

Lecture 12:

Concurrency Part II

Professor G. Sandoval

Some slides derived from : William Stallings, Tanenbaum/Bo, John Regehr and Brendan Dolan-Gavitt
Thanks !!

Review from Last Time

- We've so far seen:
 - Hardware Support
 - Spinlocks
 - Mutexes
- There are still more synchronization primitives

Mutex

- Enforce mutual Exclusion
- Puzzle: Add semaphores to the following example to enforce mutual exclusion to the shared variable count.

Thread A

```
1 count = count + 1
```

Thread B

```
1 count = count + 1
```



Mutex

- Enforce mutual Exclusion

Thread A

```
1 mutex.lock()  
2     // critical section  
3     count = count + 1  
4 mutex.unlock()
```

Thread B

```
1 mutex.lock()  
2     // critical section  
3     count = count + 1  
4 mutex.unlock()
```



- Semaphores
- Monitors
- Barriers
- Read-Copy-Update
- PThreads

Semaphores

- We saw last time that one way to solve the lost wakeup problem is by using a lock
- Another way to solve this problem is to use a *semaphore*: an integer to count the number of wakeups pending



Photo Credit: AmosWolfe

Semaphore Operations

- Semaphores have two operations:
 - *down*: checks if semaphore is greater than 0, and decrements it if so; otherwise sleep
 - *up*: increments the value of the semaphore, and if it was previously zero, wakes up one of the sleeping processes at random
- Note that these operations are *atomic* and generally implemented as system calls
- Up never blocks; it always returns immediately

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;
```

```
/* number of slots in the buffer */
/* semaphores are a special kind of int */
/* controls access to critical region */
/* counts empty buffer slots */
/* counts full buffer slots */
```

```
void producer(void)
```

```
{
    int item;

    while (TRUE) {
        item = produce_item();
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}
```


```
/* TRUE is the constant 1 */
/* generate something to put in buffer */
/* decrement empty count */
/* enter critical region */
/* put new item in buffer */
/* leave critical region */
/* increment count of full slots */
```

```
void consumer(void)
```

```
{
    int item;

    while (TRUE) {
        down(&full);
        down(&mutex);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}
```

