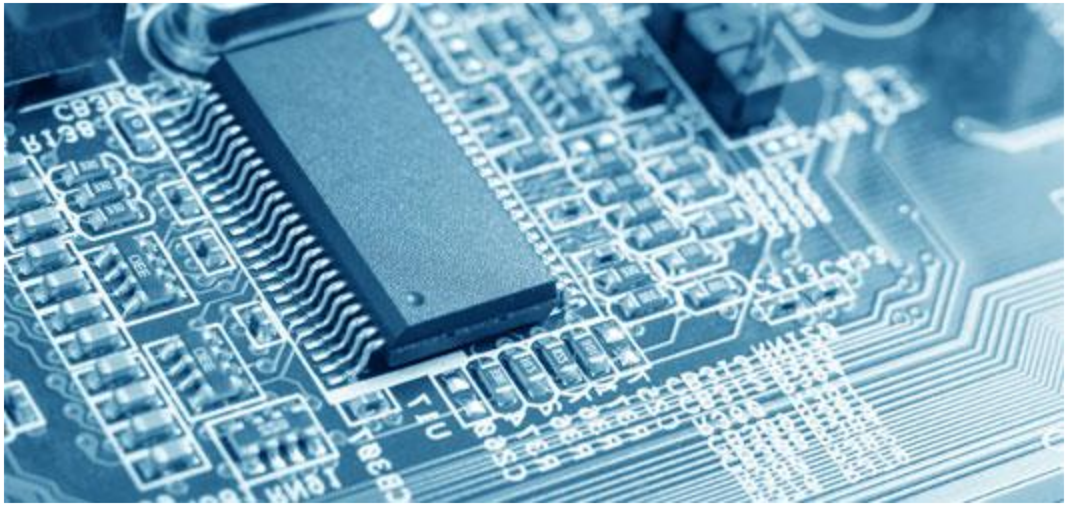
NAZARBAYEV UNIVERSITY
SCHOOL OF SCIENCE AND TECHNOLOGY



ROBT305 - Embedded Systems

Lecture 4 – POSIX Pthreads Library and Intertask Communication and Process Synchronization: Mutexes

1-3 September, 2015

Course Logistics

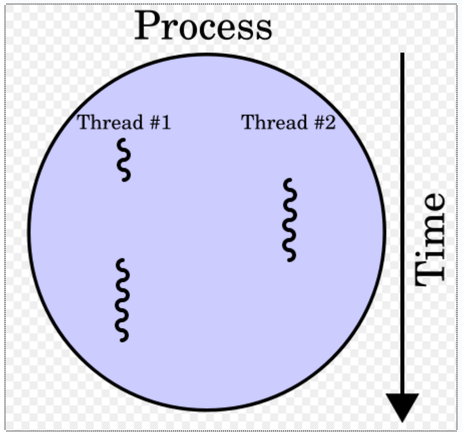
Reading Assignment: **Chapters 3, 4, 6** of the of the Operating Systems Concept textbook (relevant material only)
Chapter 3 of the Real-Time Systems Design and Analysis textbook (relevant material only)
Homework Assignment #1 is out in Moodle and due to end of 13 September (Sunday)
Quiz #1 is on 8 September – Process, Threads, Linux

