine */
{
    printf("Hello, world!\n");
    return NULL;
}
/*
 * hello.c - Pthreads "hello, world" program
 */
#include "csapp.h"
void *thread(void *vargp);

int main()
{
    pthread_t tid;
Pthread_create(&tid, NULL, thread, NULL);
Pthread_join(tid, NULL);
exit(0);
}

void *thread(void *vargp) /* thread routine */
{
    printf("Hello, world!\n");
    return NULL;
}
The Pthreads "hello, world" Program

/*
 * hello.c - Pthreads "hello, world" program
 */
#include "csapp.h"
void *thread(void *vargp);

int main()
{
    pthread_t tid;
Pthread_create(&tid, NULL, thread, NULL);
Pthread_join(tid, NULL);
    exit(0);
}
The Pthreads "hello, world" Program

/*
 * hello.c - Pthreads "hello, world" program
 */
#include "csapp.h"
void *thread(void *vargp);

int main()
{
    pthread_t tid;
    Pthread_create(&tid, NULL, thread, NULL);
    Pthread_join(tid, NULL);
    exit(0);
}

void *thread(void *vargp) /* thread routine */
{
    printf("Hello, world!\n");
    return NULL;
}
The Pthreads "hello, world" Program

/*
 * hello.c - Pthreads "hello, world" program
 */
#include "csapp.h"
void *thread(void *vargp);

int main()
{
    pthread_t tid;
    Pthread_create(&tid, NULL, thread, NULL);
    Pthread_join(tid, NULL);
    exit(0);
}

void *thread(void *vargp) /* thread routine */
{
    printf("Hello, world!\n");
    return NULL;
}
Execution of Threaded “hello, world”

Main thread
Execution of Threaded “hello, world”

Main thread

call Pthread_create()
Execution of Threaded “hello, world”

call Pthread_create()
Execution of Threaded “hello, world”

call Pthread_create()
Execution of Threaded “hello, world”

Main thread

call Pthread_create()

Peer thread

Pthread_create() returns
Execution of Threaded “hello, world”

call Pthread_create()
Pthread_create() returns

call Pthread_join()
Execution of Threaded “hello, world”

- Main thread
  - call Pthread_create()
  - Pthread_create() returns
  - call Pthread_join()
  - Main thread waits for peer thread to terminate

- Peer thread
Execution of Threaded “hello, world”

- Main thread
  - call Pthread_create()
  - Pthread_create() returns
  - call Pthread_join()
  - Main thread waits for peer thread to terminate

- Peer thread
  - printf()
  - return NULL;
  - Peer thread terminates
Execution of Threaded "hello, world"

Main thread

- call Pthread_create()
  - Pthread_create() returns

Peer thread

- printf()
- return NULL;

Main thread waits for peer thread to terminate

- call Pthread_join()
- Pthread_join() returns

Peer thread terminates
Execution of Threaded "hello, world"

Main thread

- call Pthread_create()
- Pthread_create() returns
- call Pthread_join()
- Main thread waits for peer thread to terminate
- Pthread_join() returns
- exit()
  - Terminates main thread and any peer threads

Peer thread

- printf()
- return NULL;
- Peer thread terminates
Today

• From process to threads
  • Basic thread execution model

• Shared variables in multi-threaded programming
  • Mutual exclusion using semaphore
  • Deadlock

• Thread-level parallelism
  • Amdahl’s Law: performance model of parallel programs

• Hardware implementation implications of threads
  • Multi-core
  • Hyper-threading
  • Cache coherence
One great thing about threads is that they can share the same program variables.

