SUNG KYUN KWAN UNIVERSITY

Pthread

Prof. Jinkyu Jeong (jinkyu@skku.edu)
TA – Gyusun Lee (gyusun.lee@csi.skku.edu)
TA – Jiwon Woo (jiwon.woo@csi.skku.edu)
Computer Systems and Intelligence Laboratory (http://csi.skku.edu)
Sung Kyun Kwan University

Creating Concurrent Flows

- **Processes**
  - Kernel automatically interleaves multiple logical flows.
  - Each flow has its own private address space.
- **Threads**
  - Kernel automatically interleaves multiple logical flows.
  - Each flow shares the same address space.
  - Hybrid of processes and I/O multiplexing
- **I/O multiplexing with select()**
  - User manually interleaves multiple logical flows
  - Each flow shares the same address space
  - Popular for high-performance server designs.

Concurrent Programming

Thread-based

Traditional View

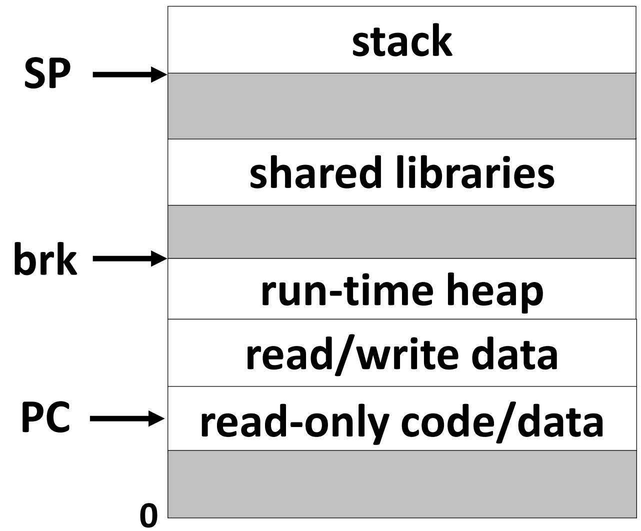
- Process = process context + address space

Process context

Program context:
- Data registers
- Condition codes
- Stack pointer (SP)
- Program counter (PC)

