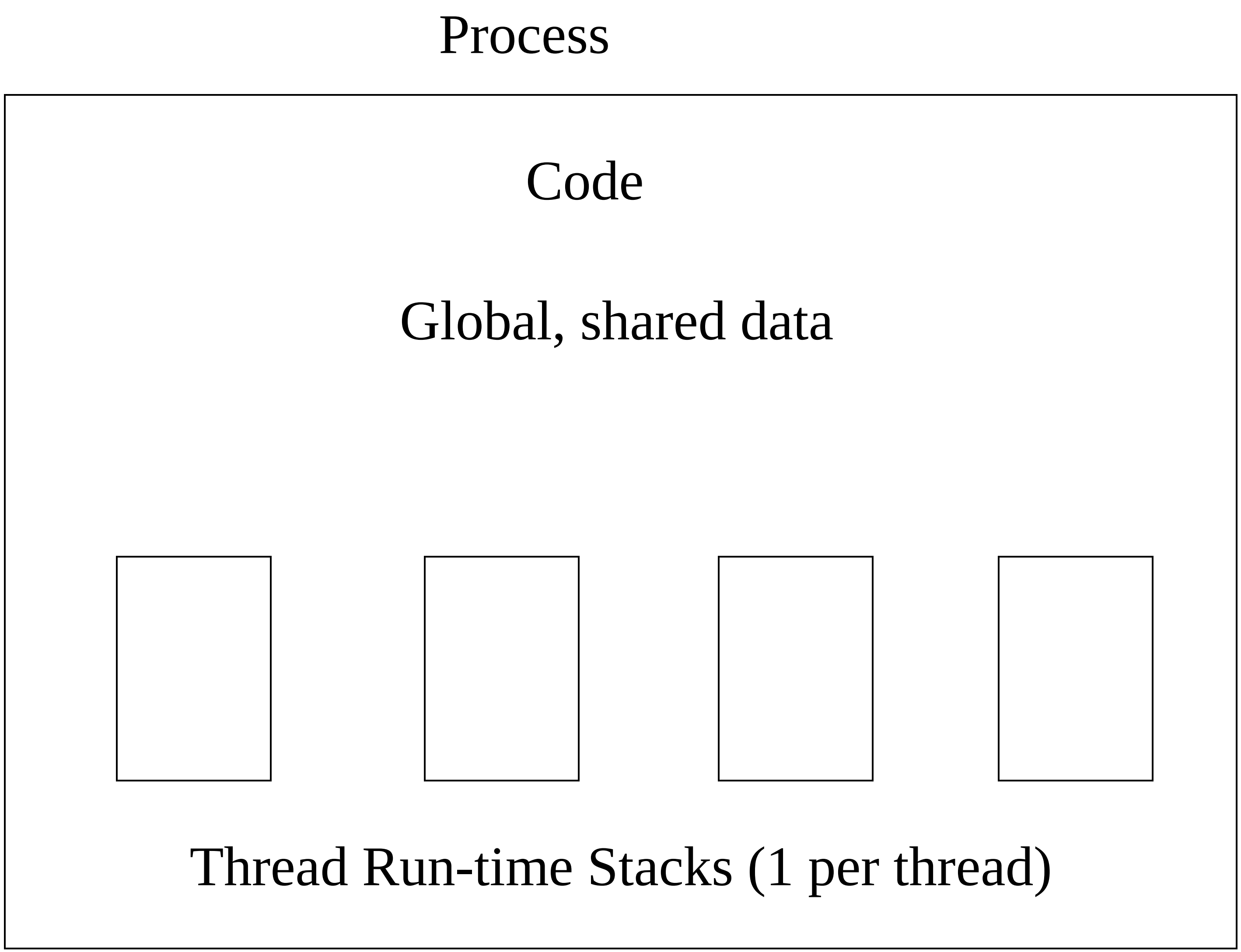
Pthreads

Threads

- Threads execute within Processes
- They share the address space of the process they execute in
- It is a schedulable entity
- Systems with no threads support are “single-threaded” systems

Threads in a Process



Process Attributes

- Process ID, process group ID, user ID, and group ID
- Environment
- Working directory.

