CSCI 3753: Operating Systems Fall 2024

Dylan Sain

Department of Computer Science

University of Colorado Boulder

Week 6: POSIX Threads (Pthreads)

POSIX Threads (pthreads) Library

- Thread management: The first class of functions work directly on threads - creating, terminating, joining, etc.
- Mutexes: provide for creating, destroying, locking and unlocking mutexes.
- Semaphores: provide for creating, destroying, waiting, and posting on semaphores.
- Condition variables: include functions to create, destroy, wait, and signal based upon specified variable values.

POSIX Threads (pthreads) Library

- May be provided either as user-level or kernel-level
- Refer to the POSIX standard (IEEE 1003.1c) API for thread creation and synchronization.

```
#include <pthread.h>
```

Thread Creation

pthread_create(tid, attr, start_routine, arg)

- It returns the new thread ID via the *tid* argument.
- The *attr* parameter is used to set thread attributes, NULL for the default values.
- The *start_routine* is the C routine that the thread will execute once it is created.
- A single argument may be passed to *start_routine* via *arg*. It must be passed by reference as a pointer cast of type void.

Thread Termination and Join

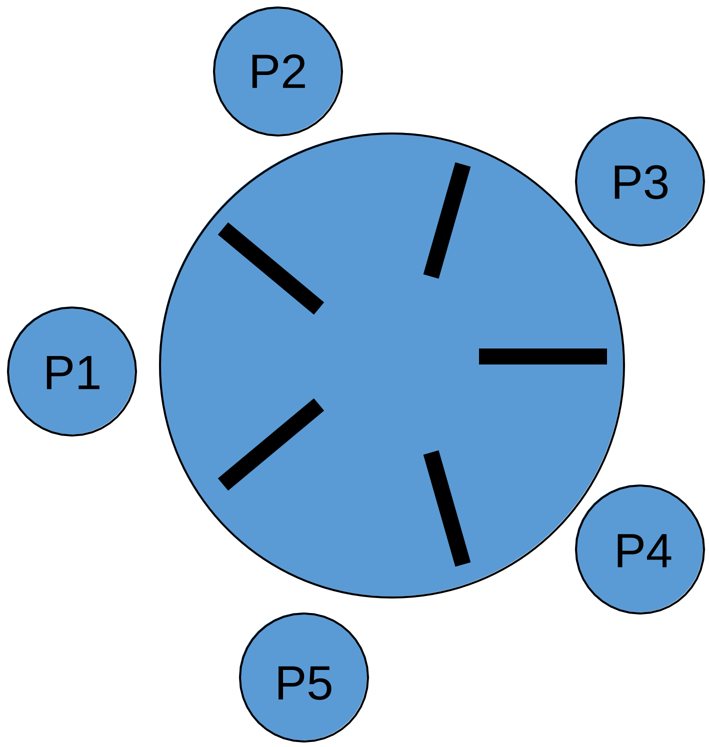
pthread_exit(value)

- This function is used by a thread to terminate
- The return value is passed as a pointer.

pthread_join(tid, value_ptr)

- The pthread_join() subroutine blocks the calling thread until the specified *threadid* thread terminates.
- Return 0 on success, and negative on failure. The returned value is a pointer returned by reference. If you do not care about the return value, you can pass NULL for the second argument.

Example (Dining Philosophers problem)



- Solution: Mutexes, Semaphores, and Condition Variables

*** Thread Pitfalls ***

- Race conditions
  - While the code may appear on the screen in the order you wish the code to execute, threads are scheduled by the operating system and are executed at random.
    - Threads may not execute in the order they are created
    - Threads may execute at different speeds
  - When threads are executing (racing to complete) they may give unexpected results (race condition)
    - Mutexes and joins must be utilized to achieve a predictable execution order and outcome.

*** Thread Pitfalls ***

- Thread safe code
  - The threaded routines must call functions which are "thread safe".
    - This means that there are no static or global variables which other threads may clobber or read assuming single threaded operation.
  - If static or global variables are used, then mutexes must be applied or the functions must be re-written to avoid the use of these variables.
    - In C, local variables are dynamically allocated on the stack.
  - Therefore, any function that does not use static data or other shared resources is thread-safe.

*** Thread Pitfalls ***

- Thread safe code
  - Thread-unsafe functions may be used by only one thread at a time in a program and the uniqueness of the thread must be ensured.
  - Many non-reentrant functions return a pointer to static data.
    - This can be avoided by returning dynamically allocated data or using caller-provided storage.