Process vs. Threads

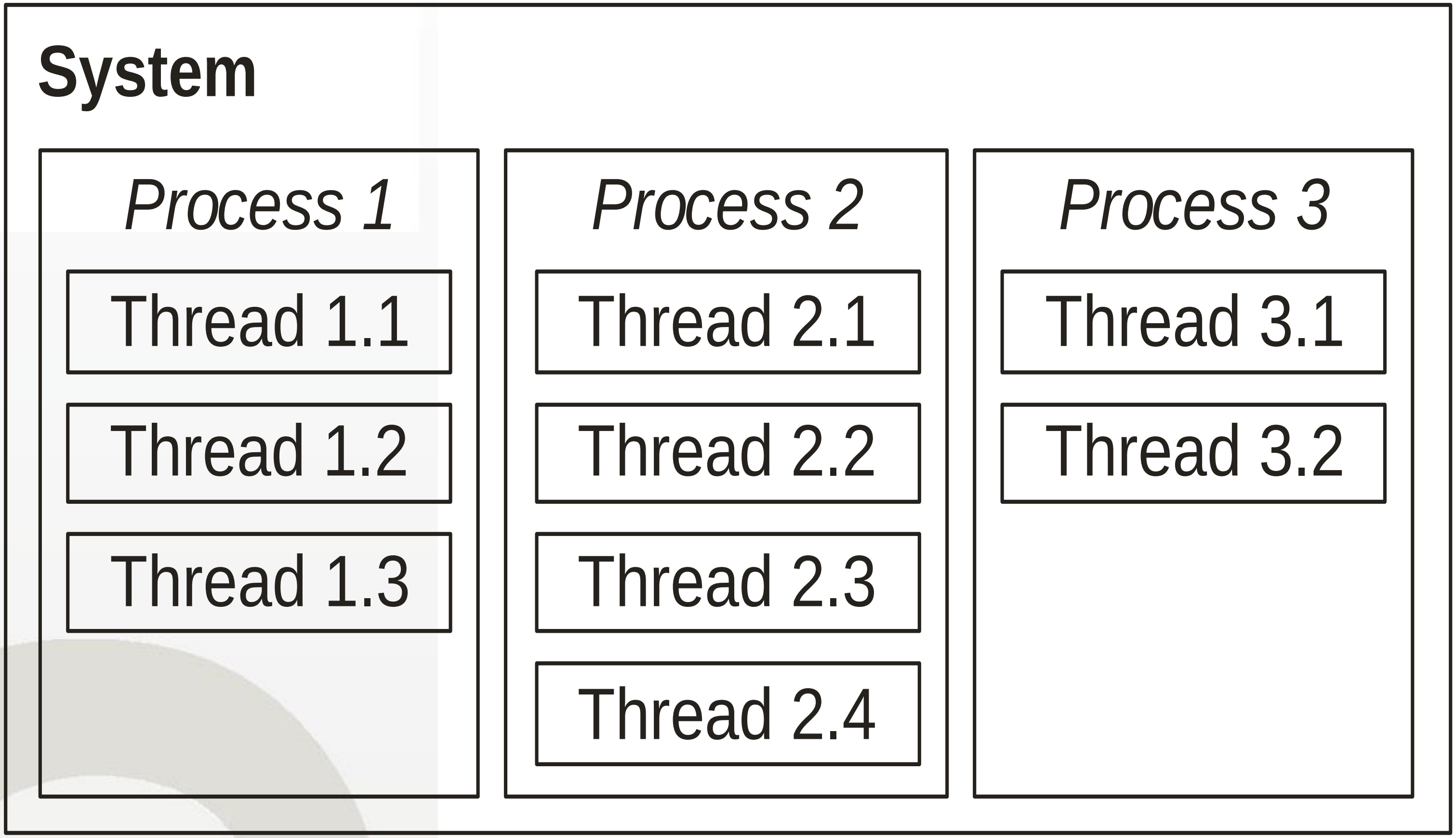
A process (synonymously called "task") is

- An abstraction of a running program
- The logical unit of work schedulable by the OS

A thread is a lightweight process within a regular process. It has access to the same memory space, and the context switching from one thread to another will be shorter than for process to process



Processes, and Multiple Threads



POSIX

- **POSIX** (or "Portable Operating System Interface“ is the collective name of a family of related standards specified by the IEEE to define the Application Programming Interface (API) for software compatible with variants of the Unix operating system.
- Originally, the name stood for IEEE Std 1003.1-1988. The family of POSIX standards is formally designated as **IEEE 1003** and the international standard name is ISO/IEC 9945. The standards emerged from a project that began in 1985. The term *POSIX* was suggested by Richard Stallman in response to an IEEE request for a memorable name!
- So, POSIX is not an OS, but an API! The API include Real-Time Services, Threads interface, Real-Time extensions, etc.

Thread Libraries

- **Thread library** provides programmer with API for creating and managing threads

- Two primary ways of implementing
  - Library entirely in user space
  - Kernel-level library supported by the OS

- when compiling under gcc & GNU/Linux, remember the –lpthread option!

