C++11 Concurrency: Background Concepts

COMP.2040 – Computing IV

Dr. Tom Wilkes

Sources & References

- Abraham Silberschatz, Peter Gagne, and Greg Peterson, *Operating Systems Concepts*, 9th edition (2013), Chapters 3-5
- Anthony Williams, *C++ Concurrency in Action: Practical Multithreading*, 1st edition (2012)
- Nicolai Josuttis, *The C++ Standard Library: A Tutorial and Reference*, 2nd edition (2012), Chapter 18
- Scott Meyers, *Effective Modern C++*, 1st edition (2015), Chapter 7

Outline

- Process concept
- Multicore programming
- Concurrency vs. parallelism
- Amdahl’s Law
- Thread concept
- Thread libraries
- Process and thread synchronization
- Race conditions and the critical section problem
- Mutex locks
- Deadlock and starvation

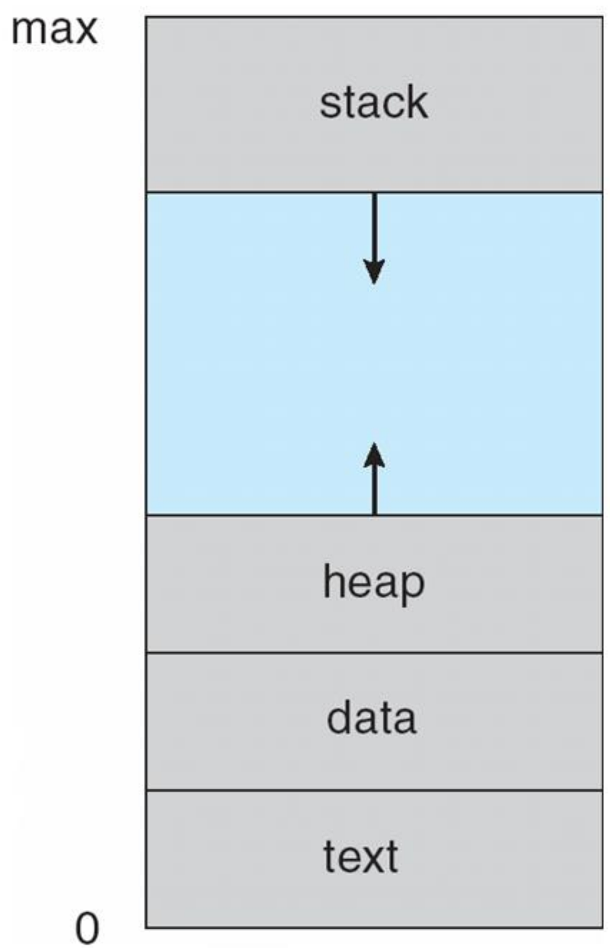
Process Concept

- An operating system executes a variety of programs:
  - Batch system – **jobs**
  - Time-shared systems – **user programs** or **tasks**
- Many authors use the terms ***job*** and ***process*** almost interchangeably
- **Process** – a program in execution
- A process comprises multiple parts:
  - The program code, also called **text section**
  - Current activity including **program counter**, processor registers
  - **Stack** containing temporary data
    - Function parameters, return addresses, local variables
  - **Data section** containing global variables
  - **Heap** containing memory dynamically allocated during run time

Process Concept (cont.)

- A program is a ***passive*** entity stored on disk (**executable file**), whereas a process is ***active***
  - A program becomes a process when its executable file is loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc.
- One program can be several processes
  - Consider executing multiple copies of the same program
  - Google Chrome browser is implemented using a separate process for each browser tab