Process Resources

- Address Space
- File descriptors
- Signal actions
- Shared libraries
- Inter-process communication tools (such as message queues, pipes, semaphores, or shared memory).

Thread Properties

- It is a schedulable entity, so properties reflect this:
  - Stack
  - Scheduling properties (such as policy or priority)
  - Set of pending and blocked signals
  - Some thread-specific data.

Threads Implementation

- Kernel threads
  - These are the entities supported by the kernel
  - They are scheduled to run on the CPUs of the system
- User Threads
  - Used by programmers to handle multiple flows in a program
  - A threads library provides an API to support them
  - Posix Threads (Pthreads)

Advantages of Threads

- Context Switching time is less
- Thread creation is easy
- Synchronization among threads in a process is cheap
- Threads can communicate through shared memory
- But, protection is not there.
  - One thread can overwrite another thread’s data
  - This makes debugging difficult

Pthreads

- Posix Standard Threads library
- Implemented in a wide number of environments
  - Linux
  - Solaris
  - AIX
  - HP-UX
  - VxWorks

Pthreads Primitives

- Thread Creation and Close
- Thread Synchronization
- Thread Scheduling Attributes
- Miscellaneous Functions

Example Program includes

```
#include <pthread.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <ctype.h>
```

- To compile program hello.c
- Cc –o hello hello.c -lpthread

Example Program

```
void print_message_function( void *ptr );
main() {
pthread_t thread1, thread2;
char *message1 = "Hello";
char *message2 = "World";
pthread_create( &thread1,
  pthread_attr_default,
(void*)&print_message_function, (void*)
  message1);
pthread_create(&thread2,
  pthread_attr_default,
(void*)&print_message_function, (void*)
  message2);
exit(0);
}
```

```
void print_message_function(
  void *ptr ) {
char *message;
message = (char *) ptr;
printf("%s ", message);
}
```

Race Conditions

- If “main” calls exit before the threads run, none of the threads will execute
- The order of “Hello” and “World” may be reversed

Using pthread_join

```
void print_message_function( void *ptr );
void main() {
pthread_t thread1, thread2;
char *message1 = "Hello";
char *message2 = "World";
pthread_create( &thread1, NULL,
(void *) &print_message_function, (void *) message1);
pthread_create(&thread2, NULL,
(void *) &print_message_function, (void *) message2);
pthread_join(thread1, NULL);
pthread_join (thread2, NULL);
exit(0);     }
void print_message_function( void *ptr ) {
char *message;
message = (char *) ptr;
printf("%s\n", message);
pthread_exit(0);   }
```

Race Conditions

THREAD 1	THREAD 2
a = data;	b = data;
a++;	b--;
data = a;	data = b;

- Both Thread1 and Thread2 may read the same value of “data”
- If initial value of “data” is 5, final value after 1 execution each of Thread1 and Thread2 should be 5.
- But it may be 4, or 6.

Critical Sections

```
pthread_mutex_init(&m1,
  pthread_mutexattr_default);
```

THREAD 1 THREAD 2

```
pthread_mutex_lock( &m1 );

                              pthread_mutex_lock( &m1 );
    a = data;                              b = data;
    a++;                                   b--;
    data = a;                              data = b;
pthread_mutex_unlock( &m1 );

                              pthread_mutex_unlock( &m1 );
```

Pthread_mutex_lock

- A call to pthread_mutex_lock will “lock” the mutex variable m1. If it is already locked, then the thread will “block”.
- A call to pthread_mutex_unlock will “unlock’ the mutex variable m1. If other threads are blocked on m1, one of them is “unblocked”.

