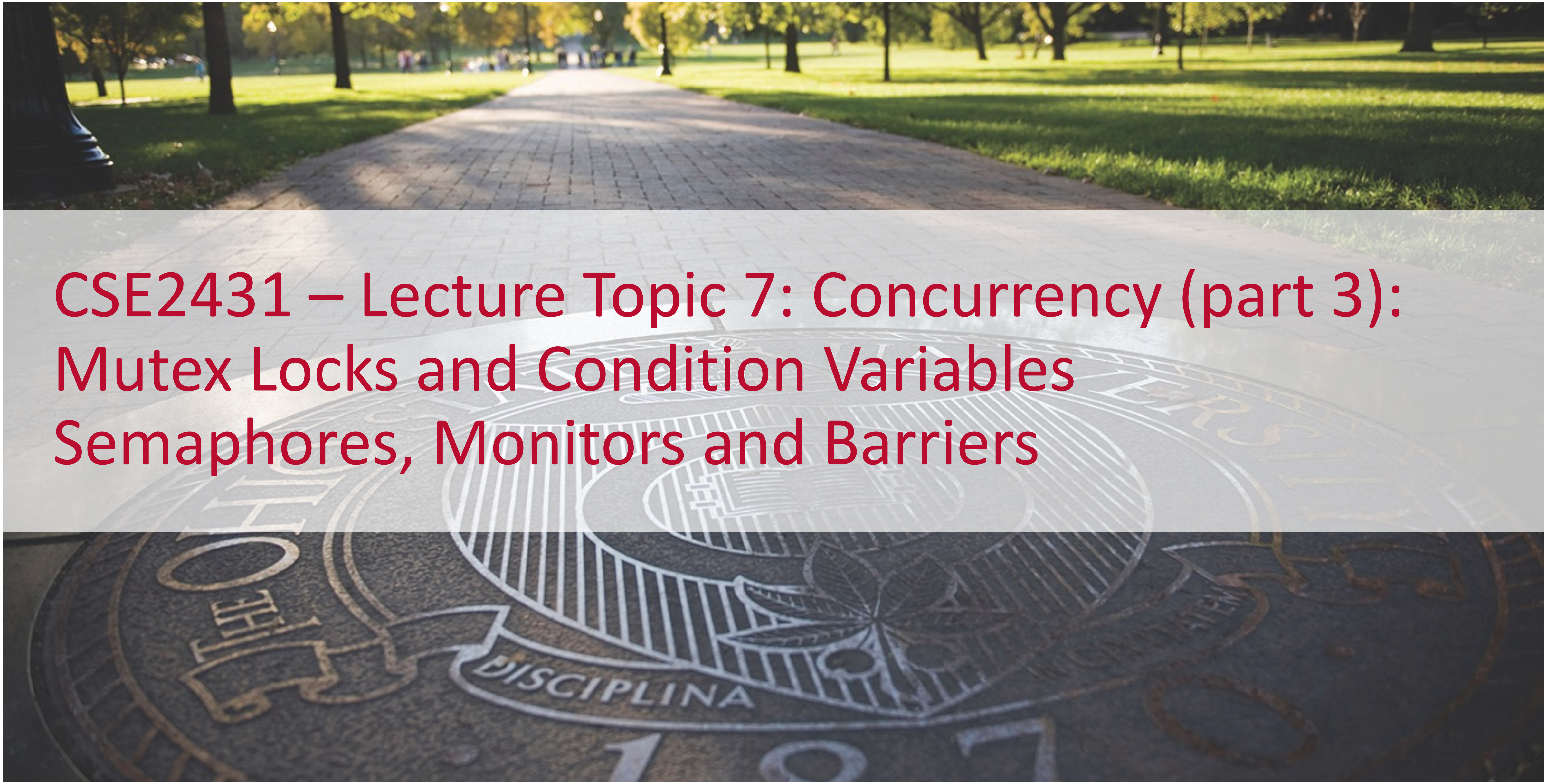
CSE2431 – Lecture Topic 7: Concurrency (part 3): Mutex Locks and Condition Variables Semaphores, Monitors and Barriers



THE OHIO STATE UNIVERSITY
COLLEGE OF ENGINEERING

Concurrency (Part 3)

Instructor: Luan Duong, Ph.D.

CSE 2431: Introduction to Operating Systems

Reading: **Chap. 30, 31 [OSTEP]**

Outline: Concurrency (part 3)

- Critical Regions
- Synchronization via busy waiting (a.k.a. Locks)
- Improve busy waiting (reduce spinning, sleep and wakeup)
- **Mutex Locks and condition Variables**
- Semaphores
- Monitors
- Barriers
- Classic Synchronization Problems

Review: “Regular” Locks

- Used to ensure mutual exclusion
- Example usage:

```
lock_t mutex; // a globally-allocated lock ‘mutex’
. . .
lock(&mutex);
balance = balance + 1;
unlock(&mutex);
```

Wait-For Relationship

- Multithreaded applications frequently need another functionality: implementing the **wait-for** relationship

- **Examples:**
  - A thread assigns some tasks to others, waiting for them to complete
  - A thread is waiting for another task to release some resource
  - ……

Condition Variables

- Sometimes a thread must wait for another thread to do something
- The logic appears like the following code:
  - Thread T1 wants to continue **only after** T2 has finished some task

Thread 1	Thread 2
`while (ready == 0)` `  ; // spin`	`ready = 1;`

Never write code like this! It's inefficient. (busy-waiting, but it might also be incorrect!)
Use condition variables instead.