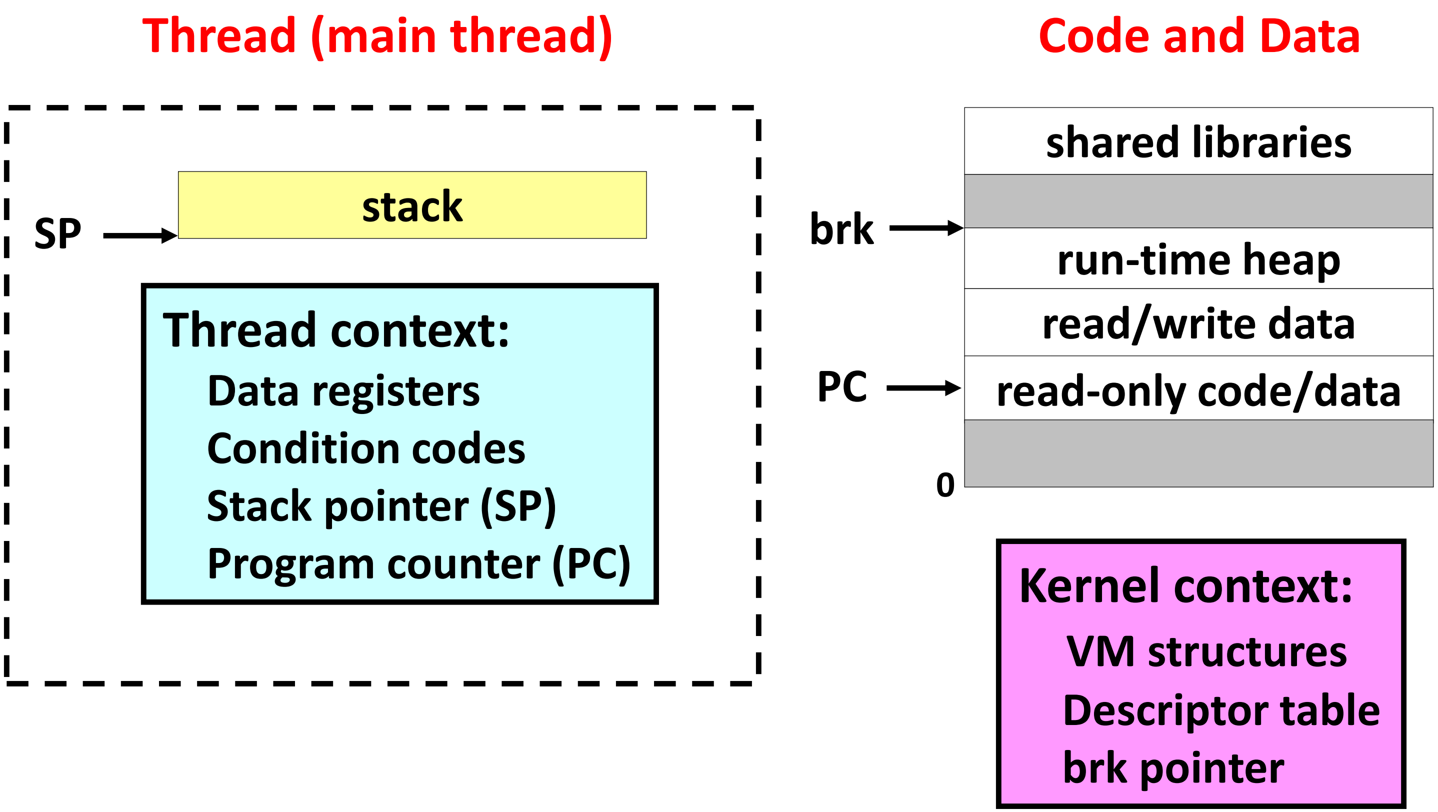
Kernel context:
- VM structures
- Descriptor table
- brk pointer