```
/* infinite loop */
/* decrement full count */
/* enter critical region */
/* take item from buffer */
/* leave critical region */
/* increment count of empty slots */
/* do something with the item */
```


- Semaphores
- Monitors
-  • Barriers
- Read-Copy-Update
- PThreads

Barriers

- Sometimes we have algorithms that are a mix of parallel and sequential steps
- When going from the parallel to the sequential phase, we may need to stop, collate results, etc.
- We want a way to force a group of processes / threads to wait until all have finished

Barrier Example

- Suppose we have an algorithm that relies on repeatedly updating a matrix by multiplying it:

$$M_n = M_{n-1} * C$$

- The matrix multiplication can be parallelized by splitting it into submatrices
- But we can't update the final matrix until all subthreads have finished – so we need a barrier after each iteration

Parallel Matrix Multiplication

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{bmatrix}$$

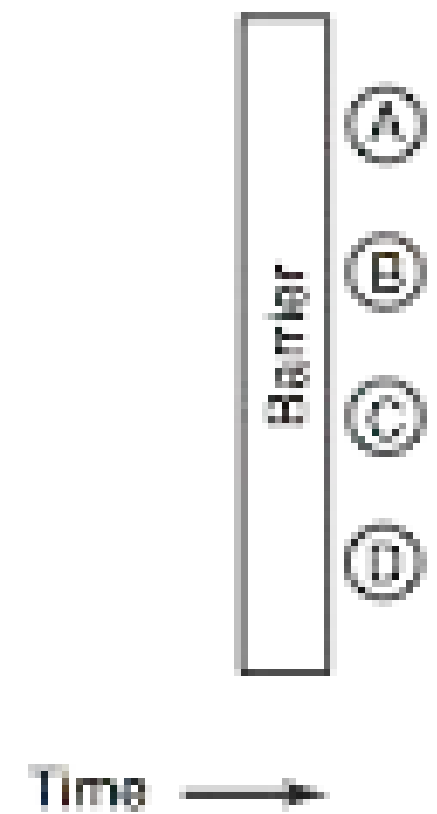
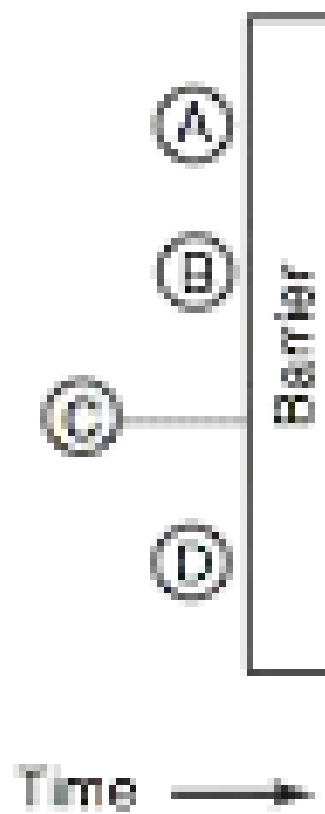
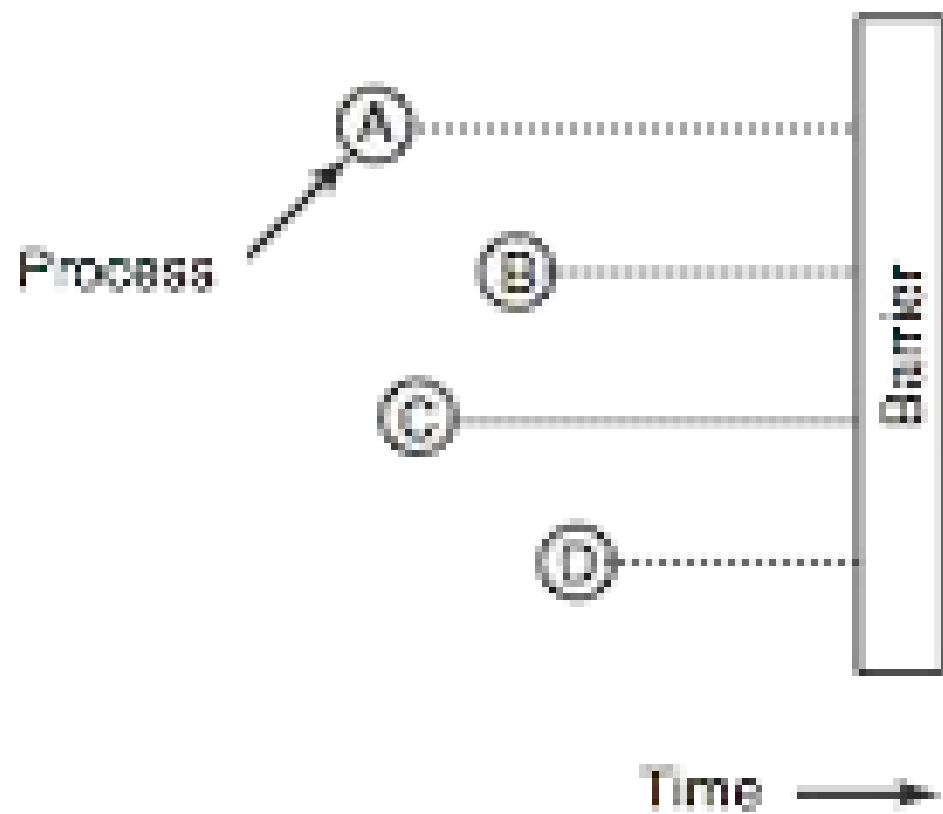
$$B = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{bmatrix}$$


$$A = \begin{bmatrix} a = \begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix} & b = \begin{bmatrix} 3 & 4 \\ 7 & 8 \end{bmatrix} \\ c = \begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix} & d = \begin{bmatrix} 3 & 4 \\ 7 & 8 \end{bmatrix} \end{bmatrix}$$

