School of Informatics, Computing, and Cyber Systems

Pthreads: condition variables, barriers, semaphores

CS450/CS550: Parallel Programming

Week 4

Condition variables

- Condition variables are **synchronization** primitives that enable threads to wait until a particular condition occurs
- Condition variables are user-mode objects that **cannot be shared across processes**
- Condition variables enable threads to atomically release a lock and enter the sleeping state
- Condition variables should be used as a place to wait and **be notified**
- They are **not the condition** itself and they are not events

Create and destroy conditions

- Create a condition variable:
  `int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);`
- Destroy a condition variable:
  `int pthread_cond_destroy(pthread_cond_t *cond);`
  - returns 0 on success, an error code otherwise
  - `cond`: output parameter, condition
  - `attr`: input parameter, attributes (default is `NULL`)

Waiting and waking up for conditions

- Waiting on a condition:

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

- Unblock at least one of the blocked threads:

```
int pthread_cond_signal(pthread_cond_t *cond);
```

- Unblock all threads currently blocked by the condition variable:

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

  - functions return 0 on success, an error code otherwise
  - `cond`: input parameter, condition
  - `mutex`: input parameter, an associated mutex

Usage pattern of condition variables

- The typical usage pattern of condition variables is following:

```
pthread_mutex_lock(&lock);
while (the_condition)
    pthread_cond_wait(&cond, &lock);
do_stuff();
pthread_mutex_unlock(&lock);
```

- A thread, signalling the condition variable, typically looks like:

```
pthread_mutex_lock(&lock);
if (alter_condition)
    pthread_cond_signal(&cond);
pthread_mutex_unlock (&lock)
```

Example 1: Condition variables

Barriers

- A barrier is a type of synchronization method
- A barrier for a group of threads or processes in the source code means any thread/process must stop at this point and cannot proceed until all other threads/processes reach it
- A barrier may be in a raised or lowered state
- A **latch** is a barrier that starts in the raised state and cannot be re-raised once it is in the lowered state
- A **count-down latch** is a latch that is automatically lowered once a pre-determined number of threads/processes have arrived

Initialize and destroy barriers

- Initialize the barrier `bar`:

```
int pthread_barrier_init(pthread_barrier_t
*restrict bar, const pthread_barrierattr_t
*restrict attr, unsigned count);
```

  - `attr`: the barrier object attributes (`NULL` for default values)
  - `count`: the number of threads that must wait before any of them successfully return from the call

- Destroy the barrier `bar`:

```
int pthread_barrier_destroy(pthread_barrier_t
*bar);
```

The use of barriers

- Synchronize participating threads at the barrier:
  `int pthread_barrier_wait(pthread_barrier_t *barrier);`
- The calling thread shall block until the required number of threads have called `pthread_barrier_wait` specifying the barrier
- If a signal is delivered to a thread blocked on a barrier, the thread shall resume waiting at the barrier if the barrier wait has not completed
- A thread that has blocked on a barrier shall not prevent any unblocked thread that is eligible to use the same processing resources from eventually making forward progress in its execution

School of Informatics, Computing,
and Cyber Systems

Example 2: Barriers

Semaphores

- A **semaphore** is a variable or abstract data type used to control access to a common resource by multiple threads and avoid critical section problems in a concurrent system
- Semaphores are a useful tool in the prevention of race conditions
- Semaphores which allow an arbitrary resource count are called **counting semaphores**
- Semaphores which are restricted to the values 0 and 1 are called **binary semaphores**
- The semaphore concept was invented by Edsger Dijkstra in 1962
- To use semaphores in C, the **semaphore.h** header should be included

Create and destroy semaphores

- Initialize a semaphore:
  `int sem_init(sem_t *sem, int pshared, unsigned int value);`
  - `sem`: a pointer to a semaphore object to initialize
  - `pshared`: a flag indicating whether or not the semaphore should be shared with forked processes
  - `value`: an initial value to set the semaphore to
- Destroy a semaphore:
  `int sem_destroy(sem_t *sem);`

Operations with semaphores

- Lock the semaphore `sem` only if it is currently not locked:
  ```
  int sem_trywait(sem_t *sem);
  ```
- Lock the semaphore `sem`:
  ```
  int sem_wait(sem_t *sem);
  ```
- Unlock the semaphore `sem`:
  ```
  int sem_post(sem_t *sem);
  ```

Example 3: Semaphores

NAU NORTHERN ARIZONA UNIVERSITY
School of Informatics, Computing, and Cyber Systems

Getting semaphore values

- Get the value of a semaphore:
```
int sem_getvalue(sem_t *restrict sem,
 int *restrict sval);
```
- The `sem_getvalue` function updates the location referenced by the `sval` argument to have the value of the semaphore referenced by `sem` without affecting the state of the semaphore
- If `sem` is locked, then the object to which `sval` points shall either be set to zero or to a negative number whose absolute value represents the number of processes waiting for the semaphore at some unspecified time during the call

Binary semaphores

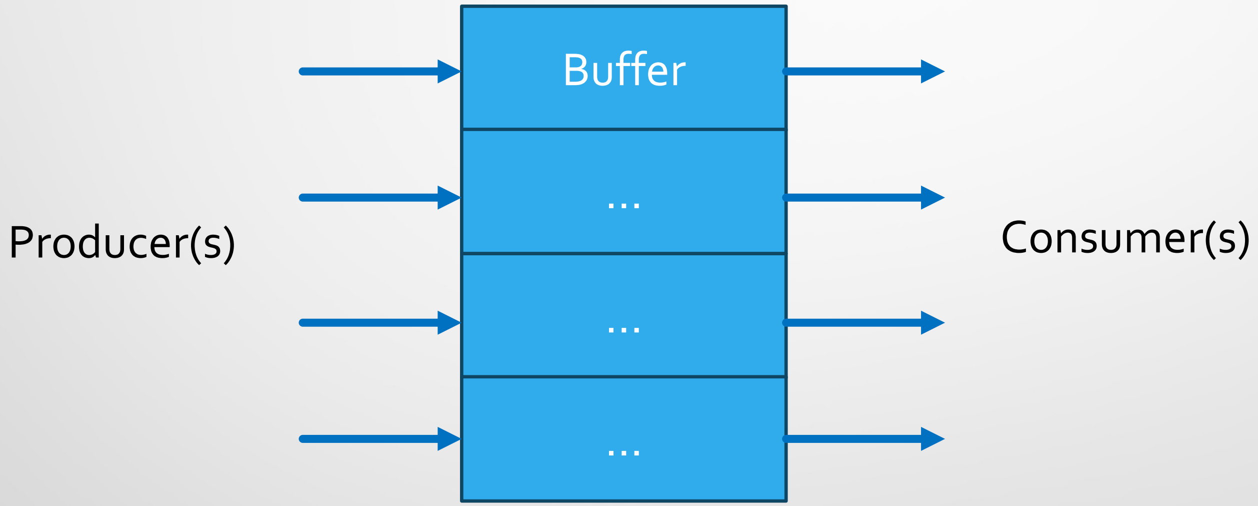
- Binary semaphores are synchronization mechanisms with integer values varying between 0 and 1
- Binary semaphores are used to implement locks
- Binary semaphores provide a single access unit to a critical section: only one entity can access the critical section at once

Example 4: Binary semaphores

NAU NORTHERN ARIZONA UNIVERSITY
School of Informatics, Computing, and Cyber Systems

Producer-Consumer Problem

- The producer-consumer problem (bounded-buffer problem) is a family of synchronization problems described by Dijkstra since 1965



School of Informatics, Computing, and Cyber Systems

Example 5: Producer-Consumer Problem