Code, data, and stack



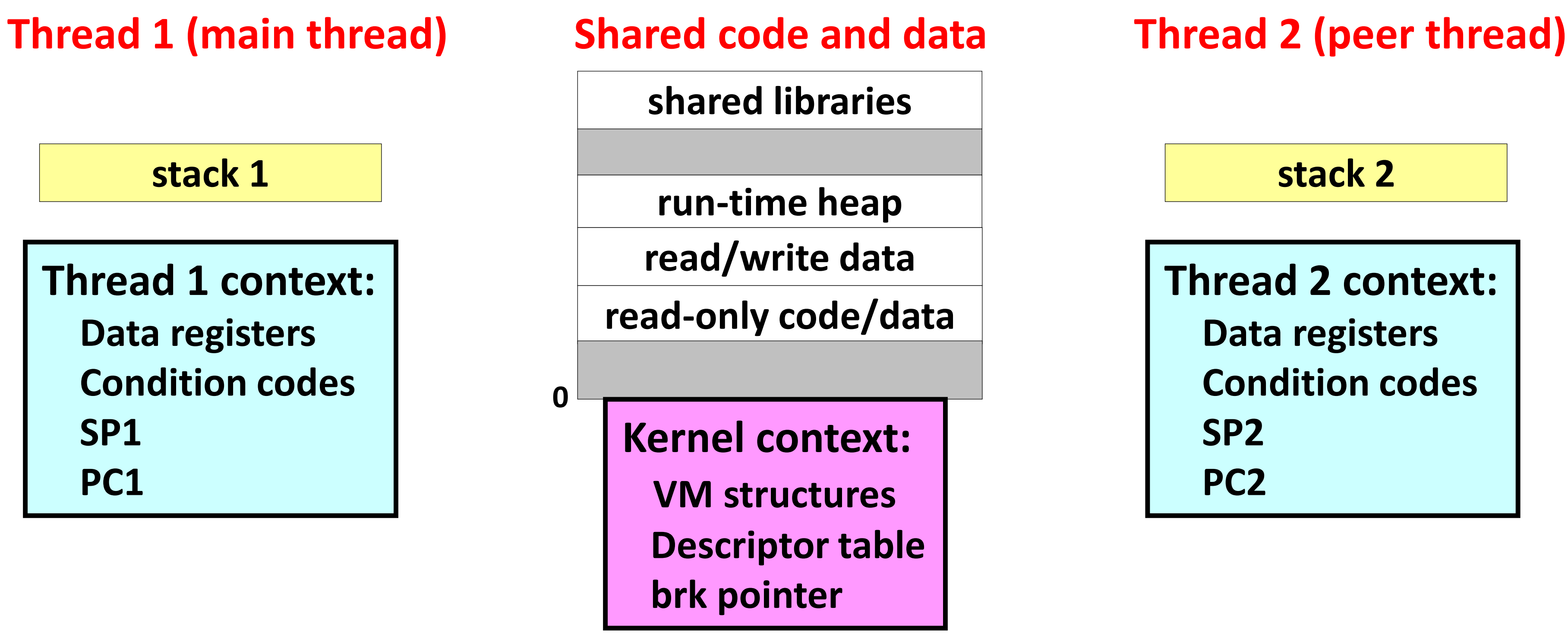
Alternate View

- Process = thread context + kernel context + address space



A Process with Multiple Threads

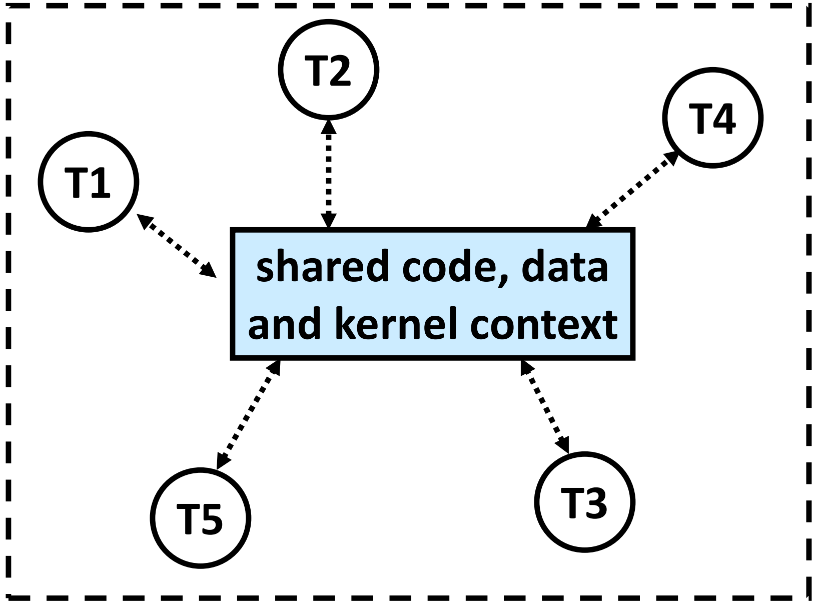
- **Multiple threads can be associated with a process.**
  - Each thread has its own logical control flow (sequence of PC values)
  - Each thread shares the same code, data, and kernel context
  - Each thread has its own thread id (TID)