Condition Variables (CVs)

- What is Condition Variable (CV)? Is a queue that a thread can put itself into when waiting on some condition.
- Another thread that makes the condition true can signal the CV to wake up a waiting thread
- Pthread in Linux provides CV for user programs:
  - OS has a similar functionality of wait/signal for kernel threads.
- Signal wakes up one thread, signal broadcast wakes up all waiting threads.

Using Condition Variables

```c
boolean done = false;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c = PTHREAD_COND_INITIALIZER;
```

```c
…
pthread_mutex_lock(&m);
done = true;
pthread_cond_broadcast(&c);
pthread_mutex_unlock(&m);
…
```

```c
…
pthread_mutex_lock(&m);
while (!done)
    pthread_cond_wait(&c, &m);
pthread_mutex_unlock(&m);
…
```

Semantics of Condition Variables

- A conditional variable provides two basic functionalities: `wait()` and `signal()` / `broadcast()`
- If a thread calls `wait()`, the thread will block/sleep until another thread calls `signal()` / `broadcast()` on the same condition variable
- If multiple threads `wait()` on the same condition variable, a `signal()` call will wake up a random one; a `broadcast()` will wake up all of them
- If thread A calls `wait()` after thread B calls `signal()` / `broadcast()`, A will be blocked.

Condition Variables (CVs): Example

```
1  int done  = 0;
2  pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
3  pthread_cond_t c  = PTHREAD_COND_INITIALIZER;
4
5  void thr_exit() {
6      Pthread_mutex_lock(&m);
7      done = 1;
8      Pthread_cond_signal(&c);
9      Pthread_mutex_unlock(&m);
10 }
11
12 void *child(void *arg) {
13     printf("child\n");
14     thr_exit();
15     return NULL;
16 }
17
18 void thr_join() {
19     Pthread_mutex_lock(&m);
20     while (done == 0)
21         Pthread_cond_wait(&c, &m);
22     Pthread_mutex_unlock(&m);
23 }
24
25 int main(int argc, char *argv[]) {
26     printf("parent: begin\n");
27     pthread_t p;
28     Pthread_create(&p, NULL, child, NULL);
29     thr_join();
30     printf("parent: end\n");
31     return 0;
```

Parent Waiting for Child: Using a Condition Variable

From main()

1. Parent create child thread [Line 28]
2. Now if child execute, then its ok.
3. But what if parent continues its execution?
4. Look at next line 29 for parent: it calls out "thr_join()". In thr_join(), parents take the lock &m (suppose it is free). But! when it check condition (done==0), it sees that the child has not done yet! Thus parent will put itself into sleep with pthread_cond_wait(&c, &m), hence releasing the lock &m
5. The child then will take the lock &m and execute (since child is the only process left)
6. Then when child is done executing, it signals back to parent by changing variable done to 1, and then release the lock (&m)

Always Use Mutex Locks with Condition Variables

```
boolean done = false;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c = PTHREAD_COND_INITIALIZER;
```

```
…
pthread_mutex_lock(&m);
done = true;
pthread_cond_broadcast(&c);
pthread_mutex_unlock(&m);
…
```

```
…
pthread_mutex_lock(&m);
while (!done)
    pthread_cond_wait(&c, &m);
pthread_mutex_unlock(&m);
…
```

Each condition variable should always be used together with a mutex lock. **`pthread_cond_wait()`** releases the lock when sleeping, **but** it will grab the lock again when it is awakened. So, when sleeping, another thread can change the value of `done`.

Always Use wait() on some condition (i.e. done)

```
void thr_exit() {
    Pthread_mutex_lock(&m);
    Pthread_cond_signal(&c);
    Pthread_mutex_unlock(&m);
}

void thr_join() {
    Pthread_mutex_lock(&m);
    Pthread_cond_wait(&c, &m);
    Pthread_mutex_unlock(&m);
}
```

Take a look at this example.

What is wrong with this?

Always wait() on Some Condition (1)

```
boolean done = false;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c = PTHREAD_COND_INITIALIZER;
```

```
…
pthread_mutex_lock(&m);
done = true;
pthread_cond_broadcast(&c);
pthread_mutex_unlock(&m);
…
```

```
…
pthread_mutex_lock(&m);
while (!done)
    pthread_cond_wait(&c, &m);
pthread_mutex_unlock(&m);
…
```

Always call wait() on some condition

Always wait() on Some Condition (2)

```
boolean done = false;
pthread_mutex_t m = …
pthread_cond_t c = …
```

Any problems?

```
…
pthread_mutex_lock(&m);
done = true;
pthread_cond_broadcast(&c);
pthread_mutex_unlock(&m);
…
```

```
…
pthread_mutex_lock(&m);
while (!done)
    pthread_cond_wait(&c, &m);
pthread_mutex_unlock(&m);
…
```

If `wait()` happens after `broadcast()`, then the thread may never wake up!