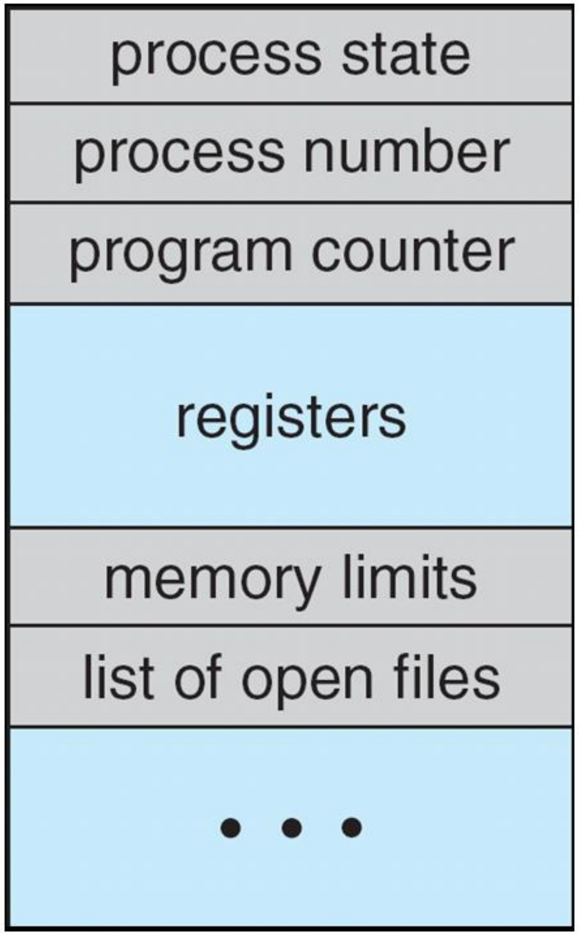
A Process in Memory



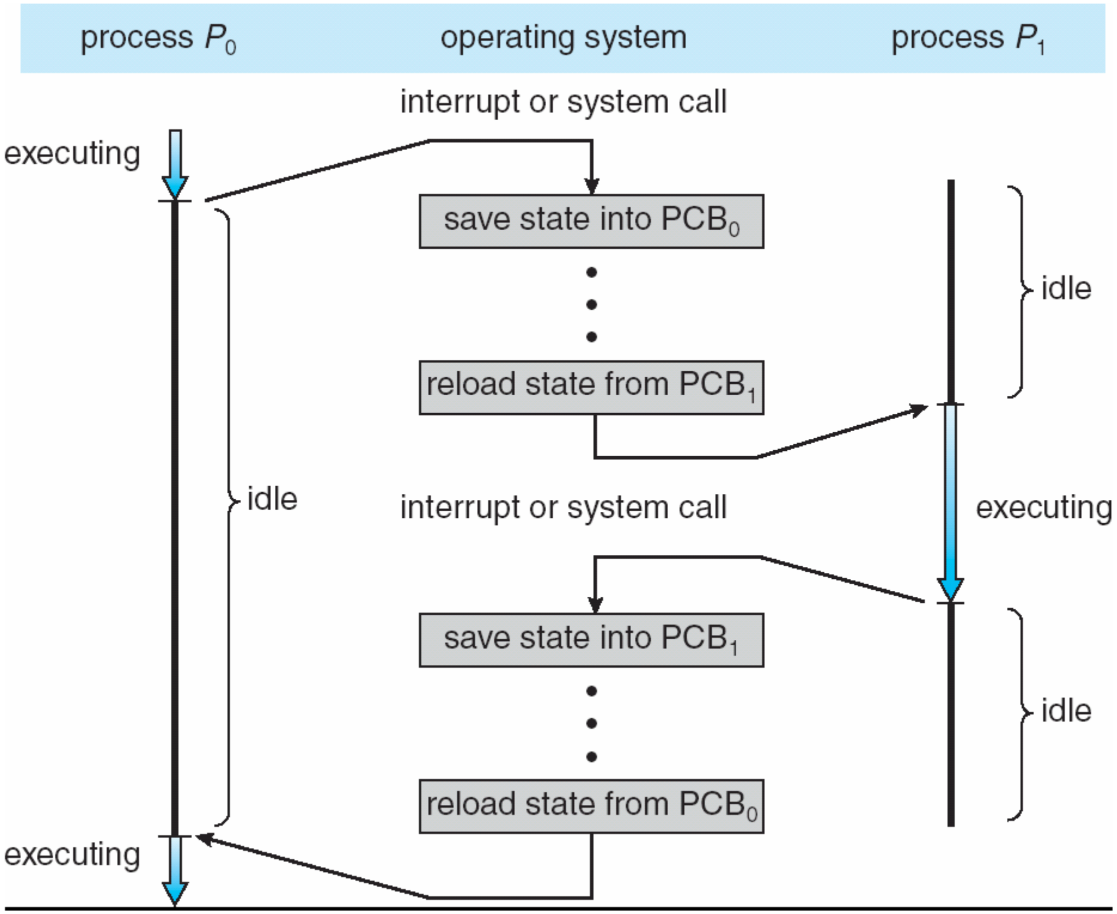
Process Control Block (PCB)

