Synchronization Review

The Race Condition Problem

Critical Sections

Mutex Locks

Semaphores

Condition Variables

Barriers

Bibliography

- [Pacheco] : Peter Pacheco, Matthew Malensek, Introduction to Parallel Programming, 2nd Edition, Morgan Kaufmann Publisher, March 2020, **Chapter 4**
- To lookup the various functions: POSIX.1-2024: The IEEE Std 1003.1™-2024 and The Open Group Standard Base Specifications, Issue 8 https://pubs.opengroup.org/onlinepubs/9799919799/

Mutex locks

- Threaded APIs provide ***mutex-locks*** (mutual exclusion locks) as the most basic support for implementing **critical sections** and **atomic operations**

Critical section

Mutex lock

Mutex

<pthread.h>

pthread_mutex_t

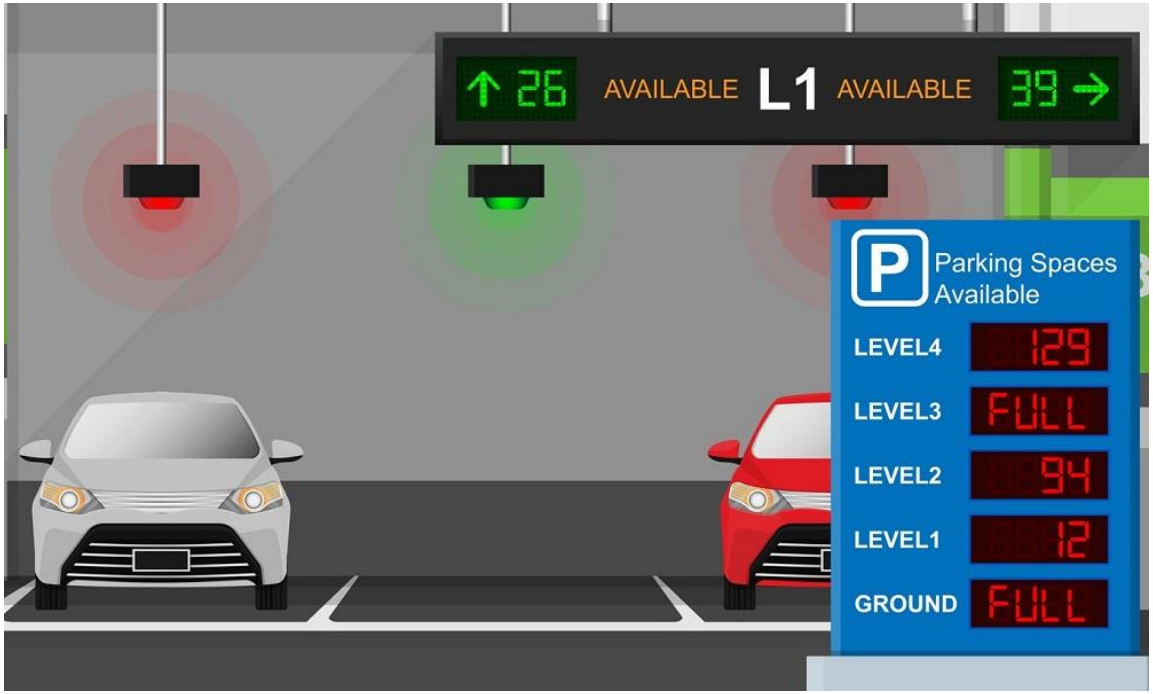
- int **pthread_mutex_init**(pthread_mutex_t *_mutex_, const pthread_mutexattr_t *_mutexattr_);
- int **pthread_mutex_destroy**(pthread_mutex_t *_mutex_);

- int **pthread_mutex_lock**(pthread_mutex_t *_mutex_));
- int **pthread_mutex_unlock**(pthread_mutex_t *_mutex_);

- int **pthread_mutex_trylock**(pthread_mutex_t *_mutex_);
- int **pthread_mutex_timedlock**(pthread_mutex_t *restrict mutex, const struct timespec *restrict abstime);

Semaphores

- Semaphore: another synchronization primitive, ***similar to a generalization of a Mutex.***
- **A semaphore is initialized to some value.** That value represents the ***number of threads allowed inside the protected region***. A Mutex is similar to a binary semaphore (value initialized with 1)