Always Use `While` instead of `If` (1)

```
boolean done = false;
pthread_mutex_t m = ….
pthread_cond_t c = …
```

Any additional potential problems?

```
…
pthread_mutex_lock(&m);
done = true;
pthread_cond_broadcast(&c);
pthread_mutex_unlock(&m);
…
```

```
…
pthread_mutex_lock(&m);
while (!done)
    pthread_cond_wait(&c, &m);
pthread_mutex_unlock(&m);
…
```

Always use `while` instead of `if`

Always Use `While` instead of `If` (2)

```
boolean done = false;
pthread_mutex_t m = ….
pthread_cond_t c = …
```

```
…
pthread_mutex_lock(&m);
done = true;
pthread_cond_broadcast(&c);
pthread_mutex_unlock(&m);
…
```

```
…
pthread_mutex_lock(&m);
while (!done)
    pthread_cond_wait(&c, &m);
pthread_mutex_unlock(&m);
…
```

Reason 1: Spurious wakeup. Sometimes a waiting thread may wake up without anyone calling signal() / broadcast(). **Use while to make it sleep.**

Always Use `While` instead of `If` (3)

```
int tickets = 0;
pthread_mutex_t m = ….
pthread_cond_t c = …
```

Any problems?

```
…
pthread_mutex_lock(&m);
ticket += 5;
pthread_cond_broadcast(&c);
pthread_mutex_unlock(&m);
…
```

Releases 5 tickets.

```
…
pthread_mutex_lock(&m);
if (!(ticket >= 2))
  pthread_cond_wait(&c, &m);
ticket -= 2;
pthread_mutex_unlock(&m);
…
```

Buys 2 tickets.

```
…
pthread_mutex_lock(&m);
if (!(ticket >= 4))
  pthread_cond_wait(&c, &m);
ticket -= 4;
pthread_mutex_unlock(&m);
…
```

Buys 4 tickets.

If we use if, both buying threads may believe they succeeded.
Reason 2: Just because a thread awakens, it does **not** mean the condition is met. **Use while to recheck the condition.**

Use `broadcast()` instead of `signal()` (1)

```
boolean done = false;
pthread_mutex_t m = ….
pthread_cond_t c = …
```

```
…
pthread_mutex_lock(&m);
done = true;
pthread_cond_broadcast(&c);
pthread_mutex_unlock(&m);
…
```

```
…
pthread_mutex_lock(&m);
if(!done)
    pthread_cond_wait(&c, &m);
pthread_mutex_unlock(&m);
…
```

Compared to `signal()`, `broadcast()` is always correct, although it might be inefficient. Consider **always using broadcast(), until its inefficiency becomes noticeable.**

Use broadcast() instead of signal() (2)

```
int tickets = 0;
pthread_mutex_t m = ….
pthread_cond_t c = …
```

Any problems?

```
…
pthread_mutex_lock(&m);
ticket += 3;
pthread_cond_signal(&c);
pthread_mutex_unlock(&m);
…
```

Releases 3 tickets.

```
…
pthread_mutex_lock(&m);
while(!(ticket >= 2))
  pthread_cond_wait(&c, &m);
ticket -= 2;
pthread_mutex_unlock(&m);
…
```

Buy 2 tickets.

```
…
pthread_mutex_lock(&m);
while(!(ticket >= 5))
  pthread_cond_wait(&c, &m);
ticket-=5;
pthread_mutex_unlock(&m);
…
```

Buy 5 tickets.

If using **signal()**, the third thread, which buys 5 tickets, may wake up; hence, no one gets tickets!

Outline: Concurrency (part 3)

- Critical Regions
- Synchronization via busy waiting (a.k.a. Locks)
- Improve busy waiting (reduce spinning, sleep and wakeup)
- Mutex Locks and condition Variables
- **Semaphores**
- Monitors
- Barriers
- Classic Synchronization Problems

Semaphores

- Another mechanism to solve concurrency problems
- Why teaching another one?
  - Semaphore can achieve the functionalities of both lock and condition variable (both a good thing and a bad thing).
  - It was introduced by Dijkastra.
  - All OS books teach that.
- You should try to master one of these two mechanisms (lock+CV vs semaphore), instead of using them interchangeably

Semaphores

- Introduced by Edsger Dijkstra.
- Heard his name before? Yes, he had quite a lot of well-known works:
  - Shortest Path algorithm (Dijkstra Algorithm)
  - “Goto Statements Considered Harmful”
  - Semaphore (in his THE operating system)
  - ALGOL
  - ...
- Turing Award 1972
- Professor in UT Austin from 1984-1999 (passed away in 2002)

Semaphores

- Synchronization primitive like condition variables.
- A semaphore is a **variable/object** with an underlying counter (integer value), representing **number of abstract resources**.
- New variable with two operations:
  - The **`sem_wait(), or down(),`** operation acquires a resource and decrements count.
  - The **`sem_post(), or up(),`** operation releases a resource and increments count.
- Also a **sem_init()** function to initialize the integer value
- Semaphore operations are indivisible (atomic)

Can we use Semaphore as a lock?