Thread Synchronization

- Mutex
  - Mutual exclusion locks (mutexes) are used to serialize the execution of threads through critical sections of code which access shared data.

- Semaphore
  - Semaphore is simply a variable that is non-negative and shared between threads. A semaphore is a signaling mechanism, and a thread that is waiting on a semaphore can be signaled by another thread. It uses two atomic operations, 1)wait, and 2) signal for the process synchronization.

Mutex vs Semaphore

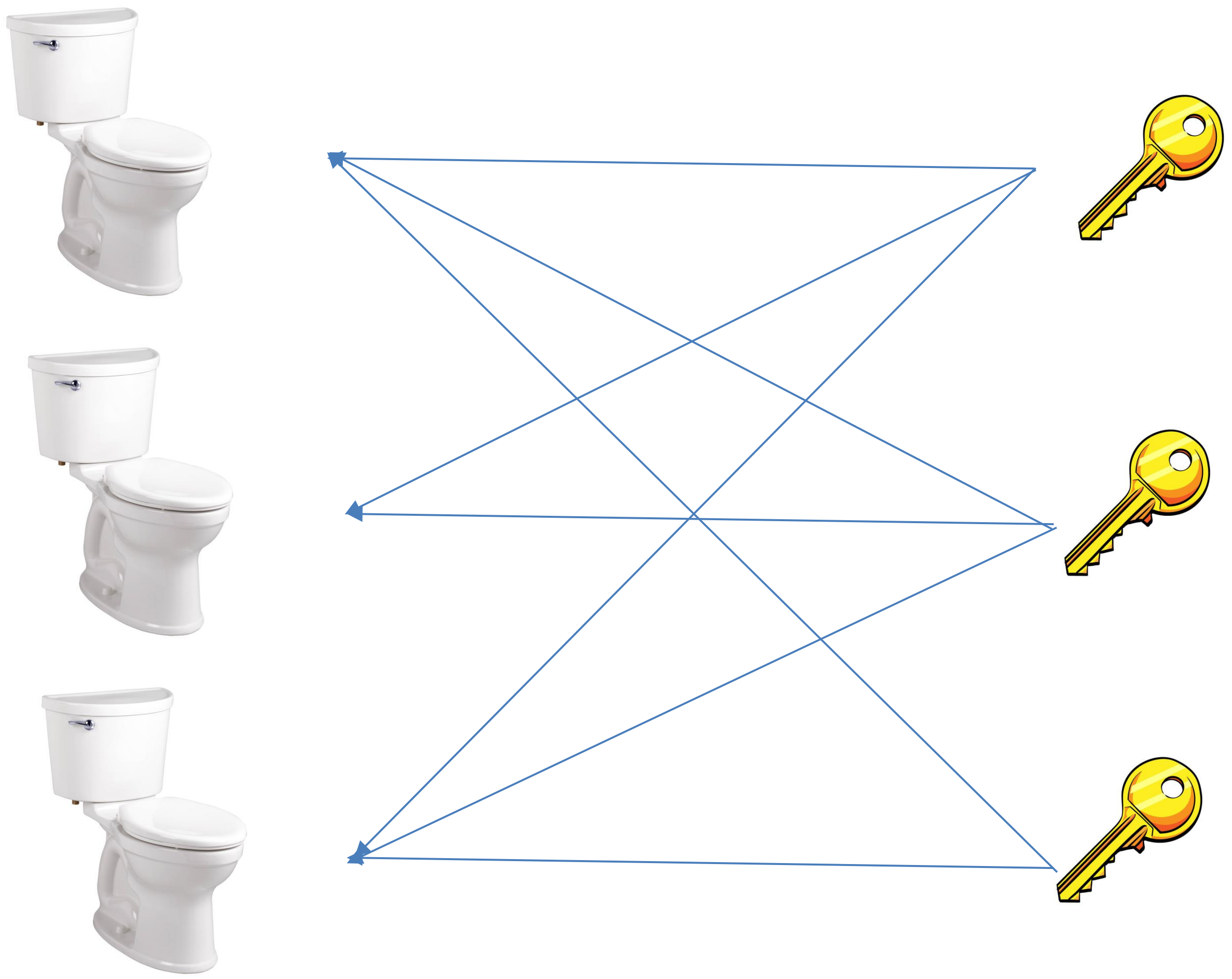
	Semaphore	Mutex
Mechanism	It is a type of **signaling** mechanism.	It is a **locking** mechanism.
Data Type	Semaphore is an **integer variable**.	Mutex is an **object**.
Modification	The wait and signal operations can modify a semaphore.	It is modified only by the process that may request or release a resource.
Resource Management	If no resource is free, then the process requires a resource that should execute wait operation. It should wait until the count of the semaphore is greater than 0.	If it is locked, the process has to wait. The process should be kept in a queue. This needs to be accessed only when the mutex is unlocked.
Thread	You can have multiple program threads simultaneously.	You can have multiple program threads in mutex but not simultaneously.
Ownership	Value can be changed by any process releasing or obtaining the resource.	Object lock is released only by the process, which has obtained the lock on it.