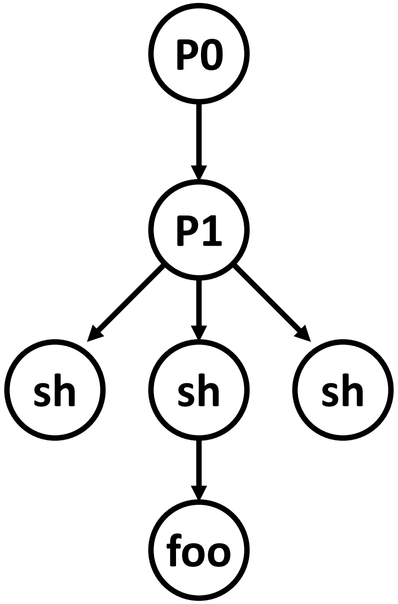
Logical View of Threads

- **Threads associated with a process form a pool of peers**
  - Unlike processes which form a tree hierarchy

Threads associated with process foo



Process hierarchy



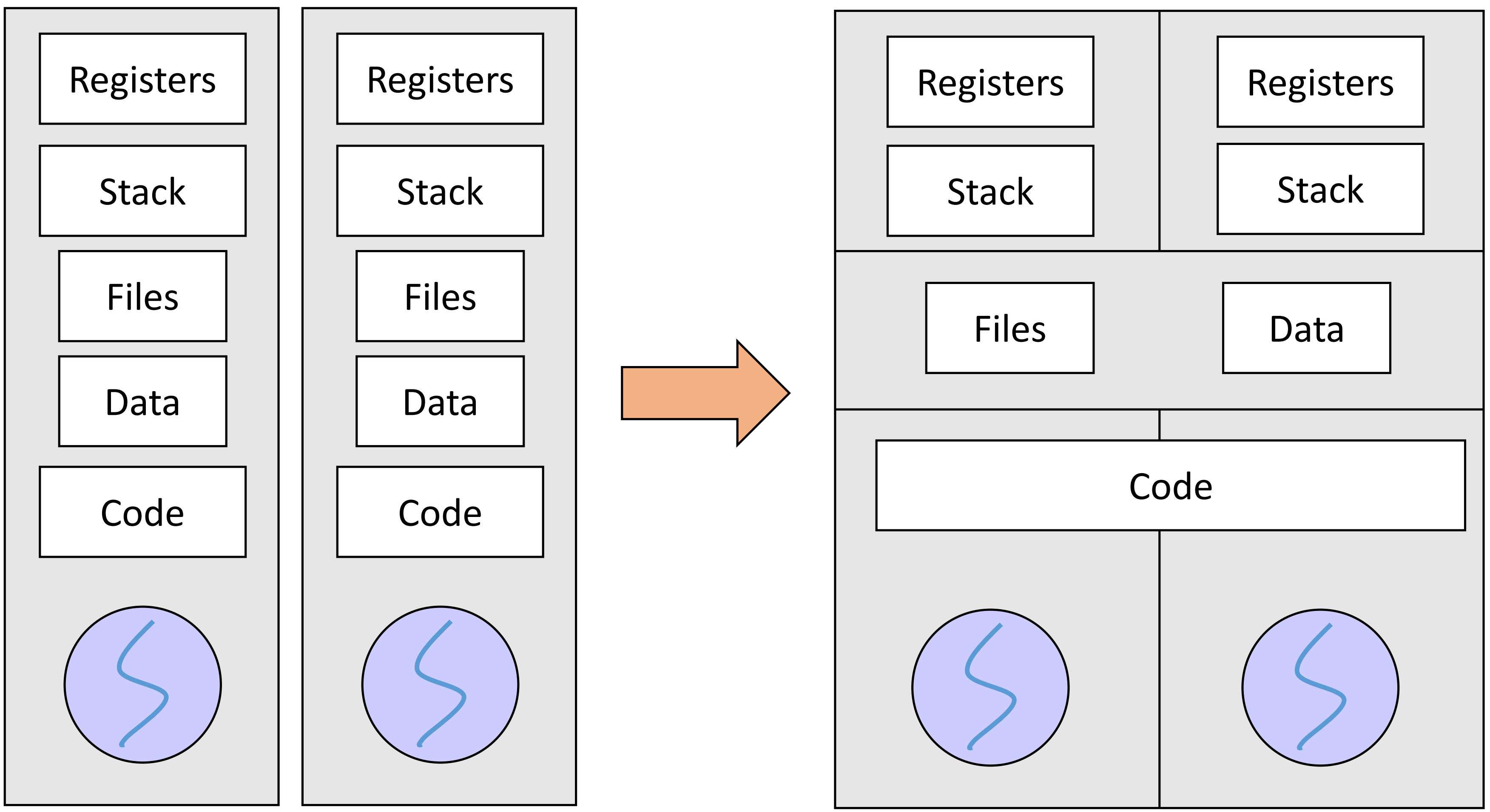
Threads vs. Processes

- **How threads and processes are similar**
  - Each has its own logical control flow.
  - Each can run concurrently.
  - Each is context switched.
- **How threads and processes are different**
  - Threads share code and data, processes (typically) do not.
  - Threads are somewhat less expensive than processes.
    - Linux 2.4 Kernel, 512MB RAM, 2 CPUs

      -> 1,811 forks()/second

      -> 227,611 threads/second (125x faster)

Threads vs. Processes



2 processes

2 threads, 1 process

The Pthread API

- **ANSI/IEEE POSIX1003.1-1995 Standard**
- **Thread management**
  - Work directly on threads – creating, terminating, joining, etc.
  - Include functions to set/query thread attributes.
- **Mutexes**
  - Provide for creating, destroying, locking and unlocking mutexes.
- **Conditional variables**
  - Include functions to create, destroy, wait and signal based upon specified variable values.

Pthreads Interface

- **POSIX Threads Interface**
  - Creating and reaping threads
    - **pthread_create()**