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

Intuitively, the answer is as simple as “global variables are shared” and “stack variables are private”. Not so simple in reality.
Example Program to Illustrate Sharing

char **ptr; /* global var */

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

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);
}
Example Program to Illustrate Sharing

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

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

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);
}
```

Diagram:
- Main thread stack
- Peer thread 0 stack
- Peer thread 1 stack
- Memory mapped region for shared libraries
- Runtime heap (malloc)
- Uninitialized data (.bss)
- Initialized data (.data)
- Program text (.text)
Example Program to Illustrate Sharing

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

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

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);
}
```

```
sharing.c
```

Diagram:
- Main thread stack
  - `i`
  - `tid`
- Peer thread 0 stack
- Peer thread 1 stack
- Memory mapped region for shared libraries
- Runtime heap (malloc)
- Uninitialized data (.bss)
  - `ptr`
- Initialized data (.data)
- Program text (.text)
Example Program to Illustrate Sharing

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

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

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);
}
```

```
 sharing.c

```

Memory mapped region for shared libraries

Main thread stack
- i
- tid
- msgs

Peer thread 0 stack
- myid

Peer thread 1 stack

Memory mapped region

Runtime heap (malloc)

Uninitialized data (.bss)
- ptr

Initialized data (.data)

Program text (.text)

Example Program to Illustrate Sharing

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

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

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);
}
```

Diagram:
- Main thread stack
  - i
tid
  - msgs
- Peer thread 0 stack
  - myid
- Peer thread 1 stack
  - myid
- Memory mapped region for shared libraries
- Runtime heap (malloc)
- Uninitialized data (.bss)
  - ptr
- Initialized data (.data)
- Program text (.text)
char **ptr; /* global var */

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

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);
}
Example Program to Illustrate Sharing

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

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

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);
}
```

---

**Memory Layout:**

<table>
<thead>
<tr>
<th>Main thread stack</th>
<th>Peer thread 0 stack</th>
<th>Peer thread 1 stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>i, tid</td>
<td>myid</td>
<td></td>
</tr>
<tr>
<td>msgs</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Main thread stack:**
  - `i`
  - `tid`
- **Peer thread 0 stack:**
  - `myid`
- **Peer thread 1 stack:**
  - `myid`

<table>
<thead>
<tr>
<th>Memory mapped region for shared libraries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime heap (malloc)</td>
</tr>
<tr>
<td>Uninitialized data (.bss)</td>
</tr>
<tr>
<td>Initialized data (.data)</td>
</tr>
<tr>
<td>Program text (.text)</td>
</tr>
</tbody>
</table>
Example Program to Illustrate Sharing

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

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

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);
}
```

Diagram:

- **Main thread stack**
  - `i`
  - `tid`
  - `msgs`

- **Peer thread 0 stack**
  - `myid`

- **Peer thread 1 stack**
  - `myid`

- **Memory mapped region** for shared libraries

- **Runtime heap** (malloc)
  - **Uninitialized data (.bss)**
  - `ptr`
  - **Initialized data (.data)**
  - `cnt`

- **Program text (.text)**
Example Program to Illustrate Sharing

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

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

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);
}
```

<table>
<thead>
<tr>
<th>Main thread stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>i     tid</td>
</tr>
<tr>
<td>msgs</td>
</tr>
<tr>
<td>Peer thread 0 stack</td>
</tr>
<tr>
<td>myid</td>
</tr>
<tr>
<td>Peer thread 1 stack</td>
</tr>
<tr>
<td>myid</td>
</tr>
<tr>
<td>Memory mapped region for shared libraries</td>
</tr>
<tr>
<td>Runtime heap (malloc)</td>
</tr>
<tr>
<td>Uninitialized data (.bss)</td>
</tr>
<tr>
<td>ptr</td>
</tr>
<tr>
<td>Initialized data (.data)</td>
</tr>
<tr>
<td>cnt</td>
</tr>
<tr>
<td>Program text (.text)</td>
</tr>
</tbody>
</table>

sharig.c
The example program to illustrate sharing is shown below:

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

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

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);
}
```

The diagram illustrates the memory layout and stack usage for the main thread and two peer threads. The main thread creates two peer threads, each of which writes a message to a global string array referenced by `ptr`. The memory map shows the allocation of different sections of the program, including:

