Thread Synchronization

CS 105: Computer Systems
Lecture 11

Melissa O'Neill

February 25, 2026

Learning Goals

- **Continue to discuss concurrency issues**
  - Critical sections and "unsafe" execution states
  - Race conditions
- **Discuss thread synchronization constructs**
- **Reason about *mutual exclusion* and *waiting for certain conditions* to be true**
  - Can use **semaphores** to accomplish both goals
- **Quiz 3**
  - Processes: understand what happens with fork()
  - Concurrency: how threads share variables in memory; the semaphore invariant

Recall: Assembly Code for `badcnt.c` Loop

C code for counter loop in thread i

```
for (j = 0; j < iters; ++j)
    ++cnt;
```

Assembly 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:
```

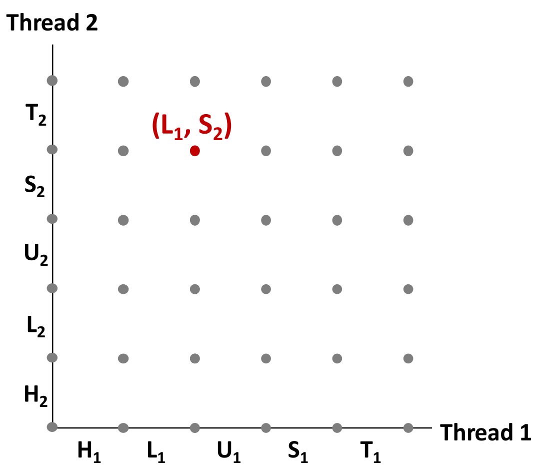
H_i : Head

L_i : Load `cnt` into %rdx
U_i : Update `cnt` in %rdx
S_i : Store `cnt` back to memory

T_i : Tail

`++cnt` is three lines of assembly!

Progress Graphs



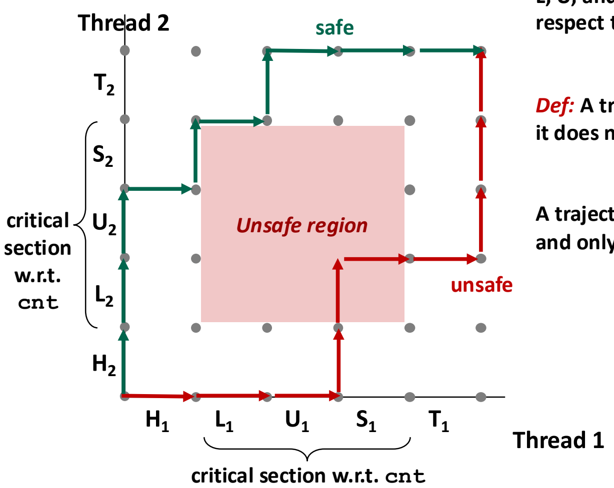
A ***progress graph*** depicts the discrete ***execution state space*** of concurrent threads.

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

Each point corresponds to a possible ***execution state*** ($Inst_1$, $Inst_2$).

E.g., (L_1, S_2) denotes state where thread 1 has completed L_1 and thread 2 has completed S_2.

Critical Sections and Unsafe Regions



L, U, and S form a ***critical section*** with respect to the shared variable `cnt`

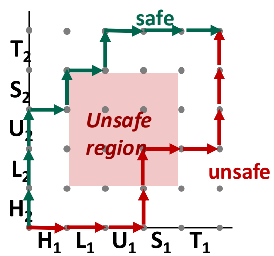
Def: A trajectory is ***safe*** if and only if it does not enter any unsafe region

A trajectory is correct (w.r.t. `cnt`) if and only if it is safe

Race Conditions

- A ***race*** occurs when correctness of the program depends on one thread reaching point *x* before another thread reaches point *y*

Enforcing Mutual Exclusion



- **How can we guarantee a safe trajectory?**
  - We must ***synchronize*** the execution of the threads so that they can never have an unsafe trajectory.
  - I.e., we need to guarantee ***mutually exclusive access*** for each critical section.
- **Classic approach**
  - Semaphores, developed by Edsger Dijkstra
- **Other approaches**
  - Mutex and condition variables (in Pthreads library) ← Ring buffer lab!
  - Monitors (Java)

Semaphores: operations

- ***Semaphore:*** **non-negative global integer synchronization variable. Manipulated by *P* and *V* operations.**

- **P(s) {** *// aka " test and decrement" aka "down"*

