Concurrent Computing 1: Introduction

1. **Introduction**
   - **thread of execution**
   - **OS threads**
   - **multi-process vs. multithreaded**
   - **advantages and disadvantages**
2. Terminology
3. Synchronization
4. C and Concurrency
5. More Asynchrony
6. Event-Driven Programming

Concurrent Computing

Concurrent computing (or **concurrent programming**) refers to programs that create *multiple* **threads of execution**.

A **thread of execution** (or **thread of control**) is a *sequence of program statements* being executed as a unit.

A concurrent program uses *multiple threads of execution* to together accomplish the overall goals of the program.

The multiple threads are run *concurrently*, i.e., *"in parallel."*

We say *"in parallel"* because parallelism will be merely *simulated* unless the hardware contains *multiple* **CPUs/cores**.

Concurrent Computing (contd.)

Parallel execution is simulated with a single core through **time slicing**: one thread runs for a brief period of time, then a second is run, etc., in a *"round robin"* manner more or less.

So the term **concurrent execution** means there are multiple threads of execution, which may be executing simultaneously or may be *interleaved* via time slicing.

The term **parallel execution** indicates *true simultaneous execution*.

Thread of Execution vs. OS Thread

An **OS thread** is an operating system construct that is used to encapsulate one of possibly multiple *threads of execution* running within the context of a *single process*.

OS threads allow concurrent programs where the program statements in the multiple threads of execution are all part of the same program.

Virtually all modern OS kernels support **multithreading** (OS threads).

The Linux/UNIX OS threads model is known as **POSIX threads** or **Pthreads**.

Thread of Execution vs. OS Thread (contd.)

The term *thread* derives from "thread of execution," but the word thread alone can refer to either (1) a thread of execution or (2) an OS thread.

The focus of these slides is on general concepts in concurrent programming.

"Thread" will be used in the generic *thread of execution* sense.

When we need to refer to the operating system capability, we will use the phrase *"OS thread."*

Multi-Process vs. Multithreaded Programs

A concurrent program can be implemented in one of two ways:

- as *multiple* **processes** (e.g., created with `fork()`)
- as *multiple* **OS threads** (e.g., created with `pthread_create()`)

(It is also possible to mix the two approaches: use multiple processes, some of which contain multiple OS threads.)

A concurrent program that uses *multiple processes* is most commonly called a **multi-process program**.

A concurrent program that uses *multiple OS threads* is most commonly called a **multithreaded program**.

Multi-Process vs. Multithreaded (contd.)

The key difference is whether each thread of execution is a *separate process* or whether all threads are *within a single process*:

- *processes* run in *separate address spaces*
- *OS threads* run in a *single, shared address space*

Normally a running program (a **process**) is being executed at a single point in its address space.

However, OS threads allow simultaneous execution at multiple points in a process' address space.

(Each OS thread has its own separate **program counter**.)

Multi-Process vs. Multithreaded (contd.)

While many of the complications of concurrent programming are common to both approaches, there are some key differences:

- Processes will have to explicitly transfer any data needed by other processes, using **interprocess communication (IPC) mechanisms**.
- OS threads automatically share data due to their shared address space.
- Processes are (largely) independent, so they do not involve as many **shared resources** that need to be protected from corruption.
- OS threads can have significant problems avoiding corruption of shared resources.

Language Support

Some programming languages contain language constructs to support concurrent programming, such as Java (`Thread` class, `run` method of `Runnable` interface, etc.).

Prior to C11, C did *not* provide language support for concurrent programming.

Concurrent programming in C can be accomplished through the use of OS **system calls**.

C11 added support for multithreading to C (more later).

Concurrency Advantages

Important reasons for implementing concurrent programs are:

- program operations that can **block** *indefinitely* may cause a program to be *unresponsive*
- programs may have to be able to rapidly respond to various **asynchronous events**
- programs may have to be able to simultaneously interact with multiple network clients, I/O devices, etc.
- large-scale applications may prefer to run certain functionality "**in the background**," using "spare" CPU cycles
- machines that have multiple CPUs/cores can support true **parallel execution** of program operations

Concurrency Advantages: Blocking

Certain operations involving *I/O or signals* can have the potential to *block indefinitely*.

In a single-thread program, this would mean that the program would get *suspended* while blocked.

The program would not be able to do other useful work or respond to events like user keyboard input, possibly for very significant periods of time.

The result is a program that appears to be "slow" or unresponsive (have poor **interactivity**).

Dividing a program's functionality into multiple threads allows a program to continue to do useful work and/or respond to inputs even while some of the thread(s) are blocked.

Concurrency Advantages: Asynchronous Events

Programs such as those involving GUIs often have to be able to respond relatively rapidly to any of several asynchronous events, such as a keyboard inputs, mouse movements/clicks, and so forth.

If significant computations must be done for each event, a single-thread program might be unable to immediately respond to new events.

This can lead to the GUI, etc. being very unresponsive.

If a separate thread is created to handle each type of event, event handling and event processing will be automatically interleaved via OS thread scheduling.

Concurrency Advantages: Multiple Clients

Most server programs will need to be able to handle more than one client at a time.

However, requests from clients will be received asynchronously and usually unpredictably, and may require varying amounts of time to service.