- The answer is YES
- But How?

Semaphores

- A semaphore with init value X=1 acts as a simple lock (binary semaphore = mutex)

```
sem_t m;
sem_init(&m, 0, X);

sem_wait(&m);
// critical section here
sem_post(&m);
```

Value of Semaphore	Thread 0	Thread 1
1		
1	call `sem_wait()`	
0	`sem_wait()` returns	
0	`(crit sect)`	
0	call `sem_post()`	
1	`sem_post()` returns	

Val	Thread 0	State	Thread 1	State
1		Run		Ready
1	call `sem_wait()`	Run		Ready
0	`sem_wait()` returns	Run		Ready
0	`(crit sect begin)`	Run		Ready
0	*Interrupt; Switch→T1*	Ready		Run
0		Ready	call `sem_wait()`	Run
-1		Ready	`decr sem`	Run
-1		Ready	`(sem<0)→sleep`	Sleep
-1		Run	*Switch→T0*	Sleep
-1	`(crit sect end)`	Run		Sleep
-1	call `sem_post()`	Run		Sleep
0	`incr sem`	Run		Sleep
0	`wake(T1)`	Run		Ready
0	`sem_post()` returns	Run		Ready
0	*Interrupt; Switch→T1*	Ready		Run
0		Ready	`sem_wait()` returns	Run
0		Ready	`(crit sect)`	Run
0		Ready	call `sem_post()`	Run
1		Ready	`sem_post()` returns	Run

Semaphore’s Values

- What does **value 1** mean?
  - No one is holding the lock
- What does **value 0** mean?
  - One thread is holding the lock and no threads are waiting.
- What does **negative value** mean?
  - One thread is holding the lock and some threads are waiting.
- Can value be **larger than 1**?
  - Yes, introduced by down() and up() operations

down() and up() operations

```
down(S) {
   S->value--;
   if (S->value < 0) {
     /* add this process to S->list; */
     block();
   }
}
up(S) {
   S->value++;
   if (S->value <= 0) {
     /* remove a process P from S->list; */
     wakeup (P);
   }
}
```

- Counting semaphores: 0, ..., *N*
- Binary semaphores: 0, 1

Counting Semaphores

(1) `Semaphore sem ← 2`   (3) `sem ← 1`   (6) `sem ← 0`   (9) `sem ← -1`

(16) `sem ← 2`   (14) `sem ← 1`   (11) `sem ← 0`

Process P1

(2) `down(&sem);`

(4) `display();`

(10) `up(&sem); // Wake up P3`

Process P2

(5) `down(&sem);`

(7) `display();`

(13) `up(&sem);`

Process P3

(8) `down(&sem); // Sleep`

(12) `display();`

(15) `up(&sem);`

This is **one** possible execution sequence. The point is that semaphore sem ensures mutually exclusive access to the displays **regardless** of the order in which the CPU scheduler runs the processes.

Binary Semaphores

① `Semaphore sem ← 1`

⑪ `sem ← 1`

③ `sem ← 0`

⑧ `sem ← 0`

⑥ `sem ← -1`

Process P1

```
② down(&sem);
④ display();
⑦ up(&sem); // Wakes up P2
```

Process P2

```
⑤ down(&sem); // Sleep
⑨ display();
⑩ up(&sem);
```

This is **one** possible execution sequence. The point is that semaphore sem ensures mutually exclusive access to the display **regardless** of the order in which the CPU scheduler runs the processes.

Mutex: Binary Semaphore

- Variable with only two states
  - locked
  - unlocked
- Mutex is used for mutual exclusion
  - Can be implemented using TSL
  - Can be a specialization of semaphore (simplified version of semaphore)

Mutex Implementation Using Test-and-Set

- Using `Test_and_Set` (TSL) instruction to implement
  - `Mutex_lock`: set lock to 1
  - `Mutex_unlock`: set lock to 0

Mutex Implementation Using Test-and-Set

```
mutex_lock:
  TSL REGISTER, MUTEX    ; copy mutex to register, set mutex to 1
  CMP REGISTER, #0       ; was register zero?
  JZE ok                 ; if it was zero, mutex unlocked, so return
  CALL thread_yield      ; mutex is busy, schedule another thread
  JMP mutex_lock         ; try again later
ok: RET                  ; return to caller; critical region entered

mutex_unlock:
  MOVE MUTEX, #0         ; store a zero in mutex
  RET                    ; return to caller
```

Implementation of `mutex_lock`, `mutex_unlock` (in assembly)