$$B = \begin{bmatrix} e = \begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix} & f = \begin{bmatrix} 3 & 4 \\ 7 & 8 \end{bmatrix} \\ g = \begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix} & h = \begin{bmatrix} 3 & 4 \\ 7 & 8 \end{bmatrix} \end{bmatrix}$$

$$A * B = \begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}$$

Barrier



- Semaphores
- Monitors
- Barriers
-  • Read-Copy-Update
- PThreads

Read-Copy-Update

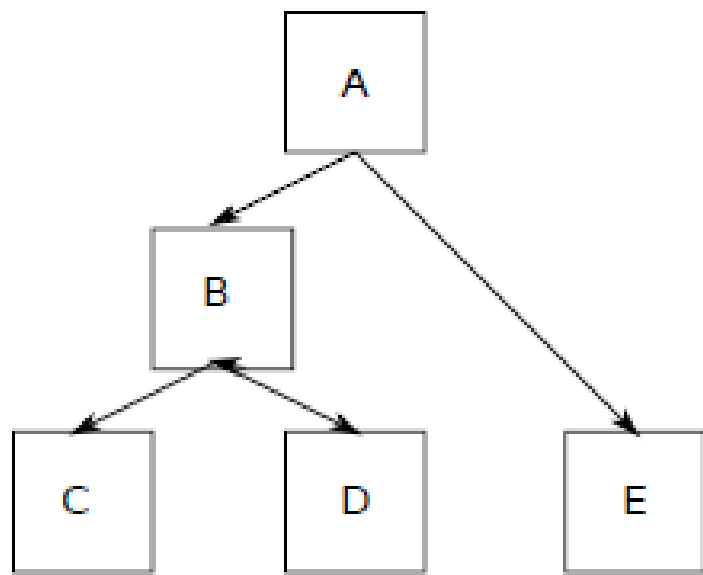
- Locking and mutual exclusion have inherent overhead
 - Time spent waiting for a lock is time you can't do other work
 - Reduces the speedup from multithreading / multiprocessing
- The best kind of locking is when you can avoid locking!

Read-Copy-Update

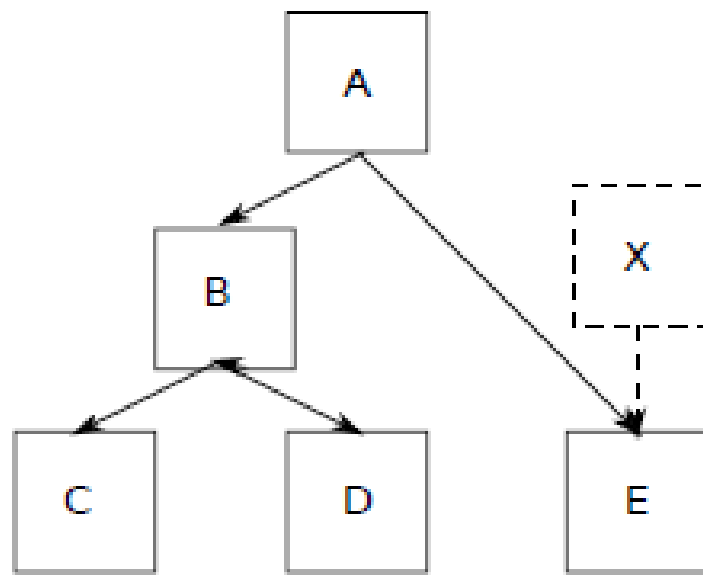
- If we're okay with some readers seeing an old version of a data structure for a while, we can sometimes avoid locking
- The trick is to update the structure but not free deleted items immediately – instead, wait until all readers are done

Avoiding Locks: Read-Copy-Update (1)

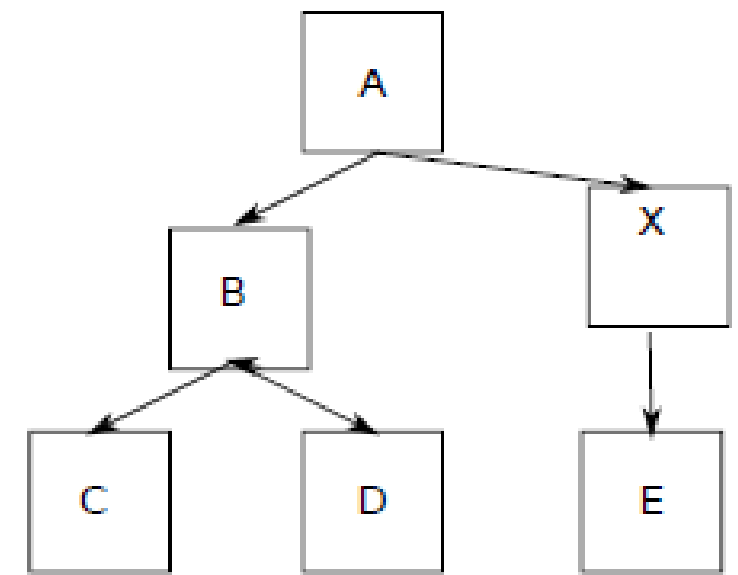
Adding a node:



(a) Original tree



(b) Initialize node X and connect E to X. Any readers in A and E are not affected.

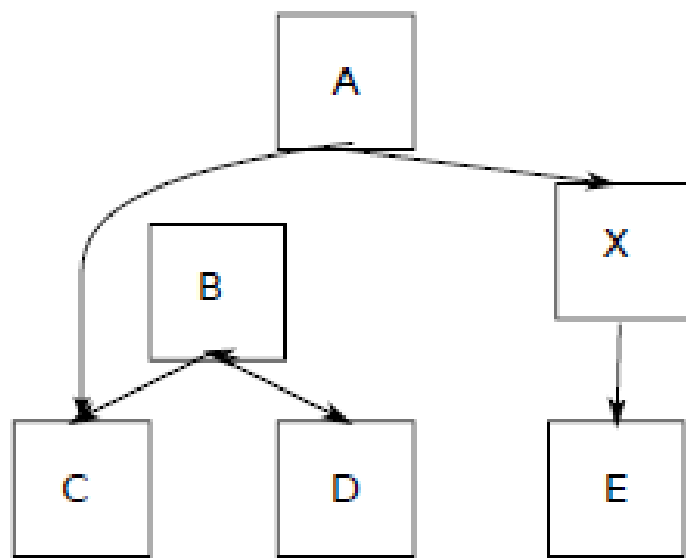


(c) When X is completely initialized, connect X to A. Readers currently in E will have read the old version, while readers in A will pick up the new version of the tree.