The most straightforward method to produce a **concurrent server** is to have the server program create a separate thread to handle each client's requests.

Concurrency Advantages: Background Tasks

Large-scale software systems may have components that perform some tasks that should have lower **priority**, i.e., be run only when there are spare CPU cycles.

Example tasks include **memory management** functions such as **garbage collection** and **defragmentation**, updating database summaries/reports, and so forth.

By assigning these tasks to separate threads with lower priority, the OS will take care scheduling them to run only when the system has nothing else useful to do.

Concurrency Advantages: Parallelism

One way to *decrease the runtime* (elapsed time) of a program is to perform at least some of the program's operations *in parallel*, using a machine with multiple CPUs/cores.

This is the standard approach for building **supercomputers**: massively parallel multi-processor systems consisting of thousands of CPUs.

Providing multiple cores is now the most cost-effective approach for increasing the computational capacity of microprocessors.

It is becoming increasingly common that laptops, tablets, and even phones contain two or more CPUs/cores.

Concurrency Advantages: Parallelism (contd.)

While it is obvious that machines with multiple CPUs can run multiple (separate) programs faster than can machines with a single CPU, increasing the speed of a single program on multi-CPU machines requires more effort.

Not only must a program use concurrent programming techniques, the program must be written in such a way that the multiple threads of execution can truly run simultaneously on different CPUs/cores.

How much parallel execution is possible will depend on the program's algorithmic logic, as dependencies between the program's steps will limit it.

Concurrency Alternatives

Of the listed situations where concurrency is an advantage, all but achieving true parallelism can be handled *without multiple threads of execution*.

In most cases, however, the resulting code will be much more complicated, requiring advanced system call programming.

A big advantage of concurrent programs is that they shift the responsibility for recognizing when program subtasks are able to be executed and scheduling among them, from the program to the OS.

In an analogous way, programming languages like Java and Lisp that include *garbage collection* runtime systems, simplify the programming task by shifting much of the burden of memory management away from the program code itself.

Concurrency Alternatives (contd.)

Writing a single-thread program that avoids blocking issues can require use of techniques such as **nonblocking I/O** and testing for **pending signals**, and may require *inefficient* **polling** designs.

Enabling a single-thread program to be able to be responsive in handling multiple events requires using techniques such as **I/O multiplexing**, **signal handlers**, **timers**, and **polling**.

Concurrent servers can be implemented using **I/O multiplexing**, but this approach can require the user of **timers** unless server requests involve little computation.

Given the increasing prevalence of multi-CPU machines, the use of concurrent programming techniques will become increasingly advantageous.

Concurrency Disadvantages

While concurrent programs have clear advantages for certain applications, they generally involve several sources of complexity that are not encountered in traditional single process, single thread programs.

The key source of complexity is that the operations in each thread occur **asynchronously** relative to those in the other threads.

This means that programmers cannot be certain what order the operations in the different threads will end up being *interleaved* (or even which may be executed simultaneously).

Furthermore, operation order may differ from run to run!

Concurrency Disadvantages (contd.)

The reason for this is the decisions about when and for how long to execute any of the threads (whether processes or OS threads) are made by the kernel **scheduler** rather than the programmer.

In traditional single thread programs, operations are **synchronous**: the programmer knows exactly what *order* all operations will be executed in (though he may not know which branches will be taken) because operation order will be based on the order he used for the program statements.

Concurrent programs can suffer from **race conditions**, where the (correct) output of the program depends critically on the *relative order* in which certain operations in the threads end up being executed.

Concurrency Disadvantages (contd.)

When operations in different threads get scheduled in undesired or unforseen orders, a concurrent program with a race condition may produce incorrect output or even **deadlock** (hang).

Concurrent programs must either be designed to work with *any possible interleaving* of thread operations or else the processes must include **synchronization** operations.

Synchronization can ensure that operations get interleaved only in appropriate orders and/or that certain operations do not occur simultaneously.

Unfortunately, testing and debugging concurrent programs is difficult; even if a concurrent program is run very many times with a range of inputs, this does not guarantee that every possible operation ordering has been seen.

Concurrent Computing 2: Terminology

1. Introduction
2. **Terminology**
   - **terminology summary**
   - **asynchrony**
   - **race conditions**
   - **deadlock**
3. Synchronization
4. C and Concurrency
5. More Asynchrony
6. Event-Driven Programming

Terminology

Synchronous (program/operations): the order in which program operations are executed is predictable.

Asynchronous/asynchrony: (some) program operations do *not* occur in a predictable order or at predictable times relative to one another.

Synchronization: operations/mechanisms that make certain that specific operations in separate threads of execution occur in the proper order or do not occur at the same time.

Serialization: enforcing one particular order among operations that might otherwise occur in various orders (i.e., asynchronously); can be both a positive as when done to avoid **race conditions** (below) and a negative as when it causes a program to block when it could do useful work.

Terminology (contd.)

Race condition: a logic error in a concurrent program, where incorrect results can be obtained if the operations in different threads of execution get scheduled to occur in a particular order (or to not occur in a particular order).

Atomicity: a program operation is said to be **atomic** if it cannot be interrupted and cannot encounter state changes during its instruction sequence.

