# Threads 4: Synchronization (part b)

1. Introduction
2. Creation and Termination
3. Synchronization (part a)
4. **Synchronization (part b)**
   - **mutexes vs. semaphores**
   - **mutex & semaphore differences**
   - **semaphore semantics with mutex+CV**
   - **joining with terminated threads**
   - **file locks**
   - **producer-consumer examples**
5. Miscellaneous

# Mutexes vs. POSIX Semaphores

**POSIX semaphores** can be used to synchronize processes *and* Pthreads.

Please see the **Process Synchronization lectures** for details of using POSIX semaphores.

Given that POSIX semaphores must be able to work between processes, mutexes will typically be *more efficient* for threads than will semaphores.

In particular, mutex operations can involve only machine code operations on common memory, while semaphore operations always require a syscall (kernel code).

Thus, if you are writing a multithreaded program, use mutexes (plus condition variables), not semaphores.

# Mutexes vs. Semaphores (contd.)

Unfortunately, many people are confused about the relationship between semaphores and mutexes, resulting in a great deal of *misinformation* posted on the Internet.

Key point: *conceptually*, there is *no difference at all* between a binary (0/1) semaphore and a mutex!

Dijkstra's original papers that introduced this synchronization mechanism referred to them as **semaphores**, whether used for mutal exclusion or for more general producer-consumer problems.

0/1 semaphores used to enforce mutual exclusion (i.e., mutex), Dijkstra termed **binary semaphores**.

Those with a wider range of positive values, Dijkstra termed **general semaphores**.

# Mutexes vs. Semaphores (contd.)

POSIX semaphores can supply the semantics of Dijkstra's binary and general semaphores.

(POSIX) mutexes can supply the semanatics of Dijkstra's binary semaphores only.

However, by combining a *mutex*, a *condition variable*, and a *shared integer*, it is possible to duplicate the semanatics of Dijkstra's general semaphores.

This means that to a large extent, mutexes and semaphores are functionally interchangeable in Linux/UNIX.

Nonetheless, there are differences (beyond Dijkstra's semantics) that can be useful to be aware of.

# Mutex & Semaphore Differences

Broadcast signaling/unlocking:

- `pthread_cond_broadcast()` signals *all* threads waiting on a condition variable
  - can be used when the shared resource has changed in a way that more than one thread can proceed (e.g., producer-consumer problems with multiple consumers, where the producer can add multiple items)
  - makes it easier to implement a **read-write lock**: wake up all waiting readers when a writer releases its lock
  - can be used in a **two-phase commit algorithm** to notify all clients of an impending commit
- semaphores do *not* have anything comparable (`sem_post()` wakes only a *single* blocked process)

# Mutex & Semaphore Differences (contd.)

Lock ownership:

- when a thread locks a *mutex* it is said to *own* the mutex
- *only the mutex owner* can unlock a locked mutex
- this is generally considered as enforcing good style
- no such ownership is enforced with POSIX semaphores
- any process can increment a semaphore decremented (to 0) by another process
- so must be careful when using semaphores for mutual exclusion

# Mutex & Semaphore Differences (contd.)

Testing/peeking:

- both mutexes and semaphores allow testing/peeking to see if lock/wait will block (without actually blocking)
- `pthread_mutex_trylock()`
- `sem_trywait()`
- `sem_getvalue()`

Limited-time locking/waiting:

- both mutexes (plus condition variables) and semaphores can limit the length of time lock/wait calls can block
- `pthread_mutex_timedlock()`
- `pthread_cond_timedwait()`
- `sem_timedwait()`

# Mutex & Semaphore Differences (contd.)

Lock count:

- mutexes can be set to support concept of **lock count**: single owner can lock multiple times and must unlock same number of times for lock to be released
- this is semantics beyond Dijkstra
- effectively like "binary semaphore" that can also go negative
- no comparable semantics for semaphores
- however, could duplicate effect with code using a second semaphore whose value was this lock count

# Mutex & Semaphore Differences (contd.)

Error situations:

- mutexes can be set so that errors are thrown on:
  (these are not errors by default)
  - attempt to unlock an unlocked mutex
  - attempt by non-owner to unlock locked mutex
  - attempt by owner to lock already locked mutex
- there are no comparable errors for semaphores

# Mutex & Semaphore Differences (contd.)

Interaction with signals mechanism:

- `pthread_cond_signal()` is not **async-signal-safe**, so not safe to use in an *asynchronously invoked signal handler*
- thus condition variablse (with mutexes) are not suitable for releasing a waiting thread by "signaling" from a signal handler (invoked asynchronously)
- `sem_post()` is *async-signal-safe*, so POSIX semaphores can be used from within asynchronously invoked signal handlers
- CVs/mutexes can be used with signals if the signals are waited for *"synchronously"* with `sigwaitinfo()`

# Semaphore Semantics with Mutex+CV

As noted above, it is possible to duplicate the semantics of Dijkstra's general semaphores using Pthread mutexes + CVs.