- **Main thread stack**: `i tid`
- **Peer thread 0 stack**: `mysd myid`
- **Peer thread 1 stack**: `mysd myid`
- **Memory mapped region for shared libraries**
- **Runtime heap (malloc)**
- **Uninitialized data (.bss)** `ptr`
- **Initialized data (.data)** `cnt`
- **Program text (.text)**
Example Program to Illustrate Sharing

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

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

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);
}
```

Diagram:
- Main thread stack
  - `i` tid
  - `msgs`
- Peer thread 0 stack
  - `myid`
- Peer thread 1 stack
  - `myid`
- Memory mapped region for shared libraries
- Runtime heap (malloc)
- Uninitialized data (.bss)
  - `ptr`
- Initialized data (.data)
  - `cnt`
- Program text (.text)

```
sharing.c
```
Example Program to Illustrate Sharing

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

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

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);
}
```

- **Main thread stack**
  - i
dt
  - msgs

- **Peer thread 0 stack**
  - myid

- **Peer thread 1 stack**
  - myid

- Memory mapped region for shared libraries

- Runtime heap (malloc)

- Uninitialized data (.bss)
  - ptr

- Initialized data (.data)
  - cnt

- Program text (.text)
Synchronizing Threads

• Shared variables are handy...

• …but introduce the possibility of nasty \textit{synchronization} errors.
Improper Synchronization

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

int main(int argc, char **argv)
{
    pthread_t tid1, tid2;
    long niters = 10000;

    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 * 10000))
        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)
{
    pthread_t tid1, tid2;
    long niters = 10000;

    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 * 10000))
        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;
}

linux> ./badcnt
OK cnt=20000

linux> ./badcnt
BOOM! cnt=13051

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++;
```

**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**
- **$T_i$: Tail**
**Concurrent Execution**

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

<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>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>1</td>
<td>S&lt;sub&gt;1&lt;/sub&gt;</td>
<td>1</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>L&lt;sub&gt;2&lt;/sub&gt;</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>U&lt;sub&gt;2&lt;/sub&gt;</td>
<td>-</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>S&lt;sub&gt;2&lt;/sub&gt;</td>
<td>-</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

- Thread 1 critical section
- Thread 2 critical section
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 \(S_2\).
A trajectory is a sequence of legal state transitions that describes one possible concurrent execution of the threads.

Execution Ordering:

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

Execution Ordering:

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

Execution Ordering:

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

Execution Ordering:

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

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

Execution Ordering:

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

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

Execution Ordering:

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

Execution Ordering:

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

Execution Ordering:

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

Execution Ordering:

H1, L1, U1, S1, T1, H2, L2, U2, S2, T2
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>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>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>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>
</tbody>
</table>
Trajectories in Progress Graphs

Thread 1

H₁ L₁ U₁ S₁ T₁

Thread 2

H₂ L₂ U₂ S₂ T₂

Execution Ordering:
H₁, L₁, U₁, H₂, L₂, S₁, U₂, S₂, T₁, T₂
Trajectories in Progress Graphs

Execution Ordering:
H1, L1, U1, H2, L2, S1, U2, S2, T1, T2
Trajectories in Progress Graphs

Execution Ordering:
H1, L1, U1, H2, L2, S1, U2, S2, T1, T2
Trajectories in Progress Graphs

Execution Ordering:
H1, L1, U1, H2, L2, S1, U2, S2, T1, T2
Trajectories in Progress Graphs

Execution Ordering:
H1, L1, U1, H2, L2, S1, U2, S2, T1, T2
Trajectories in Progress Graphs

Execution Ordering:
H1, L1, U1, H2, L2, S1, U2, S2, T1, T2
Trajectories in Progress Graphs

Execution Ordering:
H1, L1, U1, H2, L2, S1, U2, S2, T1, T2
Trajectories in Progress Graphs

Execution Ordering:
H1, L1, U1, H2, L2, S1, U2, S2, T1, T2
Trajectories in Progress Graphs

Execution Ordering:
H1, L1, U1, H2, L2, S1, U2, S2, T1, T2
Trajectories in Progress Graphs

Execution Ordering:
H1, L1, U1, H2, L2, S1, U2, S2, T1, T2
Trajectories in Progress Graphs

Execution Ordering:

H1, L1, U1, H2, L2, S1, U2, S2, T1, T2
Concurrent Execution (cont)

<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>L$_1$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>L$_2$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>U$_2$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>S$_2$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>U$_1$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>S$_1$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Concurrent Execution (cont)

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

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

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

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

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

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

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

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

• Another undesired, but legal, interleaving

<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>L_1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>L_2</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>U_2</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>S_2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>U_1</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>S_1</td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>
Trajectories in Progress Graphs

Execution Ordering:
H1, L1, H2, L2, U2, S2, U1, S1, T1, T2
Trajectories in Progress Graphs

Execution Ordering:
H1, L1, H2, L2, U2, S2, U1, S1, T1, T2
Trajectories in Progress Graphs

Execution Ordering:
H1, L1, H2, L2, U2, S2, U1, S1, T1, T2
Trajectories in Progress Graphs

Execution Ordering:
H1, L1, H2, L2, U2, S2, U1, S1, T1, T2
Trajectories in Progress Graphs

Execution Ordering:
H1, L1, H2, L2, U2, S2, U1, S1, T1, T2
Trajectories in Progress Graphs

Execution Ordering:

H1, L1, H2, L2, U2, S2, U1, S1, T1, T2
Trajectories in Progress Graphs

Execution Ordering:
H1, L1, H2, L2, U2, S2, U1, S1, T1, T2
Trajectories in Progress Graphs

Execution Ordering:
H1, L1, H2, L2, U2, S2, U1, S1, T1, T2
Trajectories in Progress Graphs

Execution Ordering:
H1, L1, H2, L2, U2, S2, U1, S1, T1, T2
Trajectories in Progress Graphs

Execution Ordering:
H1, L1, H2, L2, U2, S2, U1, S1, T1, T2
Trajectories in Progress Graphs

Execution Ordering:
H1, L1, H2, L2, U2, S2, U1, S1, T1, T2
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**
- **$T_i$: Tail**
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
movq (%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:
```