Shared resource: a program object that can be simultaneously accessed by multiple threads (e.g., a variable, an open file).

Critical section: a program fragment that accesses a *shared resource* that should not be simultaneously accessed by multiple threads.

Terminology (contd.)

Mutual exclusion: refers to both the need to avoid multiple threads simultaneously entering a *critical section* and the OS or program constructs that enforce this.

Mutex: shortened form of "mutual exclusion," but typically refers to an OS/program mechanism for enforcing mutual exclusion among threads; i.e., a *synchronization* mechanism.

Busy wait (**spinlock**): *synchronization* method where a thread of execution effectively sits in a *loop* repeatedly testing for some condition (e.g., if a variable's value is non-zero) before proceeding; generally undesirable as it can waste CPU cycles and lead to **resource starvation** for other threads.

Terminology (contd.)

Deadlock: situation where two or more threads of execution *make no progress* because each is waiting for the other(s) to take an action; most commonly it is because each is waiting for the other(s) to release *shared resources* before they can proceed.

Livelock: similar to deadlock in that two or more processes make no progress, only difference being that the processes are not suspended but are running (e.g., two processes each try to break a deadlock by taking an action that still thwarts the other).

(Resource) Starvation: a thread of execution is perpetually denied resources required to be able to run (e.g., CPU cycles).

Terminology (contd.)

Priority Inversion: type of **starvation** where a *low-priority* thread *acquires a shared resource* before a *high-priority* thread, so the high-priority thread cannot run, but the low-priority thread never gets run so it cannot release the shared resource.

Semaphore: mechanism for controlling access to a shared resource, effectively an integer representing the number of clients that may currently be allowed to access the resource.

Lock: *synchronization* method for *mutual exclusion*, usually used to prevent multiple threads of execution from simultaneously accessing a *shared resource* (in *critical sections*).

Terminology (contd.)

Monitor: basically a *mutex* plus a mechanism for *signaling* threads when a condition is met that will allow access.

Condition Variables: signaling mechanism similar to **signals** but designed to work at the thread level in conjunction with **mutexes** to avoid need for *busy waiting*; basically what is required for a *monitor* (above).

Barrier: synchronization mechanism that forces all threads to stop at a point until every one reaches that point.

Memory Barrier (or **Fence**): machine instruction that enforces memory access ordering constraints.

Terminology (contd.)

Reentrant: code whose execution can be interrupted and run again ("*re-entered*") without affecting the results from the interrupted execution when/if it is resumed; not specifically related to concurrency, also relevant to the use of **signal handlers** and to **recursive** code.

Nonreentrant: code that is *not reentrant*; typically this is because the code is capable of modifying data that is *not local to each invocation* (e.g., `static` or global variables).

Thread-Safe Function: function that can be invoked from multiple simultaneous OS threads without affecting its semantics (as compared with a single invocation).

Asynchrony

Consider the following *single-thread* program:

```
int main()
{
  int x, y;
  ...read in the initial values for x & y...
  x = x + y;
  y = x - y;
  if (x > y)
    printf("x larger than y");
  else
    printf("y larger than x");
}
```


Asynchrony (contd.)

Program overview:

- assume `x` and `y` are read in from the terminal or a file
- `x` and `y` are then updated as shown
- their values are compared and the appropriate message printed

The operations in such a *single-thread* program are **synchronous**:

- we know that `x` will have its value changed before `y` does
- we don't know which `printf` statement will be executed (that will depend on the initial values assigned to `x` and `y`), but we know that the `if` test will occur after both `x` and `y` have been updated
- the program will always print the same message given the same inital values for `x` and `y`

Asynchrony (contd.)

Now consider a similar but *multithreaded* (*Pthreads*) program:

```
//Global variables:
int x, y;

//Function to be run in second Pthread:
void *update_y(void *arg)
{
  y = x - y;
  return NULL;  //(Needed due to Pthread calling convention.)
}

int main()
{
  ...read in the initial values for x & y...
  pthread_create(...,update_y,...);
  x = x + y;
  if (x > y)
    printf("x larger than y");
  else
    printf("y larger than x");
}
```


Asynchrony (contd.)

Multithreaded program overview:

- `x` and `y` have been defined as *global variables* so that they are accessible in every **scope** (i.e., every subroutine)
- a global variable is *shared* by all Pthreads since all Pthreads share address space (including the *static* memory segment)
- a new Pthread must run a function, and this function must take a single pointer argument and have pointer return value
- the `pthread_create()` call creates a second Pthread, running the function `update_y()`
- `x` is updated in the initial/main Pthread, `y` is updated in the *second Pthread*

Asynchrony (contd.)

Now various operations are **asynchronous**:

- we don't know whether `x` or `y` will have its value changed first, since this will depend on which Pthread is run first (following `pthread_create()`)
- while `x` will definitely be updated before the `if` test, we cannot be sure whether `y` will
- as a result, the output will not only depend on the initial values assigned to `x` and `y` (in the missing code), but on the *order* in which the two Pthreads get scheduled to run
- multi-CPU/core machines may even intermingle the **machine instructions** that implement the assignment statements
- since Pthreads may be scheduled differently on each execution, the program's output might vary from run to run, even with the same initial values for `x` and `y`

Race Conditions

Because the output of the above multithreaded program depends on the order the OS threads get scheduled to run as well as the input values for `x` and `y`, the program suffers from a **race condition**.