Condition Variables

- Condition variables allow threads to wait until some event or condition has occurred. Typically, a program will use three objects:
  - A boolean variable, indicating whether the condition is met
  - A mutex to serialize the access to the boolean variable
  - A condition variable to wait for the condition.

Pthreads Cond_wait

```
pthread_mutex_lock(&condition_lock);
 while (condition_predicate == 0)
    pthread_cond_wait(&condition_variable,
    &condition_lock);
...
pthread_mutex_unlock(&condition_lock);
```

Release condition_lock on waiting in cond_wait

Re-acquire condition_lock on waking up

Pthread_cond_signal (&condition_variable)

To wake up the waiting thread

Producer Consumer with Cond_waits

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#define BUFFER_SIZE 16

/* Circular buffer of integers. */

struct prodcons {
  int buffer[BUFFER_SIZE];        /* the actual
  data */
  pthread_mutex_t lock;           /* mutex ensuring
  exclusive access to buffer */
  int readpos, writepos;          /* positions for
  reading and writing */
  pthread_cond_t notempty;        /* signaled when
  buffer is not empty */
  pthread_cond_t notfull;         /* signaled when
  buffer is not full */
};
```

```c
/* Initialize a buffer */

void init(struct prodcons * b)
{
  pthread_mutex_init(&b->lock, NULL);
  pthread_cond_init(&b->notempty, NULL);
  pthread_cond_init(&b->notfull, NULL);
  b->readpos = 0;
  b->writepos = 0;
}
```

```
/* Store an integer in the buffer */

void put(struct prodcons * b, int data)
{
  pthread_mutex_lock(&b->lock);
  /* Wait until buffer is not full */
  while ((b->writepos + 1) % BUFFER_SIZE == b->readpos)
   {
    pthread_cond_wait(&b->notfull, &b->lock);
    /* pthread_cond_wait reacquired b->lock before
   returning */
   }
  /* Write the data and advance write pointer */
  b->buffer[b->writepos] = data;
  b->writepos++;
  if (b->writepos >= BUFFER_SIZE) b->writepos = 0;
  /* Signal that the buffer is now not empty */
  pthread_cond_signal(&b->notempty);
  pthread_mutex_unlock(&b->lock);
}
```

```
/* Read and remove an integer from the buffer */

int get(struct prodcons * b)
{
  int data;
  pthread_mutex_lock(&b->lock);
  /* Wait until buffer is not empty */
  while (b->writepos == b->readpos) {
    pthread_cond_wait(&b->notempty, &b->lock);
  }
  /* Read the data and advance read pointer */
  data = b->buffer[b->readpos];
  b->readpos++;
  if (b->readpos >= BUFFER_SIZE) b->readpos = 0;
  /* Signal that the buffer is now not full */
  pthread_cond_signal(&b->notfull);
  pthread_mutex_unlock(&b->lock);
  return data;
}
```

```
/* A test program: one thread inserts integers from 1 to 10000,
   the other reads them and prints them. */

#define OVER (-1)

struct prodcons buffer;

void * producer(void * data)
{
  int n;
  for (n = 0; n < 10000; n++) {
    printf("%d --->\n", n);
    put(&buffer, n);
  }
  put(&buffer, OVER);
  return NULL;
}
```