Semaphore functions

- #include <semaphore.h>
- **sem_t**
- int **sem_init**(sem_t *sem, int pshared, unsigned value);
- int **sem_destroy**(sem_t *sem);

- int **sem_wait**(sem_t *sem);
- int **sem_trywait**(sem_t *sem);
- int **sem_timedwait**(sem_t *restrict sem, const struct timespec *restrict abstime);

- int **sem_post**(sem_t *sem);

Condition Variables

- Another synchronization problem (**producer–consumer synchronization)**: ***a thread A can't proceed until another thread B has taken some action.***
  - *Thread A* ***waits*** *(is blocked) until it is* ***signaled (notified)*** *by thread B*

Condition Variables

- **pthread_cond_t**
- int **pthread_cond_init**(pthread_cond_t *restrict cond, const pthread_condattr_t *restrict attr);
- int **pthread_cond_destroy**(pthread_cond_t *cond);

- int **pthread_cond_wait**(pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex);
- int **pthread_cond_timedwait**(pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex, const struct timespec *restrict abstime);

- int **pthread_cond_signal**(pthread_cond_t *cond);
- int **pthread_cond_broadcast**(pthread_cond_t *cond);

Barriers

- Another synchronization problem: We need to perform a multi-threaded computation that has two stages (several threads execute each stage), but we don’t want to advance to the second stage until all threads finished the first stage.
    - Dinner table manners: do not start eating second course until everyone (including the slowest eater) finished eating the first course!

Barriers

- **pthread_barrier_t**
- int **pthread_barrier_init**(pthread_barrier_t *restrict barrier, const pthread_barrierattr_t *restrict attr, unsigned count);
- int **pthread_barrier_destroy**(pthread_barrier_t *barrier);

- int **pthread_barrier_wait**(pthread_barrier_t *barrier);

Increment shared counter

```
#define NUM_THREADS 2
#define REPEAT 1000000

pthread_mutex_t m;
/* shared variable */
int count = 0;

/* thread function */
void *inc_count(void *t)
{
   int my_id = *(int *)t;

   count++;

   return NULL;
}
```