POSIX Threads Programming

- POSIX Pthreads library
- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

Thread Body

- a thread is identified by a C function, called body:

```
void *my_thread(void *arg)
{
  ...
}
```

- a thread starts with the first instruction of its body
- the threads ends when the body function ends
- it's not the only way a thread can finish

Thread Creation

- thread can be created using the primitive

```
int pthread_create( pthread_t *ID,
                    pthread_attr_t *attr,
                    void *(*body)(void *),
                    void * arg);
```

- pthread_t is the type that contains the thread ID
- pthread_attr_t is the type that contains the parameters of the thread
- arg is the argument passed to the thread body when it starts

Thread Termination

- a thread can terminate itself by calling
  void pthread_exit(void *retval);

- when the thread body ends after the last "}", pthread_exit() is called implicitly

- exception: when main() terminates, exit() is called implicitly

Thread IDs

- each thread has a unique ID
- the thread ID of the current thread can be obtained using

```
pthread_t pthread_self(void);
```

- two thread IDs can be compared using

```
int pthread_equal( pthread_t thread1,
                   pthread_t thread2 );
```

Joining a Thread

- a thread can wait the termination of another thread using

```
int pthread_join( pthread_t th,
                  void **thread_return);
```

- it gets the return value of the thread or PTHREAD_CANCELED if the thread has been killed

- by default, every task **must** be joined

- the join frees all the internal resources (stack, registers, and so on)

Pthreads Example

```
/*****************************************************************************
* DESCRIPTION: * A "hello world" Pthreads program. Demonstrates thread creation and * termination. *
******************************************************************************/
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 5

void *PrintHello(void *threadID)
{
   long tID;
   tID = (long) threadID;
   printf ("Hello World!  It's me, thread #%lD!\n", tID);
   pthread_exit(NULL);
}
```

Pthreads Example Cont.

```
int main (int argc, char *argv[])
{
    pthread_t  threads[NUM_THREADS];
    int rc;
    long t;
    for(t=0; t<NUM_THREADS; t++)
    {
        printf("In main: creating thread %ID\n", t);
        rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
        if (rc)
        {
            printf("ERROR; return code from pthread_create() is %d\n", rc);
            exit(-1);
         }
     } /* Last thing that main() should do */
     pthread_exit(NULL);
}
```

Pthreads Example

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

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
   pthread_t tid; /* the thread identifier */
   pthread_attr_t attr; /* set of thread attributes */

   if (argc != 2) {
      fprintf(stderr,"usage: a.out <integer value>\n");
      return -1;
   }
   if (atoi(argv[1]) < 0) {
      fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
      return -1;
   }
```

Pthreads Example (Cont.)

```
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* create the thread */
   pthread_create(&tid,&attr,runner,argv[1]);
   /* wait for the thread to exit */
   pthread_join(tid,NULL);

   printf("sum = %d\n",sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
   int i, upper = atoi(param);
   sum = 0;

   for (i = 1; i <= upper; i++)
      sum += i;

   pthread_exit(0);
}
```

Pthreads Code for Joining 10 Threads

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
   pthread_join(workers[i], NULL);
```

Thread Cancellation

- Terminating a thread before it has finished
- Thread to be canceled is **target thread**
- Pthread code to create and cancel a thread:

```
pthread_t tid;

/* create the thread */
pthread_create(&tid, 0, worker, NULL);

. . .

/* cancel the thread */
pthread_cancel(tid);
```

POSIX Threads Programming Practice

- Have a look at the beginning of this tutorial from Lawrence Livermore National Laboratory https://computing.llnl.gov/tutorials/pthreads