    - **pthread_join()**
  - Determining your thread ID
    - **pthread_self()**
  - Terminating threads
    - **pthread_cancel()**
    - **pthread_exit()**
    - **exit** (terminates all threads), **return** (terminates current thread)
  - Synchronizing access to shared variables
    - **pthread_mutex_init()**
    - **pthread_mutex_[un]lock()**
    - **pthread_cond_init()**
    - **pthread_cond_[timed]wait()**
    - **pthread_cond_signal()**, etc.

Example - process

Handling Signals (2)

- What is the output of the following program?

```c
pid_t pid;
int counter = 2;

void handler1(int sig) {
    counter = counter - 1;
    printf("%d", counter);
    fflush(stdout);
    exit(0);
}
int main() {
    signal(SIGUSR1, handler1);
    printf("%d", counter);
    fflush(stdout);

    if((pid = fork()) == 0) while(1);
    kill(pid, SIGUSR1);
    waitpid(-1, NULL, 0);
    counter = counter + 1;
    printf("%d", counter);
    exit(0);
}
```


5-signals

Example - thread

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/syscall.h>
```

```
int gv = 2;
void *thread(void *vargp);

int main()
{
    pthread_t tid;
    pthread_create(&tid, NULL, thread, NULL);
    pthread_join(tid, NULL);
    printf("[main    ] value:%2d | PID: %5d | TID: %lud\n”,
            gv, getpid(), (unsigned)pthread_self() );
     exit(0);
}