Race conditions are one of the most common *logic errors/bugs* that occur in concurrent programs.

Eliminating the race condition in the above program requires that we introduce **synchronization mechanisms** to ensure that:

- `y` is updated in the second thread only *after* `x` has been updated in the first/main thread
- the `if` condition in the first thread is tested only *after* `y` has been updated in the second thread

Race Conditions (contd.)

Let's examine another, less obvious race condition example.

Consider the following C code fragments:

```
//Globals:
int i = 0;

//Fragment #1:
i = 1;
i++;

//Fragment #2:
i = 3;
i += 1;
```

Assume that the two fragments occur in *separate threads* that are executing *concurrently*.

Race Conditions (contd.)

If fragment #1 runs before #2 then we expect `i` to end up with *value 4*, while if fragment #2 runs before #1 we expect `i` to end up with *value 2*.

However, scheduler decisions about the two threads could cause the fragments' statements to be *interleaved* in six different orders, leading to other possible values for `i`:

- `i=1; i++; i=3; i+=1;` `i` is 4
- `i=3; i+=1; i=1; i++;` `i` is 2
- `i=1; i=3; i++; i+=1;` `i` is 5
- `i=1; i=3; i+=1; i++;` `i` is 5
- `i=3; i=1; i++; i+=1;` `i` is 3
- `i=3; i=1; i+=1; i++;` `i` is 3

Race Conditions (contd.)

If the correct/intended value for `i` is either 2 or 4, then this program suffers from a *race condition* bug that can cause `i` to have an incorrect value of either 3 or 5.

The source of the race condition here is that we *cannot* assume that sequences of *multiple program statements* in a thread of execution will get executed **atomically**.

In fact, we will see (below) that we cannot even assume that *individual C statements* like `i++` will get executed atomically!

If we need to have one fragment get executed in its entirety before the other begins execution, then we will have to use **synchronization mechanisms** in our code to ensure that is the only order in which execution can occur.

Race Conditions (contd.)

One way that we could prevent the fragments' statements from being interleaved is by preventing either from running if the other has started running but not yet completed.

In other words, consider each fragment as a **critical region**, and enforce **mutual exclusion**.

We might imagine that this could be accomplished by using a shared `int` variable, `lock`, where if `lock` is 1 (one) it means it is safe for a fragment's critical region to begin execution, while if it is 0 (zero) it means it is not safe and the thread should wait.

This is the basic concept behind a **mutex** or **binary semaphore**.

Race Conditions (contd.)

So we modify the fragments as follows:

```
//Fragment #1:
while(lock != 1);
lock = 0;
i = 1;
i++;
lock = 1;

//Fragment #2:
while(lock != 1);
lock = 0;
i = 3;
i += 1;
lock = 1;
```


Race Conditions (contd.)

The idea here is that each fragment will wait for the other to finish before it proceeds, since each will sit in a `while` loop testing for `lock`'s value to become 1 (one).

One problem with this approach is that the `while` loop checking `lock` represents a **busy wait** or **spinlock**, so wastes CPU cycles.

A second and even more serious problem is that this approach *completely fails to eliminate the race condition*, as both processes can still end up modifying `i` concurrently.

Since testing `lock` and changing its value are *not done atomically*, we can end up with an execution sequence such as:
thread #1: `... while(lock!=1);`
thread #2: `while(lock!=1); lock=0; i=3;`
thread #1: `lock=0; i=1; i++;`
thread #2: `i+=1;...`

Race Conditions (contd.)

This race condition cannot be eliminated when using ordinary variables in a high-level programming language due to the lack of an *atomic* **test-and-set operator**.

What is required are special *OS system calls* that implement appropriate **synchronization mechanisms**.

Since the OS controls the scheduling of operations across the CPUs/cores, it can provide effective synchronization mechanisms by controlling the execution of processes/threads, blocking CPU interrupts, locking access to memory, and so forth.

All modern OS's provide synchronization mechanisms that can be used to eliminate the race condition in the sample code (e.g., **mutexes** and/or **semaphores**).

Deadlock

Another potential flaw in cocurrent programs is the possibility for **deadlock**.

Deadlock will typically involve a *race condition*, where threads deadlock only under particular interleavings of operations.

Deadlock can occur when two (or more) threads need multiple shared resources to do their processing and each is able to acquire a subset of the resources; to proceed, each thread needs resources held by the other.

This is called a **Coffman deadlock** or **resource deadlock**.

Resource deadlock can often be avoided by making sure that all threads try to acquire common resources in the same order, or by extending the critical sections of the threads to include all resource acquisition.

Deadlock (contd.)

Another common type of deadlock is **communication deadlock**: threads send messages to each other for synchronization, but the messages can get lost.

For example, suppose one thread must send a message to a second to get it to process data, after which the first thread then waits to hear back from the second that it is done.

If either of the two messages is lost, the two threads will end up deadlocked.

This situation can occur when sending **signals**:
if a process has not yet been setup to receive a signal when the signal is received, the signal will probably be discarded, meaning a subsequent `pause()` could then cause the process to sleep forever.