```
void * consumer(void * data)
{
  int d;
  while (1) {
    d = get(&buffer);
    if (d == OVER) break;
    printf("---> %d\n", d);
  }
  return NULL;
}
int main(void)
{
  pthread_t th_a, th_b;
  void * retval;
  init(&buffer);
  /* Create the threads */
  pthread_create(&th_a, NULL, producer, 0);
  pthread_create(&th_b, NULL, consumer, 0);
  /* Wait until producer and consumer finish. */
  pthread_join(th_a, &retval);
  pthread_join(th_b, &retval);
  return 0; }
```

Other Calls

- The **pthread_cond_broadcast** subroutine wakes up every thread that is currently blocked on the specified condition. However, a thread can start waiting on the same condition just after the call to the subroutine returns.
- The **pthread_cond_timedwait** subroutine blocks the thread only for a given period of time. This subroutine has an extra parameter, *timeout*, specifying an absolute date where the sleep must end.

Other Calls (contd)

- Once the condition variable is no longer needed, it should be destroyed by calling the **pthread_destroy** subroutine. This subroutine may reclaim any storage allocated by the **pthread_cond_init** subroutine. After having destroyed a condition variable, the same **pthread_cond_t** variable can be reused for creating another condition.
- while (pthread_cond_destroy(&cond) == EBUSY) { pthread_cond_broadcast(&cond);
- pthread_yield(); }
- The **pthread_yield** subroutine gives the opportunity to another thread to be scheduled, one of the awoken threads for example.

Mutex Calls

- **pthread_mutex_destroy** Deletes a mutex.
- **pthread_mutex_init** Initializes a mutex and sets its attributes.
- **PTHREAD_MUTEX_INITIALIZER** Initializes a static mutex with default attributes.
- **pthread_mutex_lock** or **pthread_mutex_trylock**

  Locks a mutex.
- **pthread_mutex_unlock** Unlocks a mutex.
- **pthread_mutexattr_destroy** Deletes a mutex attributes object.
- **pthread_mutexattr_init** Creates a mutex attributes object and initializes it with default values.

Readers Writers Problem

- Shared Data
- Some threads read the data, others write into it
- At a time, multiple readers may be reading the data
- At most one writer can be writing into the data
- While the writer writes, no readers may read.

```c
#define MAXCOUNT 5
typedef struct {
       pthread_mutex_t *mut;
       int writers;
       int readers;
       int waiting;
       pthread_cond_t *writeOK, *readOK;
} rwl;
rwl *initlock (void);
void readlock (rwl *lock, int d);
void writelock (rwl *lock, int d);
void readunlock (rwl *lock);
void writeunlock (rwl *lock);
void deletelock (rwl *lock);
typedef struct {
       rwl *lock;
       int id;
       long delay;
} rwargs;
rwargs *newRWargs (rwl *l, int i, long d);
void *reader (void *args);
void *writer (void *args);
static int data = 1;
```

```
int main (){
	pthread_t r1, r2, r3, r4, w1;
	rwargs *a1, *a2, *a3, *a4, *a5;
	rwl *lock;
	lock = initlock ();
	a1 = newRWargs (lock, 1, WRITER1);
	pthread_create (&w1, NULL, writer, a1);
	a2 = newRWargs (lock, 1, READER1);
	pthread_create (&r1, NULL, reader, a2);
	a3 = newRWargs (lock, 2, READER2);
	pthread_create (&r2, NULL, reader, a3);
	a4 = newRWargs (lock, 3, READER3);
	pthread_create (&r3, NULL, reader, a4);
	a5 = newRWargs (lock, 4, READER4);
	pthread_create (&r4, NULL, reader, a5);
	pthread_join (w1, NULL);pthread_join (r1, NULL);
	pthread_join (r2, NULL);
	pthread_join (r3, NULL);nthread_join (r4, NULL);
	free (a1); free (a2); free (a3);
	free (a4); free (a5);
	return 0;}
```

```
rwargs *newRWargs (rwl *l, int i, long d)
{
        rwargs *args;

        args = (rwargs *)malloc (sizeof (rwargs));
        if (args == NULL) return (NULL);
        args->lock = l; args->id = i; args->delay = d;
        return (args);
}
```