void *thread(void *vargp) /* Thread routine */
{
    printf(“Hello, world!\n”);
    printf(“[created] value:%2d | PID: %5d | TID: %lud\n",
            gv, getpid(), (unsigned)pthread_self() );
    return NULL;
}
```

Creating Threads (1)

- **int pthread_create (pthread_t *thread, pthread_attr_t *attr, void *(*start_routine)(void *), void *arg)**
  - **pthread_create()** returns the new thread ID via the **thread** argument.
    - The caller can use this thread ID to perform various operations on the thread.
  - The **attr** parameter is used to set thread attributes.
    - NULL for the default values.
  - The **start_routine** denotes the C routine that the thread will execute once it is created.
    - C routine that the thread will execute once it is created.
  - A single argument may be passed to **start_routine()** via arg.

Creating Threads (2)

- **Notes:**
  - Initially, **main()** comprises a single, default thread.
  - All other threads should must be explicitly created by the programmer.
  - Once created, threads are peers, and may create other threads.
  - The maximum number of threads that may be created by a process is implementation dependent.

Terminating Threads

- **void pthread_exit (void *retval)**
  - **pthread_exit()** terminates the execution of the calling thread.
    - Typically, this is called after a thread has completed its work and is no longer required to exist.
  - The **retval** argument is the return value of the thread.
    - It can be consulted from another thread using **pthread_join().**
  - It does not close files; any files opened inside the thread will remain open after the thread is terminated.

Cancelling Threads

- **int pthread_cancel (pthread_t thread)**
  - **pthread_cancel()** sends a cancellation request to the thread denoted by the **thread** argument.
  - Depending on its settings, the target thread can then either ignore request, honor it immediately, or defer it till it reaches a cancellation point.
    - pthread_setcancelstate(): PTHREAD_CANCEL_(ENABLE|DISABLE)
    - pthread_setcanceltype(): PTHREAD_CANCEL_(DEFERRED|ASYNCHRONOUS)
  - Threads are always created by **pthread_create()** with cancellation enabled and deferred.

Joining Threads

- **int pthread_join (pthread_t thread, void **retval)**
  - **pthread_join()** suspends the execution of the calling thread until the thread identified by **thread** terminates, either by calling **pthread_exit()** or by being cancelled.
  - The return value of **thread** is stored in the location pointed by **retval**.
  - It returns **PTHREAD_CANCELLED** if thread was cancelled.
  - It is impossible to join a detached thread.

Detaching Threads

- **int pthread_detach (pthread_t thread)**
  - **pthread_detach()** puts the thread in the detached state.
    - This guarantees that the memory resources consumed by **thread** will be freed immediately when thread terminates.
    - However, this prevents other threads from synchronizing on the termination of thread using **pthread_join()**.
  - A thread can be detached when it is created:

```
pthread_t tid;
pthread_attr_t attr;

pthread_attr_init (&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
pthread_create(&tid, &attr, start_routine, NULL);
pthread_attr_destroy (&attr);
```

Thread Identifiers

- **pthread_t pthread_self (void)**
  - **pthread_self()** returns the unique, system assigned thread ID of the calling thread.
- **int pthread_equal (pthread_t t1, pthread_t t2)**
  - **pthread_equal()** returns a non-zero value if **t1** and **t2** refer to the same thread.
  - Because thread IDs are opaque objects, the C language equivalence operator == should not be used to compare two thread IDs against each other.

Threads Synchronization

Example

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

int num;

void* inc (void* tid) {
   int iter = 10000;
   while(iter--) num++;
 }

void* dec (void* tid) {
   int iter = 10000;
   while(iter--) num--;
 }
```

```c
int main()
{
   pthread_t thread_inc, thread_dec;
   pthread_create(&thread_inc, NULL, &inc, NULL);
   pthread_create(&thread_dec,NULL, &dec, NULL);

   pthread_join(thread_inc,NULL);
   pthread_join(thread_dec,NULL);

   printf("%d\n",num);
   return 0;

}
```

Mutex (1)

- **Mutex is an abbrev. for “mutual exclusion”**
  - Primary means of implementing thread synchronization.
    - Protects shared data when multiple writes occurs.
  - A mutex variable acts like a “lock” protecting access to a shared resource.
    - Only one thread can lock (or own) a mutex variable at any given time.
    - Even if several threads try to lock a mutex, only one thread will be successful. Other threads are blocked until the owner releases the lock.
  - Mutex is used to prevent “race” conditions.
    - race condition: anomalous behavior due to unexpected critical dependence on the relative timing of events.