Concurrent Computing 3: Synchronization

1. Introduction
2. Terminology
3. **Synchronization**
   - **Linux synchronization mechanisms**
   - **semaphores and mutexes**
   - **the producer-consumer problem**
4. C and Concurrency
5. More Asynchrony
6. Event-Driven Programming

Linux Synchronization Mechanisms

Linux/UNIX provides several **synchronization mechanisms**, some of which are designed to synchronize *processes*, some *Pthreads*, and some that can work with both.

The synchronization mechanims that work among *processes* are:

- signals
- semaphores
- file locks
- pipes/FIFOs
- wait for child termination

Linux Synchronization Mechanisms (contd.)

The synchronization mechanims that work among *Pthreads* are:

- mutexes (Pthreads only)
- condition variables (Pthreads only)
- barriers (Pthreads only)
- semaphores
- file locks
- wait for thread termination (Pthreads only)

C11 adds thread support to C, and includes mutexes and condition variables as supported synchronization mechanisms.

Semaphores and Mutexes

Two of the most important synchronization-related concepts are the **semaphore** and the **mutex**.

The semaphore was invented by Edsger Dijkstra in 1965.

The name "semaphore" is borrowed from signaling systems such as those involving flags for use between ships.

A *semaphore* can be used to control access to a (finite size) *shared resource pool*, to eliminate *race conditions* in a concurrent program accessing the shared resource.

A *mutex* is intended to enforce *mutually-exclusive access* to *critical sections* in different threads of a concurrent program, to avoid race conditions that could arise if the critical sections were executed concurrently.

Semaphores and Mutexes (contd.)

A semaphore is a special-purpose *shared non-negative integer*:

- a **general semaphore** typically corresponds to the number of currently available objects in a shared resource pool
- a **binary semaphore** can take on only the values 0 or 1
- a semaphore gets initialized when created, typically to its maximum value or its minimum (which is zero)

The current value of a semaphore indicates how many threads could currently gain access to the resource.

When a thread obtains access to the resource, the semaphore is *decremented*.

When access is relinquished, the semaphore is *incremented*.

Semaphores and Mutexes (contd.)

The key to a semaphore is that the semaphore's value can be modified only by *two special operators*:

- an *increment* operator and a *decrement* operator
- these operators must be **atomic** so that they can be used concurrently without the possibility of race conditions
- the *decrement operator* **blocks** (i.e., causes process to *sleep*) if the integer value is zero when it is called (it cannot become negative)

Semaphores and Mutexes (contd.)

Dijkstra originally named the *decrement* operator **P()** and the *increment* operator **V()**, from Dutch words.

Nowdays, decrement/increment are typically referred to using one of the following combinations of names:

- down/up (e.g., ALGOL 68 and the Linux kernel)
- wait/post (e.g., POSIX semaphores)
- wait/signal (e.g., POSIX condition variables)
- lock/unlock (e.g., POSIX mutex)
- acquire/release (e.g., Java)
- procure/vacate (to be similar to original p and v)

Semaphores and Mutexes (contd.)

A *mutex* is effectively identical to a *binary semaphore*.

Some sources claim that they are conceptually somewhat different, but this was not Dijkstra's view:
"The semaphores are essentially non-negative integers; when only used to solve the mutual exclusion problem, the range of their values will even be restricted to '0' and '1'."

In practice, an OS might specialize its "mutex" and "semaphore" models so this equivalence does not hold.

In Linux/UNIX, mutexes can be used among only Pthreads, while semaphores can be used among processes.

The Producer-Consumer Problem

One of the classic examples used to demonstrate issues and solutions in concurrent programming is the **producer-consumer problem** (**bounded buffer** version):

- the program consists of two threads: (1) the **producer** and (2) the **consumer**
- the *producer* creates data to be processed by the *consumer*
- data is passed from the producer to the consumer by being written into a *fixed-size buffer*
- the *consumer* should read data from the buffer only when there is unread data in the buffer (otherwise it should *sleep* until there is data)
- the *producer* should *not* write data to the buffer if the buffer is *full* (it should instead *sleep* until there is free space again)

The Producer-Consumer Problem (contd.)

For simplicity, we assume the data consists of *fixed-length records*, and the buffer can hold a fixed, integer number of records.

There are several common variations on this basic problem:

- there are *multiple* producers and/or multiple consumers rather than just one of each
- an unlimited number of records can be stored (e.g., by making the buffer a **linked list**)
- the data units are of variable size

The Producer-Consumer Problem (contd.)

An example "solution" without using special mechanisms:

```
//Shared variables:
struct record buff[N];           //array of records to be used for queue
int count = 0;                   //number of records currently in buff

//Producer code:
struct record next_rec;
while (1) {
  next_rec = get_next_record();  //obtain next record--i.e., from TTY
  while (count == N);            //wait for buff to have free space
  enqueue(buff,next_rec);        //store new record in buff queue
  count++;                       //record that additional record available
}

//Consumer code:
struct record next_rec;
while (1) {
  while (count == 0);            //wait for buff to contain record(s)
  next_rec = dequeue(buff);      //get oldest record from buff queue
  count--;                       //record that a record was removed
  process_next_record(next_rec); //process the record appropriately
}
```


The Producer-Consumer Problem (contd.)