Information associated with each process (also called **task control block**):

- **Process state** – running, waiting, etc..
- **Program counter** – location of instruction to execute next
- **CPU registers** – contents of all process-centric registers
- **CPU scheduling information-** priorities, scheduling queue pointers
- **Memory-management information** – memory allocated to the process
- **Accounting information** – CPU used, clock time elapsed since start, time limits
- **I/O status information** – I/O devices allocated to process, list of open files



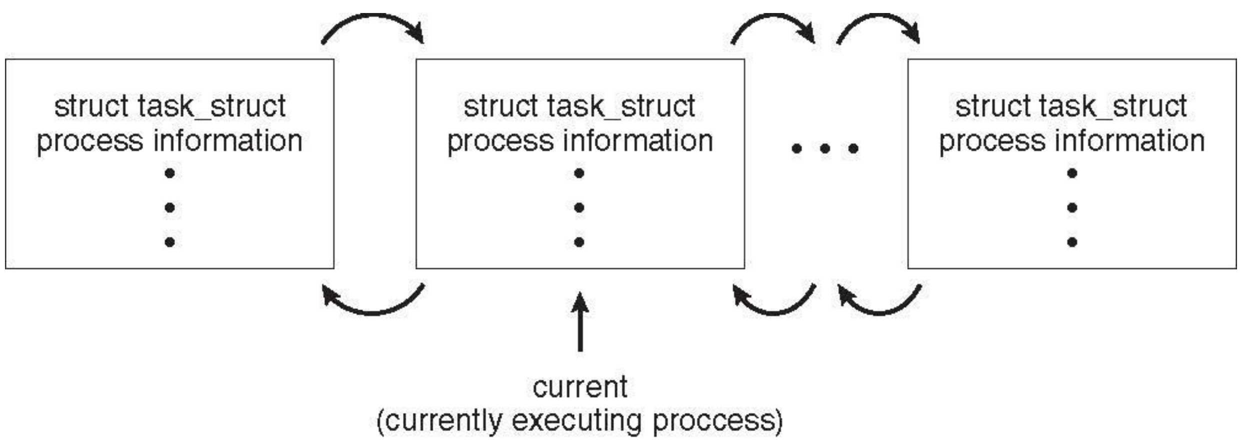
CPU Switch From Process to Process ("Context Switch")



Process Representation in Linux

Represented by the C structure `task_struct`

```
pid t_pid; /* process identifier */
long state; /* state of the process */
unsigned int time_slice /* scheduling information */
struct task_struct *parent; /* this process’s parent */
struct list_head children; /* this process’s children */
struct files_struct *files; /* list of open files */
struct mm_struct *mm; /* address space of this process */
```