Mutex (2)

```
int deposit(int amount)
{
  int balance;

  balance = get_balance();
  balance += amount;
  put_balance(balance);
  return balance;
}
```

```
int withdraw(int amount)
{
  int balance;

  balance = get_balance();
  balance -= amount;
  put_balance(balance);
  return balance;
}
```

T1 executes deposit(100)

```
balance = get_balance();
balance += 100;



put_balance(balance);
```

T2 executes withdraw(300)

```
balance = get_balance();
balance -= 300;
put_balance(balance);
```

Creating/Destroying Mutexes

- **Static initialization**
  - `pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;`
- **Dynamic initialization**
  - `pthread_mutex_t m;`
    `pthread_mutex_init (&m, (pthread_mutexattr_t *)NULL);`
- **Destroying a mutex**
  - `pthread_mutex_destroy (&m);`
  - Destroys a mutex object, freeing the resources it might hold.

Using Mutexes (1)

- **`int pthread_mutex_lock (pthread_mutex_t *mutex)`**
  - Acquire a lock on the specified **`mutex`** variable.
  - If the **`mutex`** is already locked by another thread, block the calling thread until the **`mutex`** is unlocked.

- **`int pthread_mutex_unlock (pthread_mutex_t *mutex)`**
  - Unlock a **`mutex`** if called by the owning thread.

- **`int pthread_mutex_trylock (pthread_mutex_t *mutex)`**
  - Attempt to lock a **`mutex`**.
  - If the **`mutex`** is already locked, return immediately with a “busy” error code.

Using Mutexes (2)

```c
pthread_mutex_t m =
     PTHREAD_MUTEX_INITIALIZER;

int deposit(int amount)
{
  int balance;

  pthread_mutex_lock(&m);

  balance = get_balance();
  balance += amount;
  put_balance(balance);

  pthread_mutex_unlock(&m);

  return balance;
}
```

```c
int withdraw(int amount)
{
  int balance;

  pthread_mutex_lock(&m);

  balance = get_balance();
  balance -= amount;
  put_balance(balance);

  pthread_mutex_unlock(&m);

  return balance;
}
```

Condition Variables (1)

- **Another way for thread synchronization**
  - While mutexes implement synchronization by controlling thread access to data, condition variables allow threads to synchronize based upon the actual value of data.
  - Without condition variables, the programmer would need to have threads continually polling to check if the condition is met.
    - This can be very resource consuming since the thread would be continuously busy in this activity.
  - A condition variable is always used in conjunction with a mutex lock.

Condition Variables (2)

- **How condition variables work**
  - A thread locks a mutex associated with a condition variable.
  - The thread tests the condition to see if it can proceed.
  - If it can
    - Your thread does its work
    - Your thread unlocks the mutex
  - If it cannot
    - The thread sleeps. The mutex is automatically released.
    - Some other threads signals the condition variable.
    - Your thread wakes up from waiting with the mutex automatically locked, and it does its work.
    - Your thread releases the mutex when it’s done.

Creating/Destroying CV

- **Static initialization**
  - `pthread_cond_t cond = PTHREAD_COND_INITIALIZER;`
- **Dynamic initialization**
  - `pthread_cond_t cond;`
    `pthread_cond_init (&cond, (pthread_condattr_t *)NULL);`
- **Destroying a condition variable**
  - `pthread_cond_destroy (&cond);`
  - Destroys a condition variable, freeing the resources it might hold.