Unforunately, there are a few problems with this "solution:"

- the `enqueue()` and `dequeue()` calls are obviously *not atomic*, and if they are interleaved the `buff`-based queue can be corrupted (a *race condition*)
- the `while` loops checking `count` represent *busy waits*, so will waste CPU cycles and can lead to resource starvation issues
- even the increment/decrement operations are not guaranteed to be *atomic*, potentially leading to invalid values for `count` (also a *race condition*)

The Producer-Consumer Problem (contd.)

Fixing the problems requires that:

- the `enqueue()` and `dequeue()` calls be prevented from occuring concurrently—i.e., we must impose *mutual exclsion*
- the producer must be able to be *suspended* (put to *sleep*) if the queue is full and then awakened when this changes
- the consumer must be be *suspended* (put to *sleep*) if the queue is empty and then awakened when this changes
- `count` must be incremented/decremented *atomically* or have its value modified *mutually exclusively*

The Producer-Consumer Problem (contd.)

The standard solution is to use one *binary semaphore* (or *mutex*) and two *general semaphores*.

This requires that the operating system and programming language provide/support these mechanisms.

We will represent semaphores as follows:

- a semaphore will be a variable of *type* `semaphore`
- the *decrement* operation will be called `semwait()`
- the *increment* operation will be called `semsignal()`

The Producer-Consumer Problem (contd.)

An example solution using semaphores:

```
//Shared variables:
struct record buff[N];             //array of records to be used for queue
semaphore queue_op = 1;            //can the buff queue be modified
semaphore queue_avail = 0;         //number of available records in queue
semaphore queue_free = N;          //number of free records in queue

//Producer code:
struct record next_rec;
while (1) {
  next_rec = get_next_record();    //obtain next record--i.e., from TTY
  semwait(queue_free);             //wait for buff to have free space & --
  semwait(queue_op);               //wait to be able to operate on queue
  enqueue(buff,next_rec);          //store new record in buff queue
  semsignal(queue_op);             //signal can operate on queue
  semsignal(queue_avail); }        //signal record(s) available & ++

//Consumer code:
struct record next_rec;
while (1) {
  semwait(queue_avail);            //wait for buff to contain record(s) & --
  semwait(queue_op);               //wait to be able to operate on queue
  next_rec = dequeue(buff);        //get oldest record from buff queue
  semsignal(queue_op);             //signal can operate on queue
  semsignal(queue_free);           //signal queue has free record(s) & ++
  process_next_record(next_rec); } //process the record appropriately
```

Concurrent Computing 4: C & Concurrency

1. Introduction
2. Terminology
3. Synchronization
4. **C and Concurrency**
   - **atomicity in C**
   - **concurrency and language elements**
   - **debugging concurrent programs**
5. More Asynchrony
6. Event-Driven Programming

Atomicity in C

When doing concurrent programming in a **high-level programming languge** like C, it is important to understand that each single language statement can be compiled/translated into *multiple* **machine instructions** (which are what ultimately get executed).

While each *machine instruction* is **atomic**, because a single *C statement* may be translated into multiple machine instructions, C statements are generally *not atomic* (or at least they cannot be assumed to always be atomic).

Consider an `int` variable `i` and the *increment* statement: `i++;`

On a modern CPU architecture, this statement might be compiled as a *single* machine instruction, but it might be compiled as *multiple* machine instructions.

Atomicity in C (contd.)

Compilation choices will depend on factors such as:

- the CPU architecture (e.g., Pentium vs. SPARC vs. ARM)
- the compiler plus the selected compiler architecture and optimzation options
- the variable's storage class (static, auto, etc.)
- how variables are used in the program

Intel x86 CPUs support an *integer addition* instruction, `add`, that can perform binary additions with the following:

- two **registers**
- a register and a memory location
- a constant and a register
- a constant and a memory location

Atomicity in C (contd.)

The C increment statement `i++;` can be implemented by adding the constant `0x1` to either a register or a memory location containing variable `i`'s value, so the statement can be translated into a *single machine instruction*, e.g.:

```
add  $0x1,%eax  # add constant one to i stored in register EAX
```

However, this is not what will always be done by a compiler, because of factors like:

- adding with registers is much faster than adding with memory addresses
- x86 CPUs have a limited number of *registers*, so not all variables can be held in registers

Atomicity in C (contd.)

As a result, a single C increment statement could be implemented as multiple x86 instructions, e.g.:

```
mov  0x804961c,%eax  # copy i's value from memory to register EAX
add  $0x1,%eax       # add one to i's value in EAX
mov  %eax,0x804961c  # copy i's incremented value to memory
```

Here, the C increment statement is obviously *not atomic*.

If another thread is accessing the variable, a *race condition* can result: only one or two of the above instructions are executed in a thread, that thread is suspended, a second thread runs and modifies `i`'s memory, that second thread is suspended, and the remaining of the above instructions are executed.

Atomicity in C (contd.)

The question of atomicity of C statements and *even individual machine instructions* becomes more complicated in the case of computers with multiple CPUs/cores!

In such systems, multiple threads may literally be running at the very same time, potentially trying to access/modify exactly the same memory locations.