- A PDF version of the tutorial is on the Moodle web page

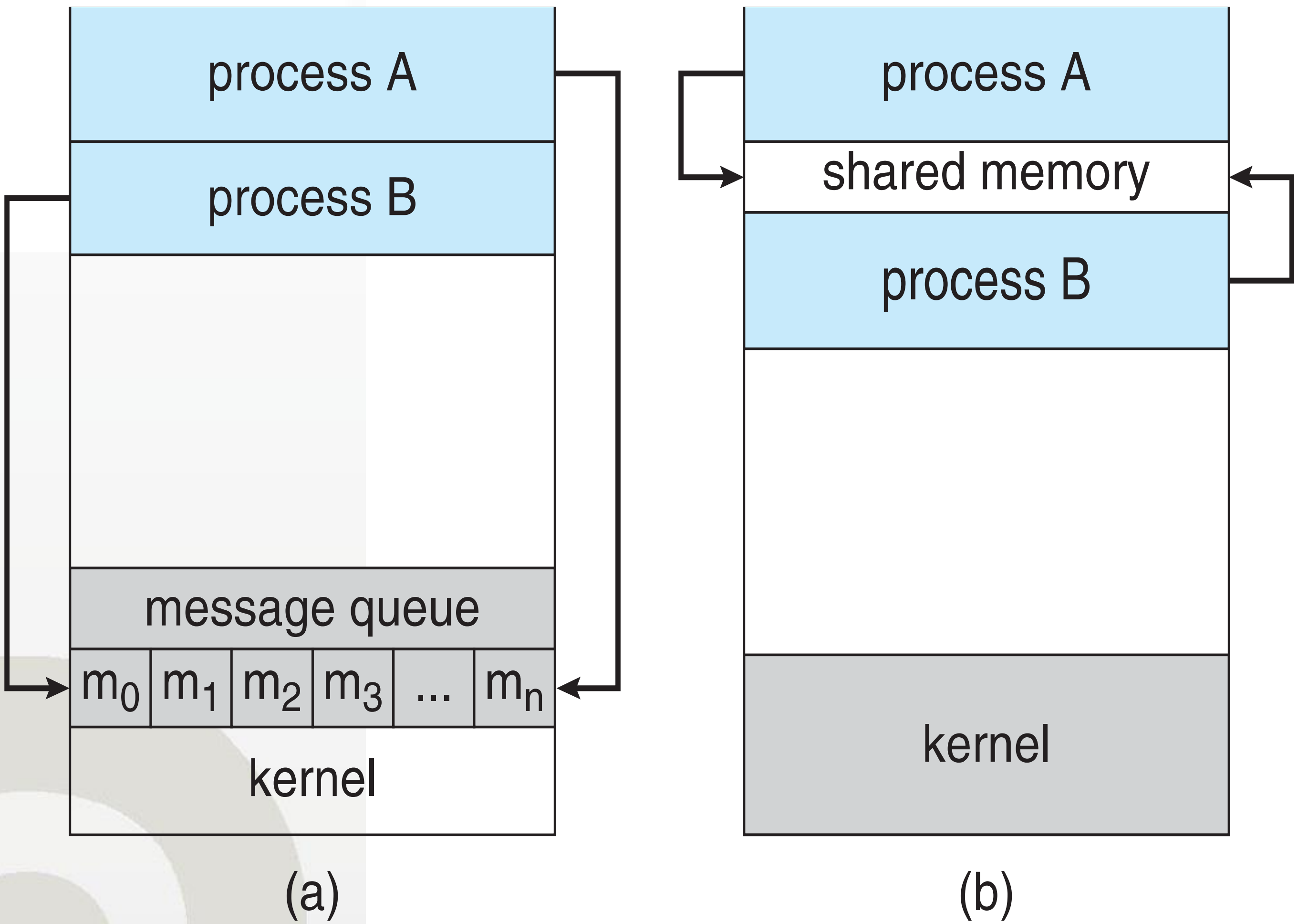
Interprocess Communication

- Processes within a system may be **independent** or **cooperating**
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need **interprocess communication (IPC)**

Intertask Communication

- Two models of intertask (interprocess) communication
  - Shared memory
  - Message passing
- Task synchronization
  - Critical Regions
  - Mutexes
  - Semaphores

Communications Models



Message Passing

Shared Memory

Mailboxes (Messages)

- Messages are sent to and received from **mailboxes**, which provide an intertask communication mechanism
- **Mailbox** is a special memory location that one or more tasks can use to pass data, or more generally for synchronization.
- Tasks rely on the kernel to allow them to write to the location via a `post(send)` operation or to read from it via a `pend (receive)` operation.
- Mailboxes are available in most commercial RTOS.

Buffering Data (Shared Memory)

- Mechanisms are needed to pass data between tasks in a multitasking system when production and consumption rates are unequal.
- Global variables are simple and fast, but have collision potential.
- Example, one task may produce data at a constant 1000 units per second, whereas another may consume these data at a rate less than 1000 units per second.
  - Assuming that the production interval is finite (and relatively short), the slower consumption rate can be accommodated if the producer fills a storage buffer with the data.
  - The buffer holds the excess data until the consumer task can catch up.
  - The buffer can be a queue or other data structure, including an unorganized mass of variables.
  - If consumer task consumes this information faster than it can be produced, or if the consumer cannot keep up with the producer, problems occur.
- Selection of the appropriate size buffer and synchronization mechanisms is critical in reducing or eliminating these problems.