Using Condition Variables

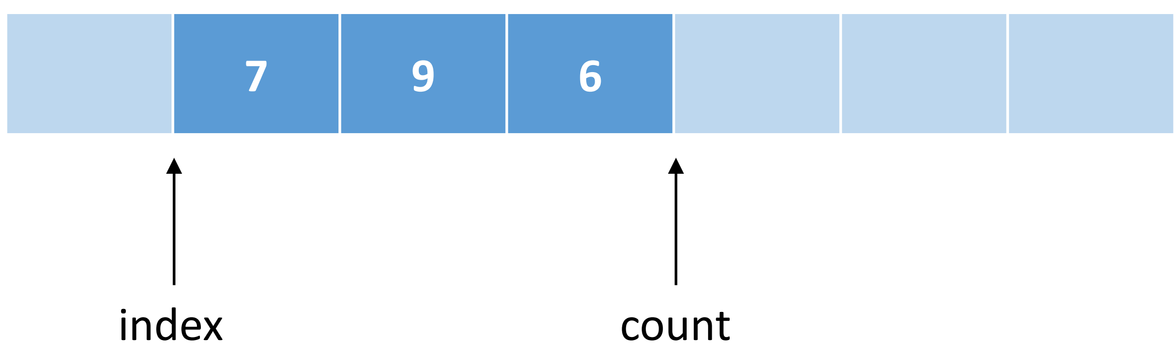
- `int pthread_cond_wait (pthread_cond_t *cond, pthread_mutex_t *mutex)`
  - Blocks the calling thread until the specified condition is signalled.
  - This should be called while mutex is locked, and it will automatically release the mutex while it waits.
- `int pthread_cond_signal (pthread_cond_t *cond)`
  - Signals another thread which is waiting on the condition variable.
  - Calling thread should have a lock.
- `int pthread_cond_broadcast(pthread_cond_t *cond)`
  - Used if more than one thread is in a blocking wait state.

Exercise

Producer-Consumer

- Bounded buffer: size N
- Producer threads writes data to buffer
  - Should not write more than N items
- Consumer threads reads data from buffer
  - Should not try to consume if there is no data

- Buffer is circular



Producer-Consumer (1)

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

#define QSIZE            5
#define LOOP             30

typedef struct {
   int data[QSIZE];
   int index;
   int count;
   pthread_mutex_t lock;
   pthread_cond_t notfull;
   pthread_cond_t notempty;
} queue_t;

void *produce (void *args);
void *consume (void *args);
void put_data (queue_t *q, int d);
int  get_data (queue_t *q);
```

Producer-Consumer (2)

```
int main ()
{
    queue_t *q;
    pthread_t producer, consumer;

    q = qinit();

    pthread_create(&producer, NULL, produce, (void *)q);
    pthread_create(&consumer, NULL, consume, (void *)q);

    pthread_join (producer, NULL);
    pthread_join (consumer, NULL);

    qdelete(q);
}
```

Producer-Consumer (3)

```
queue_t *qinit()
{
   queue_t *q;

   q = (queue_t *) malloc(sizeof(queue_t));
   q->index = q->count = 0;
   pthread_mutex_init(&q->lock, NULL);
   pthread_cond_init(&q->notfull, NULL);
   pthread_cond_init(&q->notempty, NULL);

   return q;
}

void qdelete(queue_t *q)
{
   pthread_mutex_destroy(&q->lock);
   pthread_cond_destroy(&q->notfull);
   pthread_cond_destroy(&q->notempty);
   free(q);
}
```

Producer-Consumer (4)

```
void *produce(void *args)
{
    int i, d;
    queue_t *q = (queue_t *)args;
    for (i = 0; i < LOOP; i++) {
        d = rand() % 10;
        put_data(q, d);
        printf("put data %d to queue\n", d);
    }
    pthread_exit(NULL);
}
void *consume(void *args)
{
    int i, d;
    queue_t *q = (queue_t *)args;
    for (i = 0; i < LOOP; i++) {
        d = get_data(q);
        printf("got data %d from queue\n", d);
    }
    pthread_exit(NULL);
}
```

Exercise

- **`void put_data(queue_t *q, int d)`**
  - Put data at the end of the queue
  - If the queue is full, wait for the data to be consumed.
- **`int get_data(queue_t *q)`**
  - Get data in front of the queue
  - If the queue is empty, wait for the data to be produced.