```
void *reader (void *args)
{
        rwargs *a;
        int d;

        a = (rwargs *)args;

        do {
                readlock (a->lock, a->id);
                d = data;
                usleep (a->delay);
                readunlock (a->lock);
                printf ("Reader %d : Data = %d\n", a->id, d);
                usleep (a->delay);
        } while (d != 0);
        printf ("Reader %d: Finished.\n", a->id);

        return (NULL);
}
```

```
void *writer (void *args)
{
        rwargs *a;
        int i;

        a = (rwargs *)args;

        for (i = 2; i < MAXCOUNT; i++) {
                writelock (a->lock, a->id);
                data = i;
                usleep (a->delay);
                writeunlock (a->lock);
                printf ("Writer %d: Wrote %d\n", a->id, i);
                usleep (a->delay);
        }
        printf ("Writer %d: Finishing...\n", a->id);
        writelock (a->lock, a->id);
        data = 0;
        writeunlock (a->lock);
        printf ("Writer %d: Finished.\n", a->id);
        return (NULL);
}
```

```
rwl *initlock (void){
 rwl *lock;
 lock = (rwl *)malloc (sizeof (rwl));
 if (lock == NULL) return (NULL);
 lock->mut = (pthread_mutex_t *) malloc (sizeof
(pthread_mutex_t));
 if (lock->mut == NULL) { free (lock); return (NULL); }
 lock->writeOK = (pthread_cond_t *) malloc (sizeof
(pthread_cond_t));
 if (lock->writeOK == NULL) { free (lock->mut); free (lock);
   return (NULL); }
 lock->readOK = (pthread_cond_t *) malloc (sizeof
(pthread_cond_t));
 if (lock->writeOK == NULL) { free (lock->mut); free (lock-
>writeOK);
   free (lock); return (NULL); }
       pthread_mutex_init (lock->mut, NULL);
       pthread_cond_init (lock->writeOK, NULL);
       pthread_cond_init (lock->readOK, NULL);
       lock->readers = 0;
       lock->writers = 0;
       lock->waiting = 0;
       return (lock);
}
```

```
void readlock (rwl *lock, int d)
{
	pthread_mutex_lock (lock->mut);
	if (lock->writers || lock->waiting) {
		do {
			printf ("reader %d blocked.\n", d);
			pthread_cond_wait (lock->readOK, lock->mut);
			printf ("reader %d unblocked.\n", d);
		} while (lock->writers);
	}
	lock->readers++;
	pthread_mutex_unlock (lock->mut);

	return;
}
```

```
void writelock (rwl *lock, int d)
{
        pthread_mutex_lock (lock->mut);
        lock->waiting++;
        while (lock->readers || lock->writers) {
                printf ("writer %d blocked.\n", d);
                pthread_cond_wait (lock->writeOK, lock->mut);
                printf ("writer %d unblocked.\n", d);
        }
        lock->waiting--;
        lock->writers++;
        pthread_mutex_unlock (lock->mut);

        return;
}
```

```c
void readunlock (rwl *lock)
{
        pthread_mutex_lock (lock->mut);
        lock->readers--;
        pthread_cond_signal (lock->writeOK);
        pthread_mutex_unlock (lock->mut);
}

void writeunlock (rwl *lock)
{
        pthread_mutex_lock (lock->mut);
        lock->writers--;
        pthread_cond_broadcast (lock->readOK);
        pthread_mutex_unlock (lock->mut);
}
```

```
void deletelock (rwl *lock)
{
        pthread_mutex_destroy (lock->mut);
        pthread_cond_destroy (lock->readOK);
        pthread_cond_destroy (lock->writeOK);
        free (lock);

        return;
}
```

Lab 4

1. Implement the Producer Consumer Code (after correcting errors, if any) on the AWS server and run it successfully.
2. Implement the Readers – Writers code (after correcting errors, if any) on the AWS server and run it successfully.
3. The Readers – Writers code must be commented: There has to be a comment for each data structure that is defined. There has to be at least one comment for each function stating what the function does. If necessary, put more comments.
4. Suppose we are not allowed to use “cond_broadcast”. Change the program so that you do not use “cond-broadcast” and is still works as before.
5. What may happen if we remove the “usleep” calls in the programme?
6. Can any reader or the writer starve in this implementation? Explain.
7. DUE DATE – 24 / 25 February 2021.