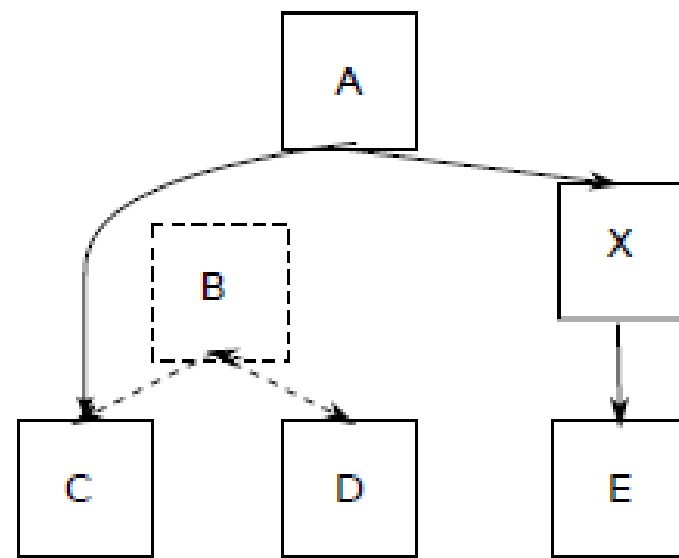
Read-Copy-Update: inserting a node in the tree and then removing a branch—all without locks

Avoiding Locks: Read-Copy-Update (2)

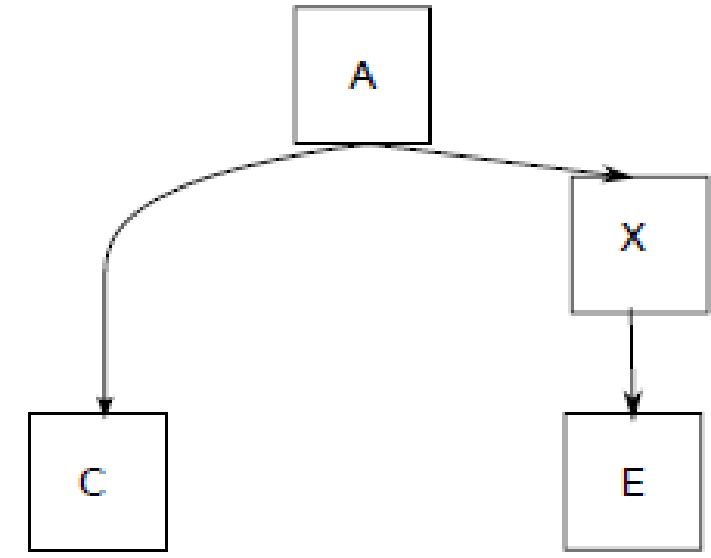
Removing nodes:



(d) Decouple B from A. Note that there may still be readers in B. All readers in B will see the old version of the tree, while all readers currently in A will see the new version.



(e) Wait until we are sure that all readers have left B and C. These nodes cannot be accessed by anymore.



(f) Now we can safely remove B and D

Read-Copy-Update: inserting a node in the tree and then removing a branch—all without locks

Read-Copy-Update API

- When can we safely delete old nodes? When there are no more readers
- We add an API called a *read-side critical section* that any readers of the structure must call when they read something from it
- Note that this doesn't involve mutual exclusion – it's just a way to track who's reading
- Now we know we can free a node when all the readers have left their critical sections

- Semaphores
- Monitors
- Barriers
- Read-Copy-Update
- PThreads



Synchronization in POSIX

- The standard way to do multithreading in C on POSIX (UNIX-like) systems is the *pthread* interface
- Allows creation & management of threads, and offers many standard synchronization primitives

pthread Basics

- Each thread is identified by a pthread_t data type
- Threads can be created using