lock

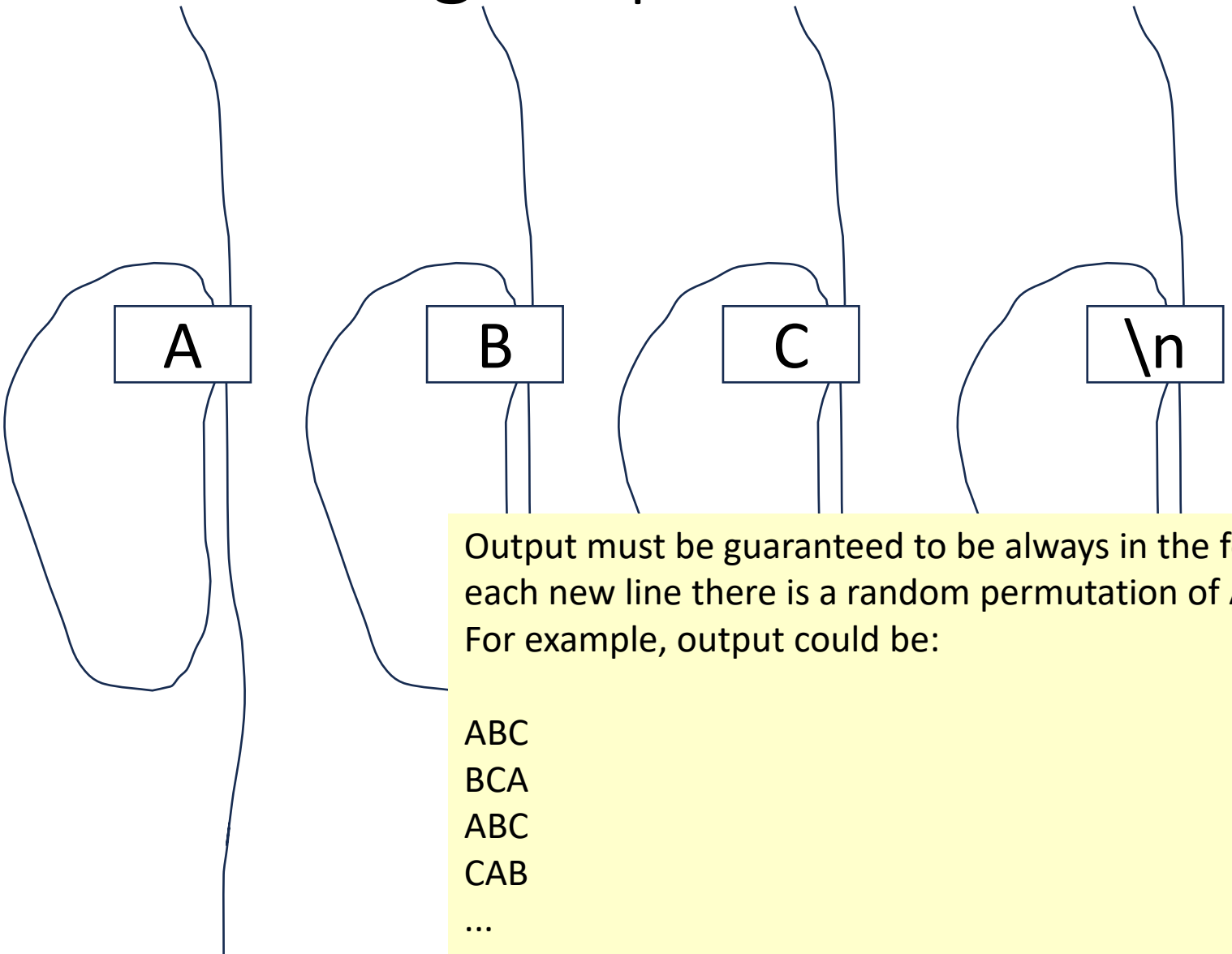
unlock

```
#define NUM_THREADS 2
#define REPEAT 1000000

pthread_mutex_t m;

/* shared variable */
int count = 0;

/* thread function */
void *inc_count(void *t)
{
   int my_id = *(int *)t;

   for (int i=0; i<REPEAT; i++){
            count++;
            }

   return NULL;
}
```