Producer-Consumer Problem Using Semaphores

```
semaphore mutex = 1; /* Controls access to critical region; assume these variables are defined */
semaphore empty = N; /* Controls empty buffer slots for both methods */
semaphore full = 0;  /* Controls full buffer slots */
```

```
void producer(void) {
  int item;
  while (TRUE) {            /* TRUE is the constant 1 */
    item = produce_item();  /* Make item for buffer */
    down(&empty); /* Decrement empty-slot count; A1 */
    down(&mutex);     /* Enter critical region; A2 */
    insert_item(item);     /* Put new item in buffer */
    up(&mutex);       /* Leave critical region; A3 */
    up(&full);     /* Increment full-slot count; A4 */
  }
}
```

```
void consumer(void) {
  int item;
  while (TRUE) {                /* Infinite loop */
    down(&full); /* Decrement full-slot count; B4 */
    down(&mutex);     /* Enter critical region; B2 */
    item = remove_item(); /* Remove item from buf */
    up(&mutex);       /* Leave critical region; B3 */
    up(&empty); /* Increment empty-slot count; B1 */
    consume_item(item); /* Do something with item */
  }
}
```

We implement down(&mutex) using mutex_lock and up(&mutex) using mutex_unlock.

Mutex Semaphore Implementation

- Using **`mutex_lock`** and **`mutex_unlock`** to implement a counter semaphore
  - down(or P())
  - up() (or V())

Busy Waiting Semaphore Implementation (1)

```
class Semaphore {
  Mutex m;        // Mutual exclusion.
  int count;      // Resource count.
public:
  Semaphore( int count );
  void Down();
  boid Up();
};

static inline Semaphore::Semaphore( int count ) {
  count = count;
}
```

Busy Waiting Semaphore Implementation (2)

```
void Semaphore::Down(){
  mutex_lock(m);
  while (count == 0) {
    mutex_unlock(m);
    yield();
    mutex_lock(m);
  }
  count--;
  mutex_unlock(m);
}
```

```
void Semaphore::Up() {
  mutex_lock(m);
  count++;
  mutex_unlock(m);
}
```

Semaphore Implementation Using Sleep and Wakeup

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

Down(Semaphore *S) {
  S->value⟵ S->value - 1;
  if (S->value < 0) {
    add this process to S->list;
    yield();
  }
}
```

```
Up(Semaphore *S) {
  S->value⟵ S->value + 1;
  if (S->value <= 0) {
    remove a process P from S->list;
    wakeup(P);
  }
}
```

Skipped locks here to provide atomicity

Tradeoffs

- Busy waiting (spinlock)
  - Wastes CPU cycles
- Sleep and Wakeup (blocked lock)
  - Context switch overhead
- Hybrid competitive solution
  - Apply spinlocks if the waiting time is shorter than the context switch time
  - Use sleep and wakeup if the waiting time is longer than the context switch time
  - **Why?**
  - **What if you don't know the waiting time?**

Possible Deadlocks with Semaphores

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);          // Line P0 (NEW LINE)
        sem_wait(&empty);          // Line P1
        put(i);                    // Line P2
        sem_post(&full);           // Line P3
        sem_post(&mutex);          // Line P4 (NEW LINE)
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);          // Line C0 (NEW LINE)
        sem_wait(&full);           // Line C1
        int tmp = get();           // Line C2
        sem_post(&empty);          // Line C3
        sem_post(&mutex);          // Line C4 (NEW LINE)
        printf("%d\n", tmp);
    }
}
```

Possible Deadlocks with Semaphores

Example:

```
P0                              P1
share two semaphores S and Q
S ⟵ 1; Q ⟵ 1;

down(&S); // S = 0 --------->  down(&Q); // Q = 0
down(&Q); // Q = -1  <------>  down(&S); // S = -1

// P0 blocked                   // P1 blocked

             DEADLOCK!

up(&S);                         up(&Q);
up(&Q);                         up(&S);
```

Be Careful When Using Semaphores

```
// Violation of Mutual Exclusion
up(mutex);                  mutexUnlock();
criticalSection()           criticalSection();
down(mutex);                mutexLock();
// Deadlock Situation
down(mutex);                mutexLock(P);
criticalSection()           criticalSection();
down(mutex);                mutexLock(P);
// Violation of Mutual Exclusion (omit down(mutex)/mutexLock())
criticalSection()           criticalSection();
up(mutex);                  mutexUnlock();
// Deadlock Situation (omit up(mutex)/mutexUnlock())
down(mutex);                mutexLock();
criticalSection()           criticalSection();
```

Outline: Concurrency (part 3)

- Critical Regions
- Synchronization via busy waiting (a.k.a. Locks)
- Improve busy waiting (reduce spinning, sleep and wakeup)
- Mutex Locks and condition Variables
- Semaphores
- **Monitors**
- Barriers
- Classic Synchronization Problems

Monitors

- A simpler way to synchronize
- A set of programmer-defined operators:

```
monitor  monitor-name {
  // variable declaration
  public  entry P1(..);
    { ... };
   ......
  public  entry Pn(..);
    { ... };
  begin
    initialization code
  end
}
```

Monitor Properties

- Various threads cannot access internal implementation of monitor type
- Encapsulation provided by monitor type limits access to the local variables to local procedures only.
- Monitor construct **does not allow concurrent** access to all procedures defined within the monitor.
- **Only one thread/process can be active within the monitor at a time.**
- Synchronization is built-in.