```
  if (s > 0){      } Test and decrement occur atomically (indivisibly)
    --s;
    return;
  } else { // s == 0
    suspend thread until notified that s > 0
    --s;
    return;
  }
}
```

- ***V(s)* {** *// aka "up"*

```
++s; // atomic
if any thread suspended on s, send wakeup to exactly one
return;
```

Semaphore invariant: *(s >= 0)*

Using Semaphores for Mutual Exclusion

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

- **Terminology:**
  - ***Binary semaphore***: semaphore whose value is always 0 or 1
  - ***Mutex:*** binary semaphore used for mutual exclusion
    - P operation: "locking" the mutex
    - V operation: "unlocking" or "releasing" the mutex
    - *"Holding"* a mutex: locked and not yet unlocked

Semaphore Operations in C

Library 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) */
```

Textbook's wrapper functions:

```
#include "csapp.h"

void P(sem_t *s); /* Wrapper function for sem_wait */
void V(sem_t *s); /* Wrapper function for sem_post */
```


`goodcnt.c`: Proper Synchronization

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

```
long cnt = 0;                /* Counter */
sem_t mutex;          /* Semaphore to protect cnt */

Sem_init(&mutex, 0, 1); /* Initialize mutex = 1 */
```

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

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

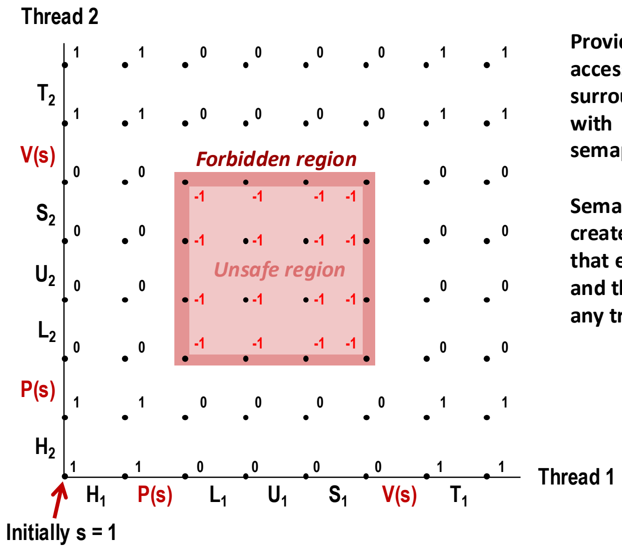
```
linux> ./goodcnt 10000
OK cnt=20000
linux> ./goodcnt 10000
OK cnt=20000
linux>
```


`goodcnt.c`: Example interleaving

Time ↓

Thread 1	Thread 2
`i = 0; i < iters`	
	`i = 0; i < iters`
`P(&mutex) // 1 → 0`	
	`P(&mutex) // = 0, blocked`
`cnt++`	
`V(&mutex) // 0 → 1`	
	`1 → 0`
	`cnt++`
	`V(&mutex) // 0 → 1`
	`i++; i < iters`
`i++; i < iters`	
	`P(&mutex) // 1 → 0`
. . .	.

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.

Exercise: check-in

- **Why must the test/decrement (P) and increment (V) operations be implemented so they happen indivisibly?**

- **What would be the impact of using P and V around the whole for-loop on the previous slide, e.g.:**

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


Another worry: Deadlock

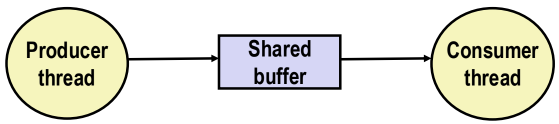
- **Definition: A process is *deadlocked* iff it is waiting for a condition that will never be true**

- **Typical Scenario**
  - Both threads 1 and 2 need two resources (A and B) to proceed
  - Thread 1 acquires A, waits for B
  - Thread 2 acquires B, waits for A
  - Both will wait forever!