lock

unlock

Controlling output order

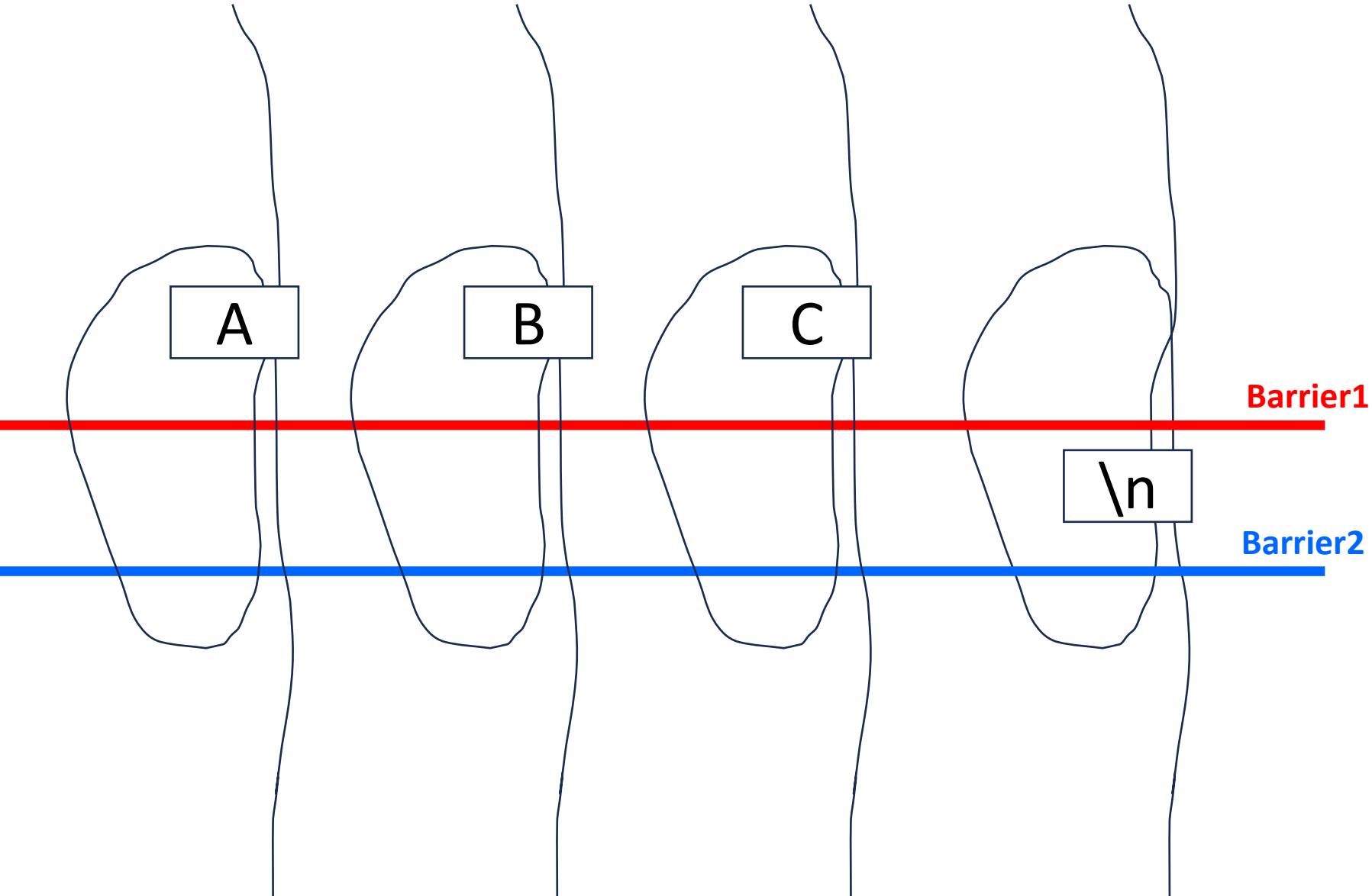


Output must be guaranteed to be always in the form: on each new line there is a random permutation of A,B,C. For example, output could be:

ABC
BCA
ABC
CAB
...

```
/* Thread function A */
void *HelloA(void *dummy)
{
   for (int i = 0; i < REPEAT; i++)
      printf("A");
   return NULL;
}

/* Thread function B */
void *HelloB(void *dummy)
{
   for (int i = 0; i < REPEAT; i++)
      printf("B");
   return NULL;
}

/* Thread function C */
void *HelloC(void *dummy)
{
   for (int i = 0; i < REPEAT; i++)
      printf("C");
   return NULL;
}
```

```
int main(int argc, char *argv[])
{
   pthread_t thread_handleA, thread_handleB,
                                    thread_handleC;
   pthread_create(&thread_handleA, NULL,
                        HelloA, NULL);
   pthread_create(&thread_handleB, NULL
                        HelloB, NULL);
   pthread_create(&thread_handleC, NULL,
                        HelloC, NULL);

   for (int i = 0; i < REPEAT; i++)
      printf("\n");

   pthread_join(thread_handleA, NULL);
   pthread_join(thread_handleB, NULL);
   pthread_join(thread_handleC, NULL);

   return 0;
}
```


A
B
C
\n
Barrier1
Barrier2

```
/* Thread function A */
void *HelloA(void *dummy)
{
   for (int i = 0; i < REPEAT; i++) {
      printf("A");
      pthread_barrier_wait(&b1);
      pthread_barrier_wait(&b2);
      }
   return NULL;
}
```

```
int main(int argc, char *argv[])
{
   pthread_t thread_handleA, thread_handleB,
                                    thread_handleC;
   pthread_barrier_init(&b1_barrier, NULL, 4);
   pthread_barrier_init(&b2, NULL, 4);
   pthread_create(&thread_handleA, NULL,
                        HelloA, NULL);
   pthread_create(&thread_handleB, NULL
                        HelloB, NULL);
   pthread_create(&thread_handleC, NULL,
                        HelloC, NULL);

   for (int i = 0; i < REPEAT; i++) {
      pthread_barrier_wait(&b1);
      printf("\n");
      pthread_barrier_wait(&b2);
      }

   pthread_join(thread_handleA, NULL);
   pthread_join(thread_handleB, NULL);
   pthread_join(thread_handleC, NULL);
   return 0;
}
```