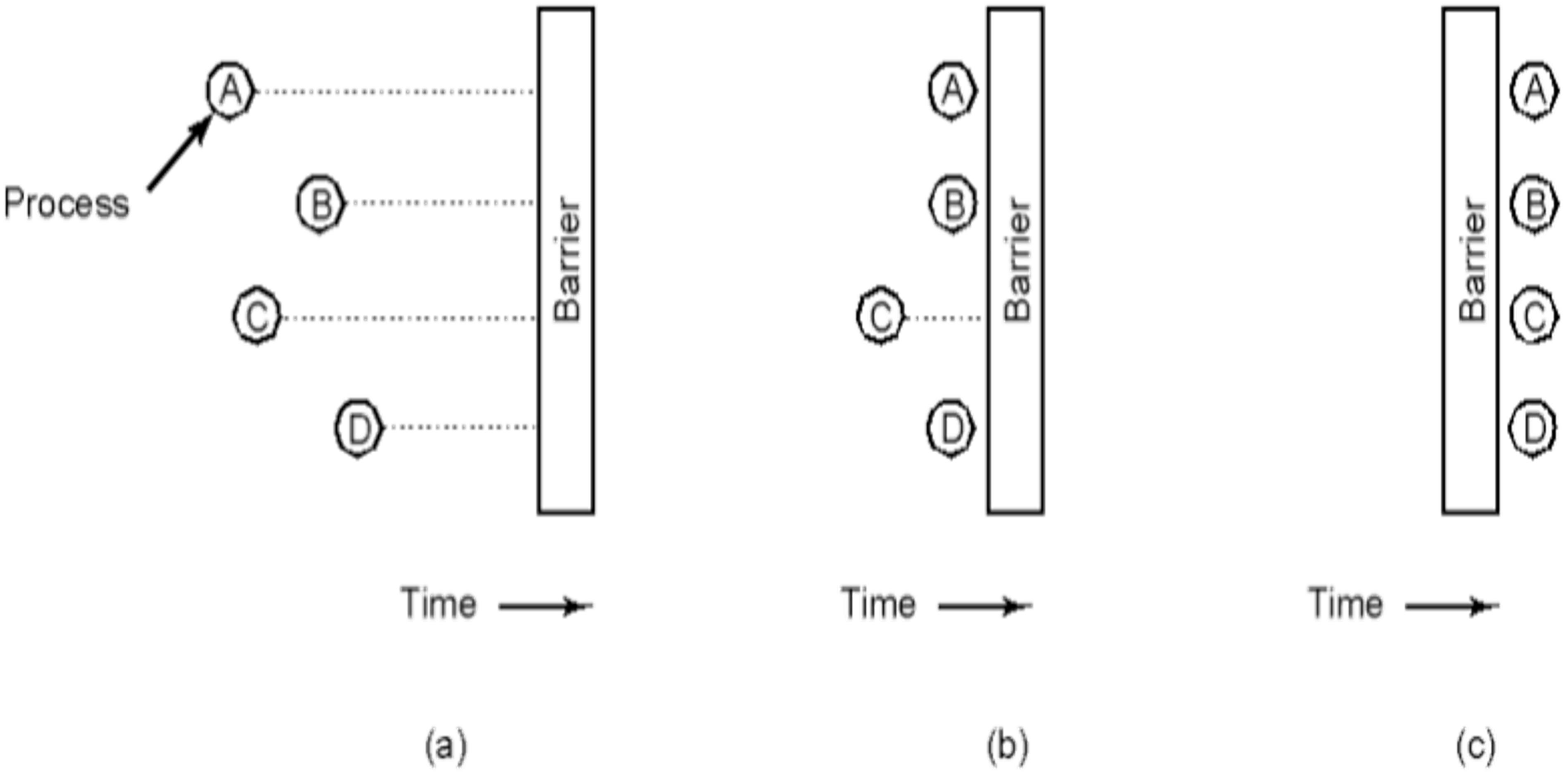
Outline: Concurrency (part 3)

- Critical Regions
- Synchronization via busy waiting (a.k.a. Locks)
- Improve busy waiting (reduce spinning, sleep and wakeup)
- Mutex Locks and condition Variables
- Semaphores
- Monitors
- **Barriers**
- Classic Synchronization Problems

Barriers (1)

- Use of a barrier
  - Processes approaching a barrier
  - All processes but one blocked at barrier
  - Last process arrives, all are let through
- Problem:
  - Wastes CPU if workloads are unbalanced

Barriers (2)



Process
A
B
C
D
Barrier
Time
(a)
A
B
C
D
Barrier
Time
(b)
Barrier
A
B
C
D
Time
(c)

How to implement a Barrier?