Time-Correlated Buffering

- Can use global variables for double buffering or ping-pong buffering.
- Used when time-relative (correlated) data need to be transferred between cycles of different rates, or when a full set of data is needed by one process but can only be supplied slowly by another process.
- Variant of the classic bounded buffer problem in which a block of memory is used as a repository for data produced by "writers" and consumed by "readers."
- Further generalization is the readers and writers problem in which there are multiple readers and multiple writers of a shared resource

Time-Correlating Buffering Example

- IMU data measures x, y and z accelerometer pulses in a 10 msec task
- Raw data processing is processed in the 40 msec task with lower priority (RM scheduling)
- Accelerometer data must be time-correlated, i.e., it is not allowed to process x(k) and y(k) with z(k+1) data samples – one reason for happening of such scenario can be the 10 msec task interrupt before z data processing.
- Solution – use buffering variables xb, yb and zb in the 40 msec task with interrupt disabled.

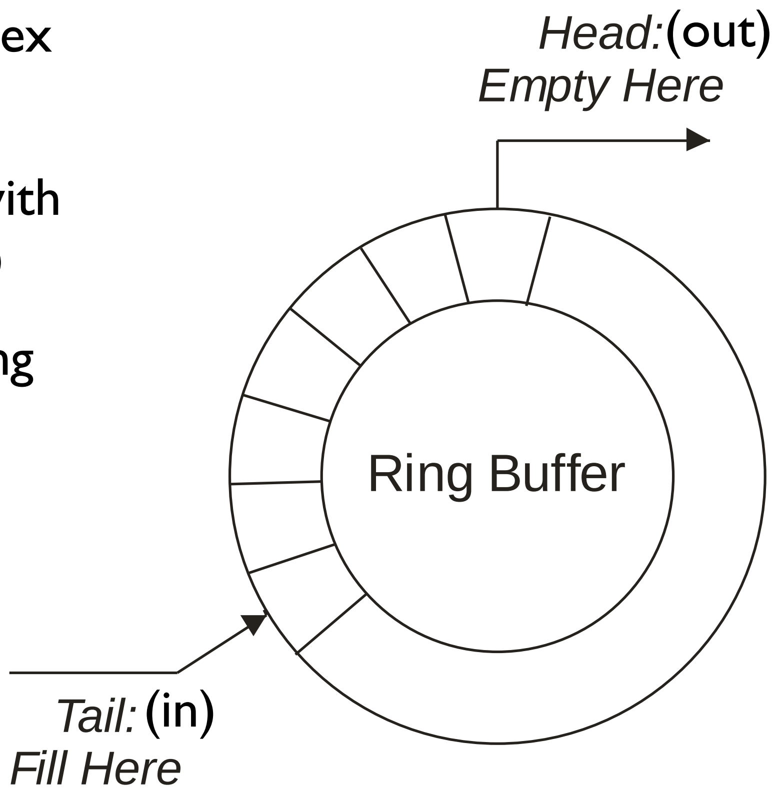
```
introff();          /* disable interrupts */
xb=x;               /* buffer x data */
yb=y;               /* buffer y data */
zb=z;               /* buffer z data */
intron();           /* enable interrupts */
process(xb,yb,zb);  /* use buffered data */
...
```

Ring Buffers

- A circular queue or ring buffer can be used to solve the problem of synchronizing multiple reader and writer tasks.
- Simultaneous input and output to the list are achieved by keeping head and tail indices.
- Data are loaded at the tail and read from the head.

Ring Buffers

- Tasks write to the buffer at the `tail` index and read data from the `head` index.
- Buffer is implemented as a circular array with two pointers `in(tail)` and `out(head)`
- Data access is synchronized with a counting semaphore set to the size of ring buffer `BUFFER_SIZE (n)`.
- Buffer is empty when:
  `tail(in) == head(out)`
- Buffer is full when:
  `((in + 1) % BUFFER_SIZE) == out`



Ring Buffering Example

- Suppose the ring buffer is a data structure of type `ring_buffer`

```
typedef struct ring_buffer
{
  int contents[n]; /* buffer area */
  int head=0;      /* head index */
  int tail=0;      /* tail index */
}
```

```
#define BUFFER_SIZE 10
typedef struct {
  . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

Ring Buffering Example Cont

- Implementation of the `write(data,&s)` and `read(data,&s)` operations, writing and reading data to the ring buffer s respectively.