Clearly, it is risky to assume that C statements even as simple as incrementing an `int` are atomic operations.

Instead, one should make use of **synchronization mechanisms** to ensure that race conditions cannot occur.

Atomicity in C (contd.)

While use of such mechanisms will tend to *slow* program execution, a program that has the potential of producing incorrect results is an *incorrect/buggy program*!

Old student programmer joke:
programmer1: "Hah, my program runs twice as fast as yours."
programmer2: "But mine always produces the correct answer!"

Question: Whose program do you want to use?

C and Concurrency

Prior to C11, the C language did not itself support concurrent programming.

Instead, *system calls* had to be used to create and control concurrent processes or OS threads.

C11 added support for writing *multithreaded* programs:

- threads control: `thrd_create()`, `thrd_join()`, etc.
- mutexes: `mtx_init()`, `mtx_trylock()`, etc.
- condition variables: `cnd_init()`, `cnd_signal()`, etc.
- thread-specific storage: `tss_create()`, `tss_set()`, etc.
- atomic data types: `atomic_int`, `ATOMIC_VAR_INIT()`, etc.
- fences: `atomic_thread_fence()`, etc.

Concurrency and C Language Elements

C99 and earlier versions do have some language elements that are relevant to concurrent programming:

- the `volatile` declaration keyword
- the `sig_atomic_t` data type
- **sequence points**
- **inline assembly**

volatile

The C keyword **volatile** is a **type qualifier**.

It can be used in declarations of variables and some other objects.

E.g., `volatile int flag;`

Basically, its use informs the compiler that the object's value might change independently of (local) code statements.

This means the compiler must *not* "optimize" code involving the object.

Its inclusion in C is to deal with:

- **memory-mapped devices**
- the `setjmp()`, `longjmp()`, etc. instructions
- `sig_atomic_t` variables in *signal handlers*

volatile (contd.)

`volatile`'s meaning appears to imply that it should also help with multithreaded programs, where one OS thread might modify a variable that another OS thread is also manipulating.

However, `volatile` is of almost *no value* in improving the safety of multithreaded programs:

- it has zero effect on the *atomicity* of operations on variables
- it has zero effect on the order of non-`volatile` variables
- while it does prevent optimizations to the `volatile` variables, one would have to declare every variable in a thread as `volatile` to help safety, which corresponds to using no optimization

sig_atomic_t

The C *data type* `sig_atomic_t` is a special *integer* type that guarantees **atomic** *access and modification* operations.

Specifically, this means that you can:

- atomically evaluate a variable's value
- atomically set its value to a constant

Note that it does *not* guarantee that *incrementing* a variable will be atomic!

The **Gnu C library** (**glibc**) documentation says this:
"In practice, you can assume that int is atomic. You can also assume that pointer types are atomic; that is very convenient. Both of these assumptions are true on all of the machines that the GNU C library supports...."

volatile sig_atomic_t

Since standard *signal handlers* cannot be passed arguments, it is common to exchange information with a handler via *global variables*.

However, because the primary program may be interrupted at any point (*asynchronously*) to run a handler (and a handler can be interrupted to run another handler), global variables must be used carefully.

To provide as much safety as possible, global signal handler variables should generally be declared as: "`volatile sig_atomic_t`"

Sequence Points

A **sequence point** in a programming language is point at which all **side-effects** of previous evaluations shall be complete and *no* side effects of subsequent evaluations shall have taken place.

The C standard defines the sequence points in C programs—see Wikipedia's "Sequence point" entry if you want more details.

The importance of this for concurrent programming is that a C programmer must be aware that *sub-expressions* in statements may not be guaranteed to be executed in a particular order.

Sequence Points (contd.)

The C99 standard contains the following example:

```
int sum;
char *p;
sum = sum * 10 - '0' + (*p++ = getchar());
```

The expression statement is *grouped* as if written:

```
sum = (((sum * 10) - '0') + ((*(p++)) = (getchar())));
```

However the *execution order* can differ:

- the increment of `p` can occur at any time between the previous sequence point and the next sequence point (the ;)
- the call to `getchar()` can occur at any point prior to the need for its returned value

Inline Assembly Code

If ultimate control over operations at the machine level are required, it is possible to include **assembly code** statements in C programs.

GCC supports **inline assembly** via the `asm()` call.

Here is an example call from the GCC documentation:

```
asm ("cmoveq %1,%2,%[result]"
     : [result] "=r"(result)
     : "r" (test), "r"(new), "[result]"(old));
```

Debugging Concurrent Programs

Concurrent programs are subject to all the bugs/errors that occur in single-thread programs—*plus* a new set of bugs/errors (e.g., *race conditions*).

The parallel and asynchronous nature of operations in concurrent programs makes it significantly more difficult for humans to find errors in concurrent programs.

Unfortunately, the kind of *testing* that programmers are used to doing with single-thread programs is also *much less effective* with concurrent programs.

It is not enough to show that a concurrent program produces the correct output for every input (class), it must be shown to do this under *every possible interleaving of its threads*.

Debugging Concurrent Programs (contd.)

However, simply running a concurrent program *repeatedly* (even *many* times) *cannot guarantee* that every possible interleaving of thread operations will get tested.