Duplicating a (single) general semphore requires *three objects*:

1. a mutex
2. a condition variable
3. a shared integer variable (holds the semaphore value)

These objects might be initialized as follows:

```
//Globals:
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cv = PTHREAD_COND_INITIALIZER;
int semaphore_value = SEM_INIT_VALUE;
```

# Semaphore Semantics with Mutex+CV (contd.)

Increment/post/release step require 4 lines of code:

```
pthread_mutex_lock(&mutex);
semaphore_value++;
pthread_mutex_unlock(&mutex);
pthread_cond_signal(&cv);
```

Decrement/wait/acquire step requires 5 lines of code:

```
pthread_mutex_lock(&mutex);
while (semaphore_value == 0)
  pthread_cond_wait(&cv, &mutex);
semaphore_value--;
pthread_mutex_unlock(&mutex);
```

# Semaphore Semantics with Mutex+CV (contd.)

The same thing could be accomplished with POSIX semaphores.

**Unnamed semaphores** are used among threads.

Intialization

```
//Globals:
sem_t *sem;

//inside main thread:
sem_init(&sem,0,SEM_INIT_VALUE;
```

Increment/post/release step:

```
sem_post(&sem);
```

Decrement/wait/acquire step:

```
sem_wait(&sem);
```

# Joining with Terminated Threads

`pthread_join()` provides Pthreads with functionality similar to that provided by `wait()`/`waitpid()` for processes:
`int pthread_join(pthread_t threadid, void **retval)`

- causes calling thread to *block* until the specified (sibling) thread *terminates*
- `threadid` is the thread ID, as obtained from `pthread_create()` or `pthread_self()` (but not that from `gettid()`)
- unless `retval` is `NULL`, return value from joined thread (as returned by `pthread_exit()` is copied to location `*retval`
- returns 0 on success, else (positive) error code

This call can be used to make a thread wait to proceed beyond some point until another thread has completed its subtask.

# Joining with Terminated Threads (contd.)

A few details about `pthread_join()`:

- target thread must be **joinable** (else error)
- if target thread has already terminated, `pthread_join()` returns immediately
- if target thread is *cancelled*, `PTHREAD_CANCELED` is placed in `*retval`
- if multiple threads simultaneously try to join with the same thread, the results are *undefined*
- if the thread calling `pthread_join()` gets *canceled*, target thread remains *joinable*
- returns error if two threads try to join with each other
- there is no analog of `waitpid(-1, &status, 0)`, i.e., "join with any terminated thread"

# File Locks

**File locks** are useful with Pthreads just as with processes.

The three process-level calls can also be used with Pthreads:

- `fcntl()` – primary syscall, can lock regions or entire files
- `flock()` – not in SUS, can lock entire files
- `lockf()` – in SUS, in Linux just a simpler interface to `fcntl()`

In addition, there are a set of C library routines that can be used to lock *streams* (`FILE*`'s) in *Pthreads*:

- `void flockfile(FILE *filehandle)`
- `int ftrylockfile(FILE *filehandle)`
- `void funlockfile(FILE *filehandle)`

# Producer-Consumer Example #1

Multithreaded example using *bounded* queue and *mutexes*:

- producer run in main thread, consumer in second thread
- *global* record *queue*
- *fixed-size* records
- *bounded* queue size (e.g., implemented as circular array)
- producer adds single record whenever it accesses queue
- consumer removes single record whenever it accesses queue
- requires three (Dijkstra) *semaphores* for synchronization:
  - *binary semaphore* to enforce *mutual exclusion* accessing queue
  - *general semaphore* count of available/unread records in queue, to limit when consumer retrieves records
  - *general semaphore* number of free record slots in queue, to limit when producer stores new records
- Dijkstra semaphores implemented as follows:
  - binary semaphore: *mutex*
  - general semaphore count of records: *global* (`int`) + *mutex* + *condition variable*
  - general semaphore number of free slots: *global* (`int`) + *mutex* + *condition variable*

# Producer-Consumer Example #1 (contd.)

Global objects, shared among threads:

```
//Queue for records:
queue_t *queue;

//Global mutexes, etc:
pthread_mutex_t q_op_mtx = PTHREAD_MUTEX_INITIALIZER;

pthread_mutex_t q_avail_mtx = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t q_avail_cv = PTHREAD_COND_INITIALIZER;
int q_avail_sem = 0;

pthread_mutex_t q_free_mtx = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t q_free_cv = PTHREAD_COND_INITIALIZER;
int q_free_sem = QUEUE_MAX;
```

# Producer-Consumer Example #1 (contd.)

```
int main()
{
  //Initialize (circular array) queue:
  queue =  queue_init();

  //Create thread to run consumer:
  pthread_t threadid;
  pthread_create(&threadid,NULL,consumer,NULL);

  //In main thread here:
  producer();

  //Should not get here, infinite P-C loops:
  pthread_join(threadid,NULL);
  return EXIT_FAILURE;
  }
```