```
void write(int data,ring_buffer *s)
{
 if ((s->tail+1) % n==head)
  error();    /* buffer overflow */
 else
 {
  s->contents+tail=data;     /* write data */
  tail=(tail+1) % n;         /* update tail */
 }
}
```

```
item next_produced;
while (true) {
      /* produce an item in next_produced */
      while (((in + 1) % BUFFER_SIZE) == out)
            ; /* do nothing */
      buffer[in] = next_produced;
      in = (in + 1) % BUFFER_SIZE;
}
```

Ring Buffering Example Cont

```
void read(int data,ring_buffer *s)
{
 if (s->head==s->tail)
 data=NULL; /* buffer underflow */
else
{
 data=s->contents+head;    /* read data */
 s->head=(s->head+1) % n; /* update head */
 }
}
```

Allows at most `BUFFER_SIZE-1` items in the buffer at the same time

```
item next_consumed;
while (true) {
      while (in == out)
            ; /* do nothing */
      next_consumed = buffer[out];
      out = (out + 1) % BUFFER_SIZE;
      /* consume the item in next consumed */
}
```

Ring Buffering Example Cont

- Suppose that we need to fills ***all*** the buffer `BUFFER_SIZE`.
- Integer **`counter`** keeps track of the number of full buffer.
- Initially, **`counter`** is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

```
while (true) {
        /* produce an item in next produced */
        while (counter == BUFFER_SIZE) ;
                /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

```
while (true) {
        while (counter == 0)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        counter--;
        /* consume the item in next consumed */
}
```

Race Condition

- `counter++` could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- `counter--` could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

To guard against the **race condition,** only one task at a time can be manipulating the variable `counter`. **Tasks should be synchronized**

- Consider this execution interleaving with "count = 5" initially:

S0: producer execute `register1 = counter` {register1 = 5}
S1: producer execute `register1 = register1 + 1` {register1 = 6}
S2: consumer execute `register2 = counter` {register2 = 5}
S3: consumer execute `register2 = register2 - 1` {register2 = 4}
S4: producer execute `counter = register1` {counter = 6 }
S5: consumer execute `counter = register2` {counter = 4}

Critical Regions

- Multitasking systems are concerned with resource sharing.
- When resources can only be used by one task at a time, and use of the resource cannot be interrupted they are said to be "**serially reusable**"
- Serial reusable devices include certain peripherals, shared memory, and the CPU.
- The CPU protects itself against simultaneous use.
- Code that interacts with the other serially reusable resources is called a "**critical region**" ("**critical section**")
- If two tasks enter the same critical region (section) simultaneously, a catastrophic error can occur.
- Simultaneous use of a serial reusable resource results in a "**collision**."
- The concern is to provide a mechanism for preventing collisions.

Critical Regions

- Consider two C programs, **Task_A** and **Task_B**, in a round-robin system.
- **Task_B** outputs the message "**I am task_B**" and **Task_A** outputs the message "**I am Task_A**."
- In the midst of printing, Task_B is interrupted by Task_A, which begins printing. The result is the incorrect output:

  I am I am Task_A Task_B

- More serious complications could arise if both tasks were controlling devices in an embedded system.

Critical Regions

- A task may have a **critical region (section)** segment of code changing common variables, updating table, writing file, etc.
- When one task in critical section, no other may be in its critical section
- Each task must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

```
do {

    entry section

        critical section

    exit section

        remainder section

} while (true);
```

Synchronization Hardware

- Many systems provide hardware support for critical section code
- All solutions below based on idea of **locking** - Protecting critical regions via locks
- Single-processor environment – interrupts could be disabled
  - *Currently running critical region code execute without preemption*
  - *No unexpected modifications could be made to the shared variable*
- Modern machines provide special atomic hardware instructions (system calls) to solve the critical region problem
  - **Atomic** = non-interruptible