{critical section wrt cnt

\{H_i: Head \}

\{L_i: Load cnt \}

\{U_i: Update cnt \}

\{S_i: Store cnt \}

\{T_i: Tail \}
Critical Section

• Code section (a sequence of instructions) where no more than one thread should be executing concurrently.
• Critical section refers to code, but its intention is to protect data!
• Threads need to have mutually exclusive access to critical section

Asm code for thread i

```asm
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
- **T_i**: Tail

critical section wrt cnt
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
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*.
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.

Unsafe region

Critical sections and unsafe regions diagram
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.
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
  - Monitors (Java)
Using Semaphores for Mutual Exclusion

• Basic idea:
  • Associate each shared variable (or related set of shared variables) with a unique variable, called semaphore, initially 1
  • Every time a thread tries to enter the critical section, it first checks the mutex value. If it’s still 1, the thread decrements the mutex value to 0 (through a P operation) and enters the critical section
  • Every time a thread exits the critical section, it increments the mutex value to 1 (through a V operation) so that other threads are now allowed to enter the critical section
  • No more than one thread can be in the critical section at a time
Using Semaphores for Mutual Exclusion

• 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.
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`.
Semaphores
Semaphores

- **Semaphore**: non-negative global integer synchronization variable. Manipulated by P and V operations.
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 to the caller.
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 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 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 >= 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) */
```
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 $s = 1$
Mutual Exclusion 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.
Coordinate Access to Shared Resources

- So far we have described how to use binary semaphore to ensure mutual exclusion.
- We can also use counting (non-binary) semaphore to achieve more sophisticated coordination of shared data accesses.
- Basic idea: Use counting semaphores to keep track of resource state and to notify other threads that some condition has become true.
- Two classic examples:
  - The Producer-Consumer Problem
  - The Readers-Writers Problem (see textbook)
Producer-Consumer Problem

- Common synchronization pattern:
  - Producer waits for empty *slot*, inserts item in buffer, and notifies consumer
  - Consumer waits for *item*, removes it from buffer, and notifies producer