Mutex vs Semaphore

	Semaphore	Mutex
Types	+ Counting semaphore + Binary semaphore	No subtype
Operation	Semaphore value is modified using **wait () and signal ()** operation.	Mutex object is **locked or unlocked**.
Resources Occupancy	It is occupied if all resources are being used and the process requesting for resource performs wait () operation and blocks itself until semaphore count becomes > 1.	In case if the object is already locked, the process requesting resources waits and is queued by the system before lock is released.

The Toilet Example: Mutexes

The Toilet Example: Semaphores



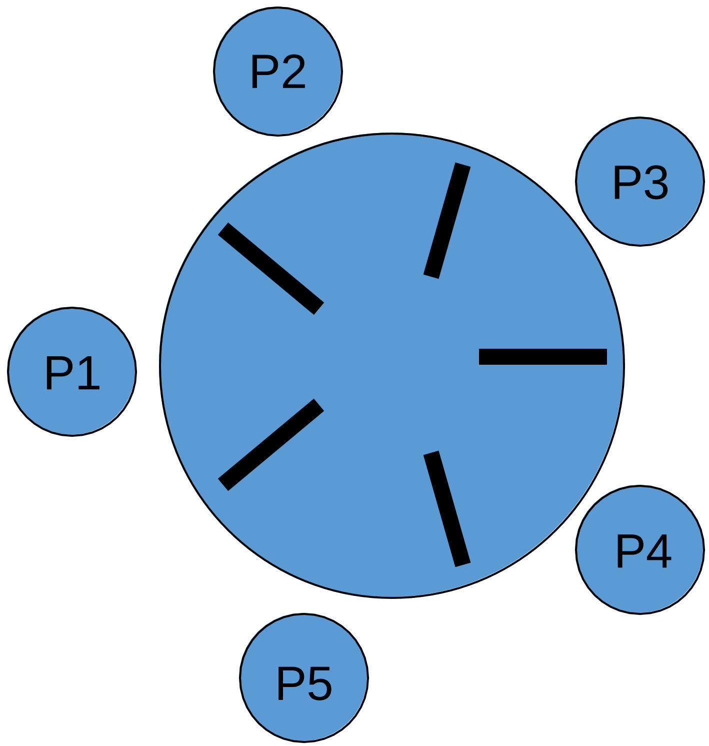
Thread Synchronization

- Mutex
  - Mutual exclusion locks (mutexes) are used to serialize the execution of threads through critical sections of code which access shared data.

```
int pthread_mutex_init(pthread_mutex_t *mutex, const
pthread_mutexattr_t *attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

  - A null value of `attr` initializes mutex with default attributes.

Dining Philosophers problem part 2



What happens if everyone picks up the chop stick to their left?

*** Thread Pitfalls ***

- Mutex deadlock
  - Occur when a mutex is applied but then not "unlocked".
    - Cause program execution to halt indefinitely.
  - It can also be caused by poor application of mutexes or joins.
    - Be careful when applying two or more mutexes to a section of code. If the first pthread_mutex_lock is applied and the second pthread_mutex_lock fails due to another thread applying a mutex, the first mutex may eventually lock all other threads from accessing data including the thread which holds the second mutex. The threads may wait indefinitely for the resource to become free causing a deadlock. It is best to test and if failure occurs, free the resources and stall before retrying

Mutex Deadlock

```
a_function(){
   lock(mutex a)
      //1st critical code
      lock(mutex b)
         //2nd critical code

      unlock(mutex b)
   unlock(mutex a)
}
```

```
b_function(){
   lock(mutex b)
      //1st critical code
      lock(mutex a)
         //2nd critical code

      unlock(mutex a)
   unlock(mutex b)
}
```

Thread Synchronization

- Condition variables
  - A mechanism
    - Allow threads to suspend execution and relinquish the processor until some condition is true.
    - Provide high performance synchronization primitives
  - A condition variable
    - A variable of type pthread_cond_t
    - Used with the appropriate functions for waiting and later, process continuation
    - *Must always be associated with a mutex to avoid a race condition* created by one thread preparing to wait and another thread which may signal the condition before the first thread actually waits on it resulting in a deadlock.

Thread Synchronization

- Condition variables