- For *N* processes: using messages
- For *N* threads: using shared variables

Outline: Concurrency (part 3)

- Critical Regions
- Synchronization via busy waiting (a.k.a. Locks)
- Improve busy waiting (reduce spinning, sleep and wakeup)
- Mutex Locks and condition Variables
- Semaphores
- Monitors
- Barriers
- **Classic Synchronization Problems**

Classic Synchronization Problems

- Bounded-buffer problem
- Reader-writer problem
- Dining-philosophers problem
- Sleeping barber

Bounded Buffer Problem (1)

- Producer: in an infinite loop, produces one item each iteration into the buffer
- Consumer: in an infinite loop, consumes one item each iteration from the buffer
- Buffer size: only holds **at most *N*** items

Bounded Buffer Problem (2)

```
Semaphore mutex; // shared and initialized  to 1
Semaphore empty; // counts empty buffers, initialized to N
Semaphore full;  // counts full buffers, initialized to 0
```

```
// Producer
repeat
  ....
  /* produce an item in nextp  */
  ....
  down(&empty);
  down(&mutex);
  ....
  /* add nextp to buffer */
  ....
  up(&mutex);
  up(&full);
  .....
until false;
```

```
// Consumer
repeat
  ...
  down(&full);
  down(&mutex);
  ....
  /* remove an item from buffer to nextc */
  ....
  up(&mutex);
  up(&empty);
  ....
  /* consume the item in nextc */
  ....
until false;
```

Readers-Writers Problem

- Readers read data, and writers write data
- Rule:
  - Multiple readers can read the data simultaneously
  - Only one writer can write the data at any time
  - A reader and a writer cannot in critical section concurrently.
- Locking table: whether any two can be in the critical section simultaneously

	Reader	Writer
Reader	**OK**	No
Writer	No	No

Readers-Writers Solution

Does it work? Why?

Consider various scenarios; any problems with this solution?

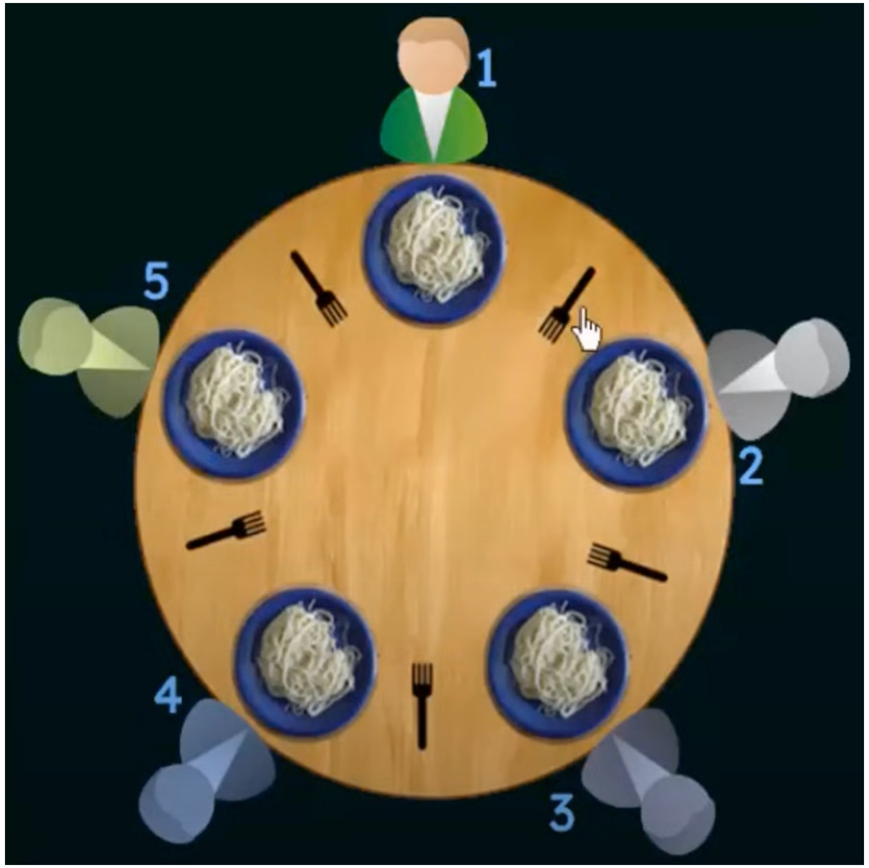
```
Semaphore mutex, wrt;  // shared and initialized to 1
int readcount;         // shared and initialized to 0
```

```
// Writer

down(&wrt);
......
/* writing performed */
......

up(&wrt);
```

```
// Reader

down(&mutex);
readcount ⟵ readcount + 1;
if (readcount == 1) then down(wrt);
up(&mutex);
......
/* reading performed */
......
down(&mutex);
readcount ⟵ readcount - 1;
if (readcount == 0) then up(wrt);
up(&mutex);
```

Dining Philosophers Problem

- Philosophers eat, think
- Eating requires two forks
- Pick one fork at a time
- Possible deadlock?
- If so, how to prevent deadlock?