```
int pthread_create(  
    pthread_t *thread,  
    const pthread_attr_t *attr,  
    void *(*start_routine) (void *),  
    void *arg  
);
```

- You can exit a thread by either returning from its start routine or calling

```
void pthread_exit(void *retval);
```

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 5

void *PrintHello(void *threadid)
{
    long tid;
    tid = (long)threadid;
    printf("Hello World! It's me, thread #%ld!\n", tid);
    pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
    pthread_t threads[NUM_THREADS];
    int rc;
    long t;
    for(t=0;t<NUM_THREADS;t++){
        printf("In main: creating thread %ld\n", t);
        rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
        if (rc){
            printf("ERROR; return code from pthread_create() is %d\n", rc);
            exit(-1);
        }
    }

    /* Last thing that main() should do */
    pthread_exit(NULL);
}
```

Mutexes

- You can create a mutex with
`int pthread_mutex_init(pthread_mutex_t *mutex,
 const pthread_mutexattr_t *attr);`
- Statically allocated mutexes can just do:
`pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;`
- Then, you have some standard operations on it:
`int pthread_mutex_lock(pthread_mutex_t *mutex);`
`int pthread_mutex_unlock(pthread_mutex_t *mutex);`

Coordinating Threads

- The main thread can *join* a thread to wait until it's finished

```
int pthread_join(pthread_t thread, void  
    **retval);
```

- By doing this in a loop for all threads we can implement a *barrier*

Mutexes in Pthreads

Thread call	Description
Pthread_cond_init	Create a condition variable
Pthread_cond_destroy	Destroy a condition variable
Pthread_cond_wait	Block waiting for a signal
Pthread_cond_signal	Signal another thread and wake it up
Pthread_cond_broadcast	Signal multiple threads and wake all of them

Some of the Pthreads calls relating to condition variables.

Mutexes in Pthreads

```
#include <stdio.h>
#include <pthread.h>
#define MAX 1000000000 /* how many numbers to produce */
pthread_mutex_t the_mutex; /* used for signaling */
pthread_cond_t condc, condp; /* buffer used between producer and consumer */
int buffer = 0; /* produce data */

void *producer(void *ptr)
{
    int i;

    for (i= 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
        while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
        buffer = i; /* put item in buffer */
        pthread_cond_signal(&condc); /* wake up consumer */
        pthread_mutex_unlock(&the_mutex); /* release access to buffer */
    }
    pthread_exit(0);
}
```

~~void *consumer(void *ptr)~~

Using threads to solve the
producer-consumer problem.

Mutexes in Pthreads

```
pthread_exit(0);  
}  
  
void *consumer(void *ptr)                /* consume data */  
{  
    int i;  
    for (i = 1; i <= MAX; i++) {  
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */  
        while (buffer == 0) pthread_cond_wait(&condc, &the_mutex);  
        buffer = 0;                      /* take item out of buffer */  
        pthread_cond_signal(&condp);     /* wake up producer */  
        pthread_mutex_unlock(&the_mutex); /* release access to buffer */  
    }  
    pthread_exit(0);  
}  
  
int main(int argc, char **argv)
```

Using threads to solve the
producer-consumer problem.