Mutex Locks

- **Mutex lock** - simplest software tool to solve critical region problems
  - The term *mutex* is short for *mut*ual *ex*clusion
- A process must acquire the lock () before entering a critical region - **`acquire()`** function;
- It releases the lock when it exits the critical region - **`release()`** function
  - A boolean variable **`available`** indicate if lock is available or not
- Calls to **`acquire()`** and **`release()`** must be atomic
  - Usually implemented via hardware atomic instructions
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**

acquire() and release()

```
do {
    acquire lock
        critical section
    release lock
        remainder section
} while (true);
```

The definitions:

```
acquire()
{
   while (!available)
       ; /* busy wait */
   available = false;;
}
```

```
release()
{
   available = true;
}
```

Mutex Lock Example

```
#define MAX_PROCESSES 255
int number_of_processes = 0;

/* the implementation of fork() calls this function */
int allocate_process() {
int new_pid;

   if (number_of_processes == MAX_PROCESSES)
      return -1;
   else {
      /* allocate necessary process resources */
      ++number_of_processes;

      return new_pid;
   }
}

/* the implementation of exit() calls this function */
void release_process() {
    /* release process resources */
    --number_of_processes;
}
```

a. Identify the race condition(s).

b. Assume you have a mutex lock named `mutex` with the operations `acquire()` and `release()`. Indicate where the locking needs to be placed to prevent the race condition(s).

Mutex Lock Example

```
#define MAX_PROCESSES 255
int number_of_processes = 0;

/* the implementation of fork() calls this function */
int allocate_process() {
int new_pid;           acquire();

   if (number_of_processes == MAX_PROCESSES)
      return -1;
   else {
      /* allocate necessary process resources */
      ++number_of_processes;
                                release();
      return new_pid;
   }
}

/* the implementation of exit() calls this function */
void release_process() {       acquire();
   /* release process resources */
   --number_of_processes;      release();
}
```

a. Identify the race condition(s).

b. Assume you have a mutex lock named `mutex` with the operations `acquire()` and `release()`. Indicate where the locking needs to be placed to prevent the race condition(s).

Pthreads Synchronization

- Mutex locks – main synchronization techniques used with Pthreads

```
#include <pthread.h>

pthread_mutex_t mutex;

/* create the mutex lock */
pthread_mutex_init(&mutex,NULL);

/* acquire the mutex lock */
pthread_mutex_lock(&mutex);

/* critical section */

/* release the mutex lock */
pthread_mutex_unlock(&mutex);
```

*If the mutex lock is unavailable when **thread_mutex_lock()** is invoked, the calling thread is blocked until the owner invokes **pthread_mutex_unlock()***

Pthreads Synchronization Example

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

pthread_t tid[2];
int counter;

void* doSomeThing(void *arg)
{
    unsigned long i = 0;
    counter += 1;
    printf("\n Job %d started\n", counter);

    for(i=0; i<(0xFFFFFFFF);i++);
    printf("\n Job %d finished\n", counter);

    return NULL;
}
```

```
int main(void)
{
    int i = 0;
    int err;

    while(i < 2)
    {
        err = pthread_create(&(tid[i]), NULL, &doSomeThing, NULL);
        if (err != 0)
            printf("\ncan't create thread :[%s]", strerror(err));
        i++;
    }

    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);

    return 0;
}
```

Pthreads Synchronization Example

Result of the program execution:

```
$ ./tgsthreads
Job 1 started
Job 2 started
Job 2 finished
Job 2 finished
```

Insert Mutexes to prevent race condition

Pthreads Synchronization Example

Insert Mutexes to prevent race condition

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

pthread_t tid[2];
int counter;
pthread_mutex_t lock;

void* doSomeThing(void *arg)
{
    pthread_mutex_lock(&lock);

    unsigned long i = 0;
    counter += 1;
    printf("\n Job %d started\n", counter);

    for(i=0; i<(0xFFFFFFFF);i++);

    printf("\n Job %d finished\n", counter);

    pthread_mutex_unlock(&lock);

    return NULL;
}
```

```c
int main(void)
{
    int i = 0;
    int err;

    if (pthread_mutex_init(&lock, NULL) != 0)
    {
        printf("\n mutex init failed\n");
        return 1;
    }

    while(i < 2)
    {
        err = pthread_create(&(tid[i]), NULL, &doSomeThing, NULL);
        if (err != 0)
            printf("\ncan't create thread :[%s]", strerror(err));
        i++;
    }

    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
    pthread_mutex_destroy(&lock);

    return 0;
}
```

Any Questions?