# Producer-Consumer Example #1 (contd.)

```
void producer()
{
  record_t new_record;

  while(1) {
    //Wait for free slot in queue and decrement:
    pthread_mutex_lock(&q_free_mtx);
    while (q_free_sem == 0)
      pthread_cond_wait(&q_free_cv, &q_free_mtx);
    q_free_sem--;
    pthread_mutex_unlock(&q_free_mtx);

    pthread_mutex_lock(&q_op_mtx);  //acquire lock on queue
    new_record = ...create new record...
    queue_enqueue(queue, new_record);  //store new record in queue
    pthread_mutex_unlock(&q_op_mtx);  //release lock on queue

    //Increment avail count and signal:
    pthread_mutex_lock(&q_avail_mtx);
    q_avail_sem++;
    pthread_mutex_unlock(&q_avail_mtx);
    pthread_cond_signal(&q_avail_cv);
  }

  return;
}
```

# Producer-Consumer Example #1 (contd.)

```
void *consumer(void *ignore)
{
  record_t next_record;

  while(1) {
    //Wait for non-zero avail count and decrement:
    pthread_mutex_lock(&q_avail_mtx);
    while (q_avail_sem == 0)
      pthread_cond_wait(&q_avail_cv, &q_avail_mtx);
    q_avail_sem--;
    pthread_mutex_unlock(&q_avail_mtx);

    pthread_mutex_lock(&q_op_mtx);  //acquire lock on queue
    queue_dequeue(queue, &next_record);  //retrieve next record in the queue
    pthread_mutex_unlock(&q_op_mtx);  //release lock on queue

    //Increment free count and signal:
    pthread_mutex_lock(&q_free_mtx);
    q_free_sem++;
    pthread_mutex_unlock(&q_free_mtx);
    pthread_cond_signal(&q_free_cv);

    ...process next_record...
  }

  return NULL;
}
```

# Producer-Consumer Example #2

Multithreaded example using *bounded* queue and *semaphores*:

- Same setup as example #1, except for synchronization mechanisms.
- Dijkstra semaphores implemented as follows:
  - *binary semaphore*: POSIX semaphore (unnamed)
  - count of available records: POSIX semaphore (unnamed)
  - number of free record slots: POSIX semaphore (unnamed)

# Producer-Consumer Example #2 (contd.)

Global objects, shared among threads:

```
//Queue for records:
queue_t *queue;

//Global (unnamed) semaphores:
sem_t q_op;
sem_t q_avail;
sem_t q_free;
```

# Producer-Consumer Example #2 (contd.)

```
int main()
{
  //Initialize (unnamed) semaphores:
  sem_init(&q_op,0,1);
  sem_init(&q_avail,0,0);
  sem_init(&q_free,0,QUEUE_MAX);

  //Initialize (circular array) queue:
  queue =  queue_init();

  //Create thread to run consumer:
  pthread_t threadid;
  pthread_create(&threadid,NULL,consumer,NULL);

  //In main thread here:
  producer();

  //Should not get here, infinite P-C loops:
  pthread_join(threadid,NULL);
  return EXIT_FAILURE;
  }
```

# Producer-Consumer Example #2 (contd.)

```
void producer()
{
  record_t new_record;

  while(1) {
    //Wait for free slot in queue and decrement:
    sem_wait(&q_free);

    sem_wait(&q_op);  //acquire lock on queue

    new_record = ...create new record...
    queue_enqueue(queue, new_record);  //store new record in queue

    sem_post(&q_op);  //release lock on queue

    //Increment avail count and signal:
    sem_post(&q_avail);
  }

  return;
}
```

# Producer-Consumer Example #2 (contd.)

```
void *consumer(void *ignore)
{
  record_t next_record;

  while(1) {
    //Wait for non-zero avail count and decrement:
    sem_wait(&q_avail);

    sem_wait(&q_op);  //acquire lock on queue

    queue_dequeue(queue, &next_record);  //retrieve next record in the queue

    sem_post(&q_op);  //release lock on queue

    //Increment free count and signal:
    sem_post(&q_free);

    ...process next_record...
  }

  return NULL;
}
```

# Running the P-C Examples

Running the P-C example code requires the following:

- define the record type `record_t`
- include the **circular array queue** code from the **Process Synchronization lectures**
- replace "create new record" with code to create a record consistent with `record_t`
- replace "process next record" with code to do something wth a record (e.g., print it)

# Running the P-C Examples (contd.)

For example, we can create records that are strings with the format "`Message #%03d`" by doing:

- add before queue definition:
  ```
  typedef char record_t[13];
  ```
- add message counting to `producer()` and `consumer()`
- make "create new record":
  ```
  snprintf(new_record,13,"Message #%03d",rec_cnt);
  ```
- make "process next record":
  ```
  printf("next_record: %s\n",next_record);
  ```