Dining Philosophers Problem: Solution

```
#define N 5                   /* Number of philosophers */

void philosopher(int i) { /* i is philosopher num, from 0 to 4 */
  while (TRUE) {
    think();                  /* Philosopher thinks */
    take_fork(i);             /* Take left fork */
    take_fork((i+1) % N);     /* Take right fork; % is modulo operator */
    eat();                    /* Eat */
    put_fork(i);              /* Put left fork back on table */
    put_fork((i+1) % N);      /* Put right fork back on table */
  }
}
```

NOT a solution to the dining philosophers problem

Dining Philosophers Problem: Real Solution

```
#define N 5                /* Number of philosophers */
#define LEFT (i+N-1)%N     /* Number of i's left neighbor */
#define RIGHT (i+1)%N      /* Number of i's right neighbor */
#define THINKING 0         /* Philosopher thinking */
#define HUNGRY 1           /* Philosopher trying to get forks */
#define EATING 2           /* Philosopher eating */
typedef int semaphore;     /* Semaphores are special ints */
int state[N];              /* Array to track everyone's state (thinking, hungry, or eating) */
semaphore mutex = 1;       /* Mutual exclusion for critical regions */
semaphore s[N];            /* One semaphore per philosopher */

void philosopher(int i) { /* i is philosopher number (from 0 to N-1) */
  while (true) {           /* Infinite Loop */
    think();               /* Philosopher thinks */
    take_forks(i);         /* Grab two forks or block */
    eat();                 /* Eat your food */
    put_forks(i);          /* Put forks back on table */
  }
}
```

How do we implement **take_forks()** and **put_forks()**?

Dining Philosophers Problem: Real Solution

```
void take_forks(int i) {  /* i: philosopher number, from 0 to N-1 */
  down(&mutex);           /* Enter critical region; B1 */
  state[i] = HUNGRY;      /* Record that philosopher is hungry */
  test(i);                /* Try to acquire two forks; C1 */
  up(&mutex);             /* Exit critical region; B2 */
  down(&s[i]);            /* Block if forks not acquired; A1 */
}
void put_forks(int i) {   /* i: philosopher number, from 0 to N-1 */
  down(&mutex);           /* Enter critical region; B3 */
  state[i] = THINKING;    /* Philosopher finished eating */
  test(LEFT);             /* Check if left neighbor can now eat; C2 */
  test(RIGHT);            /* Check if right neighbor can now eat; C3 */
  up(&mutex);             /* Exit critical region; B4 */
}
void test(int i) {      /* i: philsopher number, from 0 to N-1 */
  if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
    STATE[i] = EATING; /* If philosopher i is hungry and i’s neighbors not eating, i can eat */
    up(&s[i]);         /* Unblock any of i’s neighbors so they can eat; A2 */
  }
}
```

The Sleeping Barber Problem

- N customer chairs
- One barber can cut one customer's hair at any time
- If no customer, barber sleeps

The Sleeping Barber Problem: Solution (1)

```
#define CHAIRS 5               /* # chairs for waiting customers */
typedef int semaphore;         /* use your imagination */
semaphore customers = 0;       /* # of customers waiting for service */
semaphore barbers = 0;         /* # of barbers waiting for customers */
semaphore mutex = 1;           /* for mutual exclusion */
int waiting = 0;               /* customers are waiting (not being cut) */
```

The Sleeping Barber Problem: Solution (2)

```
void barber (void) {
  while (TRUE) {
    down(&customers);       /* go to sleep if # of customers is 0 */
    down(&mutex);           /* acquire access to 'waiting' */
    waiting = waiting - 1; /* decrement count of waiting customers */
    up(&barbers); /* one barber is now ready to cut hair */
    up(&mutex); /* release 'waiting' */
    cut_hair(); /* cut hair (outside critical region) */
  }
}
```

The Sleeping Barber Problem: Solution (3)

```
void customer(void) {
  down(&mutex);               /* enter critical region */
  if (waiting < CHAIRS) (     /* if there are no free chairs, leave */
    waiting = waiting + 1;    /* increment count of waiting customers */
    up(&customers);           /* wake up barber if necessary */
    up(&mutex);               /* release access to 'waiting' */
    down(&barbers):           /* go to sleep if # of free barbers is 0 */
    get_haircut();            /* be seated and be serviced */
  } else {
    up(&mutex);               /* shop is full; do not wait */
  }
}
```

Summary (1)

- Critical region and mutual exclusion
- Mutual exclusion using busy waiting
  - Disabling Interrupts
  - Lock Variables
  - Strict Alternation
  - TSL
  - Sleep and Wakeup
- Semaphores
- Monitor and Barrier

Summary (2): Important Notes

- Synchronization is very important in OS when **accessing kernel data structures**
- System performance **may vary considerably**, depending on kind of sync. primitive selected
- **Rule of thumb adopted by kernel devs:** Always maximize system concurrency level
- Concurrency level **depends on two factors**:
    - Number of I/O devices that operate concurrently
    - Number of CPUs that do productive work
- To maximize I/O throughput, interrupts should be **disabled for short times**
- To use CPUs efficiently, sync primitives based on **spinlocks should be avoided** whenever possible
- **Choice of sync primitives** depends on kernel control flows, access data structures