Multicore Programming

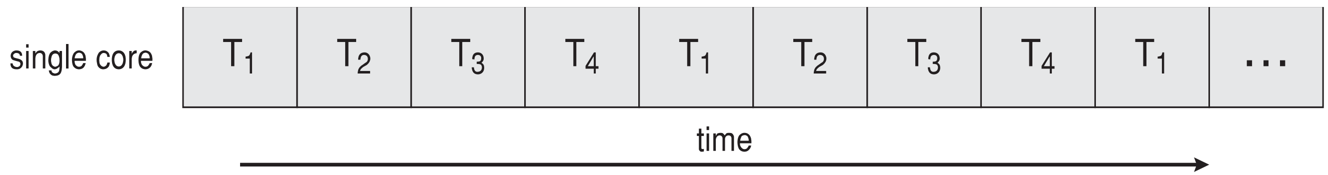
- **Multicore** or **multiprocessor** systems are putting pressure on programmers; challenges include:
  - **Dividing activities**
  - **Balance**
  - **Data splitting**
  - **Data dependency**
  - **Testing and debugging**
- ***Parallelism*** implies a system can perform more than one task simultaneously
- ***Concurrency*** supports more than one task making progress
  - Single processor / core, scheduler providing concurrency

Multicore Programming (Cont.)

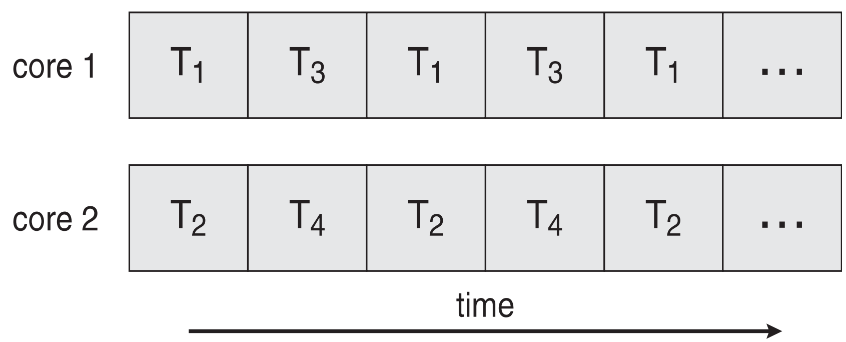
- Types of parallelism
  - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
  - **Task parallelism** – distributing threads across cores, each thread performing unique operation
- As # of threads grows, so does architectural support for threading
  - CPUs have cores as well as ***hardware threads***
  - Consider Intel® Core™ i9-7940X Processor with 18 general purpose cores, and 2 hardware threads per core
  - High-end graphics cards have thousands of specialized graphics cores (GPUs)

Concurrency vs. Parallelism

- **Concurrent execution on a single-core system:**



- **Parallelism on a multi-core system:**



Amdahl's Law

- Identifies performance gains from adding additional cores to an application that has both serial and parallel components
- *S* is serial portion
- *N* processing cores

What if S = 0%?

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

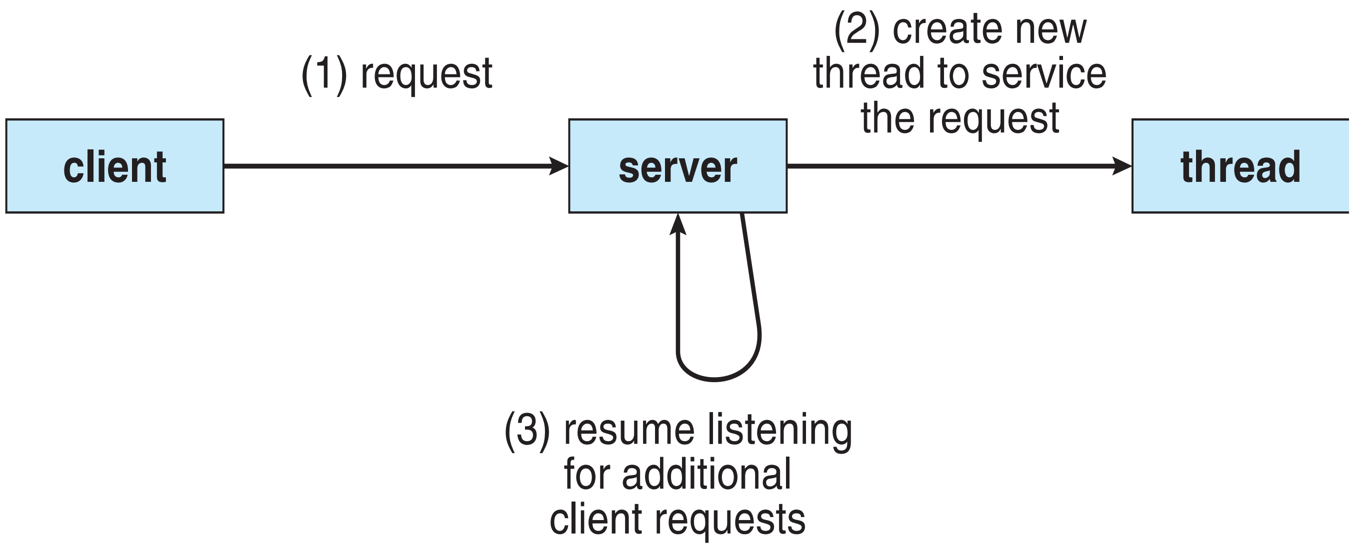
- That is, if application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times
- As *N* approaches infinity, speedup approaches 1 / *S*