***Unfortunate fact*: deadlock is often nondeterministic (race)**

- **Various solutions, e.g., deadlock *prevention* involves threads acquiring locks in the same order.**

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

Exercise: one producer and one consumer

(attempt)

- Desired outcome: producer gives consumer each of the values 0,1,2,3,4
- Using a binary semaphore; the “buffer” is a single `int`
  - Assume we have these global variables and their initial values:

```
int buf = -1;
sem_t mutex; /* initially 1*/
```

- What is the problem with this implementation?

```
/* producer thread */
void *producer(void *arg) {
  int i, data;

  for (i=0; i < 5; ++i) {
    data = i; /* produce item */

    /* write data to buf */
    P(&mutex);
    buf = data;
    V(&mutex);
  }
  return NULL;
}
```

```
/* consumer thread */
void *consumer(void *arg) {
  int i, item;

  for (i=0; i < 5; ++i) {

    /* read item from buf */
    P(&mutex);
    item = buf;
    V(&mutex);
  }
  return NULL;
}
```


Exercise: one producer and one consumer

(another attempt)

- Assume these global variables and initial values:

```
int buf = -1;
int buf_full = 0;
sem_t mutex /* initially 1*/
```

- Do you see any issues with this code? *Hint*: consider performance

```
/* producer thread */
void *producer(void *arg) {
  int i, data;

  for (i=0; i < 5; i++) {
    data = i; /* produce item */
    /* wait for space */
    while (buf_full) {}

    /* write item to buf */
    P(&mutex);
    buf = data;
    buf_full = 1;
    V(&mutex);
  }
 return NULL;
```

```
/* consumer thread */
void *consumer(void *arg) {
  int i, item;

  for (i=0; i < 5; i++) {
    /* wait for item */
    while (!buf_full) {}

    /* read item from buf */
    P(&mutex);
    item = buf;
    buf_full = 0;
    V(&mutex);
  }
  return NULL;
}
```


Using Semaphores to Coordinate Access to Shared Resources

- **Basic idea: Thread uses a semaphore operation to notify another thread that some condition has become true**
  - Use **counting semaphores** to keep track of resource state and to notify other threads
  - May also need a **mutex** to protect access to resource(s)
- **Classic synchronization problems:**
  - The Producer-Consumer Problem
  - The Readers-Writers Problem
  - The Dining-Philosophers Problem

Producer-Consumer on Buffer That Holds One Item (correct solution)

```
/* buf1.c - producer-consumer
on 1-element buffer */
#include "csapp.h"

#define ITERS 5

void *producer(void *arg);
void *consumer(void *arg);

struct {
  int buf; /* shared var */
  sem_t full; /* sems */
  sem_t empty;
} shared;
```

```
int main() {
  pthread_t tid_producer;
  pthread_t tid_consumer;

  /* initialize the semaphores */
  Sem_init(&shared.empty, 0, 1);
  Sem_init(&shared.full,  0, 0);

  /* create threads and wait */
  Pthread_create(&tid_producer, NULL,
                 producer, NULL);
  Pthread_create(&tid_consumer, NULL,
                 consumer, NULL);

  Pthread_join(tid_producer, NULL);
  Pthread_join(tid_consumer, NULL);

  exit(0);
}
```


Producer-Consumer with One Item (cont)

Initially: empty = 1, full = 0.

```
/* producer thread */
void *producer(void *arg) {
  int i, data;

  for (i=0; i< ITERS; i++) {
    /* produce item */
    data = i;
    printf("produced %d\n", item);

    /* write item to buf */
    P(&shared.empty);
    shared.buf = data;
    V(&shared.full);
  }
  return NULL;
}
```

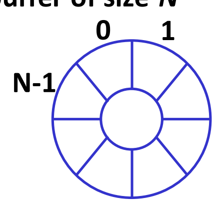
```
/* consumer thread */
void *consumer(void *arg) {
  int i, item;

  for (i=0; i< ITERS; i++) {
    /* read item from buf */
    P(&shared.full);
    item = shared.buf;
    V(&shared.empty);

    /* consume item */
    printf("consumed %d\n", item);
  }
  return NULL;
}
```


Exercise

- **Consider extending the solution to the producer/consumer problem on the previous slide to allow for a buffer of size *N***
  - This "**ring buffer**" used circularly such that after using spot *N*-1, the next spot to use is spot 0
  - To circle around to access item `i`, access buffer at index `i % N`



- **Brainstorm with people around you: What are some changes to the previous solution you would need to consider to make this version work?**