- Examples
  - Multimedia processing:
    - Producer creates MPEG video frames, consumer renders them
  - Graphical user interfaces (GUI) applications:
    - Producer detects mouse clicks, mouse movements, and keyboard hits and inserts corresponding events in buffer
    - Consumer retrieves events from buffer and paints the display
Producer-Consumer on an $n$-element Buffer

- Consume element by moving front pointer forward. Produce element by moving rear pointer forward
Producer-Consumer on an $n$-element Buffer

- Consume element by moving front pointer forward. Produce element by moving rear pointer forward.
- Requires a mutex (binary semaphore) to ensure mutual exclusive access to the shared buffer.
Producer-Consumer on an $n$-element Buffer

- Consume element by moving front pointer forward. Produce element by moving rear pointer forward.
- Requires a mutex (binary semaphore) to ensure mutual exclusive access to the shared buffer.
- But how do we know if there are enough spaces for the producer and if there are available elements for the consumer?
Producer-Consumer on an $n$-element Buffer

- Consume element by moving front pointer forward. Produce element by moving rear pointer forward
- Requires a mutex (binary semaphore) to ensure mutual exclusive access to the shared buffer
- But how we know if there are enough space for the producer and there are available elements for the consumer?
- Use two counting semaphores:
  - slots: counts the available slots in the buffer
  - items: counts the available items in the buffer
Implementation

Removing an item from a shared buffer:

```c
/* Remove and return the first item from buffer sp */
int sbuf_remove(sbuf_t *sp)
{
    int item;

    P(&sp->mutex);                        /* Lock the buffer */
    item = sp->buf[(++sp->front)%(sp->n)]; /* Remove the item */
    V(&sp->mutex);                        /* Unlock the buffer */

    return item;
}
```

Inserting an item into a shared buffer:

```c
/* Insert item onto the rear of shared buffer sp */
void sbuf_insert(sbuf_t *sp, int item)
{
    P(&sp->mutex);                        /* Lock the buffer */
    sp->buf[(++sp->rear)%(sp->n)] = item; /* Insert the item */
    V(&sp->mutex);                        /* Unlock the buffer */
}
```
/* Remove and return the first item from buffer sp */
int sbuf_remove(sbuf_t *sp)
{
    int item;
    P(&sp->items);                          /* Wait for available item */
    P(&sp->mutex);                           /* Lock the buffer */
    item = sp->buf[++sp->front]%(sp->n)];   /* Remove the item */
    V(&sp->mutex);                           /* Unlock the buffer */
    V(&sp->slots);                           /* Announce available slot */
    return item;
}

/* Insert item onto the rear of shared buffer sp */
void sbuf_insert(sbuf_t *sp, int item)
{
    P(&sp->mutex);                           /* Lock the buffer */
    sp->buf[++sp->rear]%(sp->n)] = item;     /* Insert the item */
    V(&sp->mutex);                           /* Unlock the buffer */
    V(&sp->items);                           /* Announce available item */
}
Implementation

Removing an item from a shared buffer:

```c
/* Remove and return the first item from buffer sp */
int sbuf_remove(sbuf_t *sp)
{
    int item;
    P(&sp->items);    /* Wait for available item */
    P(&sp->mutex);    /* Lock the buffer */
    item = sp->buf[(++sp->front)%(sp->n)];  /* Remove the item */
    V(&sp->mutex);    /* Unlock the buffer */
    V(&sp->slots);    /* Announce available slot */
    return item;
}
```

Inserting an item into a shared buffer:

```c
/* Insert item onto the rear of shared buffer sp */
void sbuf_insert(sbuf_t *sp, int item)
{
    P(&sp->slots);    /* Wait for available slot */
    P(&sp->mutex);    /* Lock the buffer */
    sp->buf[((++sp->rear)%(sp->n)] = item;  /* Insert the item */
    V(&sp->mutex);    /* Unlock the buffer */
    V(&sp->items);    /* Announce available item */
}
```