Serial portion of an application has disproportionate effect on performance gained by adding additional cores

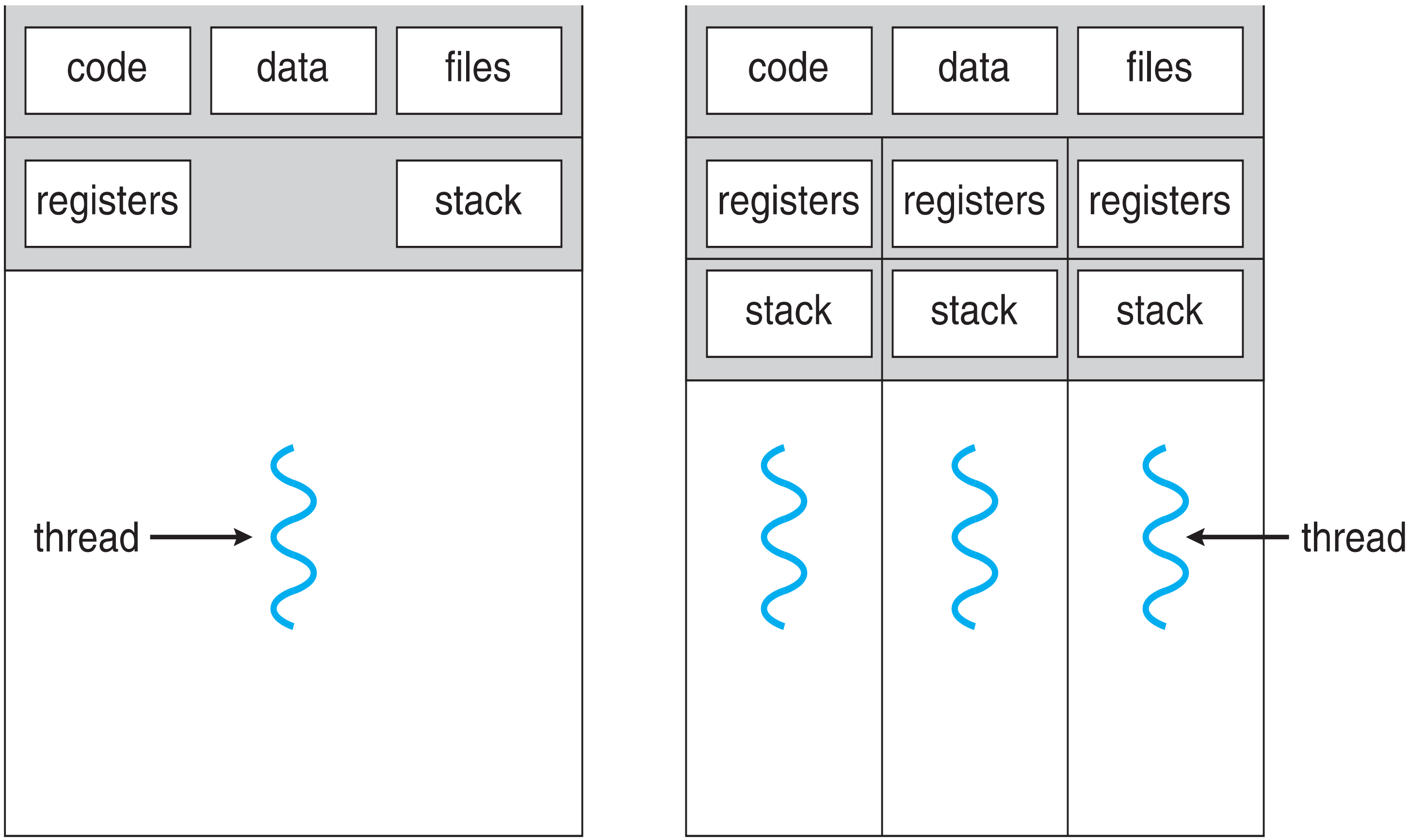
Thread Concept

- So far, each process has a single thread of execution
- Consider having multiple program counters per process
    - Multiple locations can execute at once
        - Multiple threads of control -> **threads**
- Must then have storage for thread details, multiple program counters in PCB

Multithreaded Server Architecture



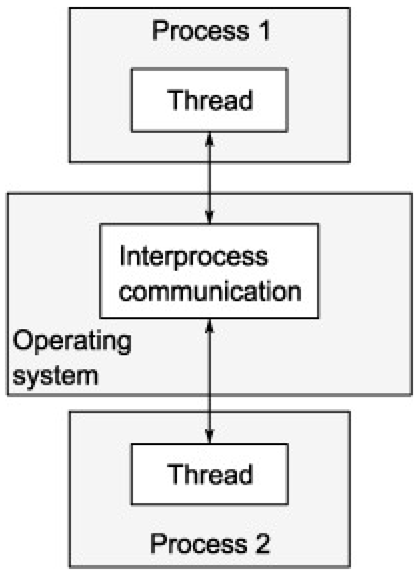
Single and Multithreaded Processes



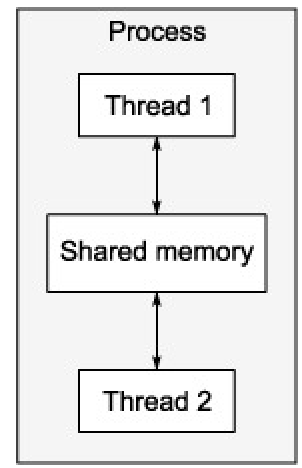
single-threaded process

multithreaded process

Communication between Threads



Communication between single threads in a pair of processes running concurrently



Communication between a pair of threads running concurrently in a single process

User Threads and Kernel Threads

- **User threads** - Management done by user-level threads library
  - Primary user thread libraries:
    - POSIX **Pthreads**
      - Used by most C++ programmers prior to C++11
    - Windows threads
    - Java threads
    - C++11 threads
- **Kernel threads** - Supported by the operating system kernel
  - Examples – virtually all general purpose operating systems, including:
    - Linux
    - Mac OS X
    - Windows

Thread Libraries

- A **thread library** provides the programmer with an API for creating and managing threads
- Two primary ways of implementing thread libraries:
  - Library entirely in user space
  - Kernel-level library supported by the OS

Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- ***Specification***, not ***implementation***
- 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)