```
int pthread_cond_init(pthread_cond_t  *cond,
pthread_condattr_t *attr);

int pthread_cond_destroy(pthread_cond_t *cond);

int pthread_cond_signal(pthread_cond_t *cond);
```

~~int pthread_cond_broadcast(pthread_cond_t *cond);~~

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t
*mutex);
```

~~int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex, const struct timespec *abstime);~~

Thread Synchronization

- Condition variables
  - pthread_cond_wait() and pthread_cond_timedwait()
    - Used to block on a condition variable
    - Must be called with mutex locked by the calling thread
    - Atomically release mutex and block the calling thread on the condition variable `cond`. Before return, the mutex is reacquired for the calling thread.
  - pthread_cond_timedwait()
    - An error is returned if the absolute time specified by `abstime` passes before the waiting thread is signaled.
    - If a time-out occurs, it still reacquires mutex before returning to the caller.

Thread Synchronization

- Semaphore
  - Linked with the following library

```
#include <fcntl.h>
#include <sys/stat.h>
#include <semaphore.h>
```

  - Two types of POSIX semaphores
    - Named

```
sem_t *sem_open (const char *name, int oflag);
sem_t *sem_open (const char *name, int oflag, mode_t mode,
unsigned int value);
```

    - Unnamed

```
int sem_init (sem_t *sem, int pshared, unsigned int value);
```

Thread Synchronization

- Semaphore
  - To increment (unlock) the semaphore

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

  - To decrement (lock) the semaphore

```
int sem_wait (sem_t *sem);
```

~~int sem_trywait (sem_t *sem);~~
~~int sem_timedwait (sem_t *sem, const struct timespec *abs_timeout);~~

  - To get value of semaphore

```
int sem_getvalue (sem_t *sem, int *sval);
```

  - To destroy the unnamed semaphore

```
int sem_destroy (sem_t *sem);
```

Example (Producer Consumer problem)

- The producer is putting numbers into the shared buffer (in this case sequentially).
- The consumer is taking them out.
- The buffer can store only ONE value.

- Solution: Using ptheads, a mutex, and condition variables

Exercise: Test your skills!

Go to Kahoot.it
Use the code
Add your name (please use your real name)

Choose the correct answer!

Example (Producer Consumer problem: Mutex)

```
pthread_mutex_t M;
pthread_cond_t DC, DP;
int buffer = 0;

void* producer(void *ptr) {
  int i;
  for (i = 1; i <= MAX; i++) {
  (1) mutex_lock(&M);
      while (buffer != 0)
            (2) cond_wait(&DP, &M);
      buffer = i;
      printf("Producer wrote %d\n", buffer);
  (3) cond_signal(&DC);
  (4) mutex_unlock(&M);
  }
  pthread_exit(0);
}
```

```
void* consumer(void *ptr) {
  int i;
  for (i = 1; i <= MAX; i++) {
  (5) mutex_lock(&M);
      while (buffer == 0)
         (6) cond_wait(&DC, &M);
      printf("Consumer reads %d\n", buffer);
      buffer = 0;
  (7) cond_signal(&DP);
  (8) mutex_unlock(&M);
  }
  pthread_exit(0);
}
```

(Note: "pthread_" prefix removed from all synchronization calls for compactness)

Example (Producer Consumer problem: Semaphore)

```
sem_t mutex;
sem_t full;
sem_t empty;
int buffer = 0;
```

```
void* producer(void *ptr) {
  int i;
  for (i = 1; i <= MAX; i++) {
  (1)  sem_wait(&empty);
  (2)  sem_wait(&mutex);

       buffer = i;
       printf("Producer wrote %d\n", buffer);

  (3)  sem_post(&mutex);
  (4)  sem_post(&full);
  }
  pthread_exit(0);
}
```

```
void* consumer(void *ptr) {
  int i;
  for (i = 1; i <= MAX; i++) {
  (5)  sem_wait(&full);
  (6)  sem_wait(&mutex);

       printf("Consumer reads %d\n", buffer);
       buffer = 0;

  (7)  sem_post(&mutex);
  (8)
  }    );
  pthread_exit(0);
}
```