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);
```

Process and Thread Synchronization

- Processes and threads can execute concurrently
  - May be interrupted at any time, partially completing execution, and resume later
- Concurrent access to shared data may result in data inconsistency
  - "**Race conditions**": Program results depend on precise details of interleaving of operations between processes/threads, which can vary
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes or threads
  - These mechanisms enforce a **serial ordering** of critical operations in a concurrent system (also called **serialization**)

Synchronization Problem Example

- Suppose that we wanted to provide a solution to the consumer-producer problem that fills ***all*** the slots in a circular buffer.
- We can do so by having an integer **`counter`** that keeps track of the number of full slots.
- Initially, **`counter`** is set to 0.
- **`counter`** is incremented by the producer after it produces a new item in the buffer, and is decremented by the consumer after it consumes an item from the buffer.

Producer/Consumer Pseudocode

Producer code

```
while (true)
{
  /* produce an item */
  while (counter == BUFFER_SIZE)
    /* wait */;
  buffer[in] = next_produced;
  in = (in + 1) % BUFFER_SIZE;
  counter++;
}
```

Consumer code

```
while (true)
{
  while (counter == 0)
    /* wait */;
  next_consumed = buffer[out];
  out = (out + 1) % BUFFER_SIZE;
  counter--;
  /* consume the item */
}
```

Race Condition Example

- **`counter++`** could be implemented at the CPU instruction level as:

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

- **`counter--`** could be implemented as:

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

- Consider the following possible 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 Section Problem

- Consider a system of ***n*** processes or threads $\{p_0, p_1, \ldots p_{n-1}\}$
- Each thread has a segment of code called a **critical section**
  - Thread may be changing shared variables, updating table, writing file, etc.
  - When one thread p_i is in its critical section, no other thread p_k may be in its own critical section
- ***Critical section problem*** is to design a protocol to solve this
  - Protocol must enforce ***mutual exclusion*** among processes with respect to their critical sections
- Each thread must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

Critical Section

- General structure of process or thread $\boldsymbol{P_i}$

```
do {
    [entry section]
        critical section
    [exit section]
        remainder section
} while (true);
```

Mutex Locks

- Operating system designers build software tools (provided via library APIs) to solve the critical section problem
- Simplest is **mutex lock**
- Protect a critical section by first acquiring the mutex lock via **`lock()`**, then releasing the mutex lock via **`unlock()`**
  - The mutex lock encapsulates a Boolean variable indicating whether the lock is available or not
- Calls to **`lock()`** and **`unlock()`** must be **atomic** (indivisible / uninterruptible)
  - Usually implemented via hardware atomic instructions
- The simplest implementation of mutex locks requires **busy waiting** (see next slide)
  - This type of mutex lock implementation is therefore called a **spinlock**
- A more sophisticated implementation uses a queue for processes or threads waiting for the mutex lock to become available
  - The process or thread yields control of the processor ("sleeps") while waiting on the queue

Solution to Critical-section Problem Using Mutex Locks

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

`lock()` and `unlock()` Operations: Implementation using Busy Wait

- Implementation of `lock()` operation:

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

- Implementation of `unlock()` operation:

```
unlock() /* release lock */
{
   available = true;
}
```

`lock()` and `unlock()` Operations: Implementation using Wait Queue

- Implementation of `lock()` operation:

```
lock()
{
   if (!available)
   {
      yield control of the processor;
      enqueue the calling process or thread on the wait queue
        of this mutex lock;
   }
   /* at this point, the process/thread has been woken */
   available = false;
}
```

- Implementation of `unlock()` operation:

```
unlock()
{
   available = true;
   if the wait queue of this mutex lock is not empty
     wake up the process/thread at the head of the queue;
}
```

Deadlock and Starvation

- **Deadlock** – two or more processes or threads are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let ***S*** and ***Q*** be two mutex locks:

P_0	P_1
`S.lock();`	`Q.lock();`
`Q.lock();`	`S.lock();`
`...`	`...`
`S.unlock();`	`Q.unlock();`
`Q.unlock();`	`S.unlock();`

- **Starvation** – **indefinite blocking**
  - A process may never be removed from the mutex queue in which it is suspended
- **Priority Inversion** – Scheduling problem when a lower-priority process or thread holds a lock needed by a higher-priority process
  - Solved via **priority-inheritance protocol**