The OS, number of CPUs/cores, and current load can have a big impact on which interleavings occur when testing.

On one machine certain orderings may be unlikely to occur, while on a different machine those orderings could be most likely.

Even different OS versions may perform differently: e.g., some Linux kernel versions run the parent first after a `fork()` while others run the child first.

It is not uncommon for student code to immediately fail when graded, despite the students claiming they tested it "many times."

Debugging Concurrent Programs (contd.)

The only way a programmer has of forcing certain orderings is by the (temporary) insertion of *synchronization primitives* or `sleep()` and related calls in strategic locations.

Obviously, this is going to be extremely tedious and time consuming.

Concurrent programs are also much more likely than single-thread programs to suffer from *"hangs"* (e.g., **deadlock**).

Tools such as **GDB**, **strace**, and **ltrace** will be critical to identify what calls are blocking when the hang occurs, and to trace the order of calls leading to the hang.

Of course, the particular ordering of calls that lead to the hang may occur infrequently and so may be difficult to replicate.

Standard Synchronization Problems

The difficulty of debugging and testing means that it is critical to design concurrent programs to be *correct from the start!*

Most concurrent programs will involve one or more of a set of standard **synchronization problems**.

There are many sources programmers can consult to find the **design patterns** that correctly solve each problem.

Patterns will specify the ordering of program operations and the application of synchronization mechanisms that are required to avoid race conditions and deadlock.

Thus, the first step in implementing a concurrent program is to identify which synchronization problems apply to the program.

Standard Synchronization Problems (contd.)

Among the standard synchronization problems are:

- **producers consumers** variations:
  - bounded/finite buffer version
  - unbounded/infinite buffer version
  - single vs. multiple producers
  - single vs. multiple consumers
- **dining philosophers**
- **readers writers**
- **cigarette smokers**
- **barbershop** (**sleeping barber**)
- **dining savages** (obviously named some time ago :)

Concurrent Computing 5: More Asynchrony

1. Introduction
2. Terminology
3. Synchronization
4. C and Concurrency
5. **More Asynchrony**
   - **other sources of asynchrony**
   - **signals and asynchrony**
   - **reentrant vs. thread-safe**
6. Event-Driven Programming

Other Sources of Asynchrony

The chief source of complexity in writing concurrent programs is the uncertainty (and variability) in how operations in concurrent threads of execution will end up being *interleaved*.

The most obvious cause for this uncertainty is the freedom that the *OS scheduler* has to suspend any thread at any point and start another thread executing.

We mentioned earlier that with a high-level programming language like C, uncertainty is compounded by the fact that even simple C statements may *not* be *atomic*, and the atomicity of a C statement can vary.

Other Sources of Asynchrony (contd.)

Unfortunately, there are even more reasons why the order in which individual program operations may occur or be interleaved is uncertain.

The most common additional sources of order uncertainty are:

- CPUs using a **superscalar architecture**
- **optimizing compilers**
- computers that contain *multiple CPUs/cores*

Other Sources: Superscalar Architecture

Superscalar architecture CPUs implement **instruction-level parallelism** within each single CPU/core.

That is, multiple machine-level instructions (or components of an instruction) are simultaneously being computed across multiple **functional units** in each core.

To obtain the utmost speed/efficiency from the functional units, most superscalar CPUs support **out-of-order execution**:
the execution of machine instructions can be *reordered* as long as **data dependencies** are not violated.

This means that even if we know the order of the machine instructions in a thread, we cannot know the order in which the CPU will actually execute the instructions.

Other Sources: Superscalar Arch. (contd.)

This leads to additional possible interleavings of the machine instructions that comprise a concurrent program.

Most modern CPUs use a superscalar architecture—including Pentium and Core CPUs.

Other Sources of Asynchrony (contd.)

An **optimizing compiler** is one that can adjust the **object code** it produces for a program so as to *minimize* ("optimize") one or more *costs*, such as execution speed, executable size.

The produced code must of course implement the required semantics of the program!

Optimizing compilers can adjust object code by reordering instructions, removing instructions, combining instructions, and so forth.

This means that the instructions in the executable may be very different in order and even structure from what would be expected based on the source code.

Other Sources: Compiler Optimization

For example, consider the C function fragment:

```
  ...
  i++;
  j++;
  i++;
  return i;
}
```

An optimizing compiler might produce code more like:

```
  ...
  i += 2;
  return i;
}
```


Other Sources: Compiler Optimization (contd.)

Compiler optimizations can result in reordered instructions in threads, and thus unexpected interleavings of instructions in concurrent programs.

However, it is possible to **disassemble** the object code produced by an optimzing compiler to see the resulting machine-level code.

For example, you could use `objdump`:
`objdump -S prog`
(compile with the `-g` flag)

The optimization option for GCC is `-O`:
`gcc -Wall -g -O2 -oprog program.c`

Other Sources: Multiple CPUs

Modern computers often contain multiple CPUs and/or multiple cores, and such machines can provide true computation parallelism with concurrent programs.

However, because these machines can have multiple threads running at exactly the same time, the programmer is faced with an even larger set of possible relative orderings of the operations in different threads.

For example, every single machine instruction in every thread could end up being interleaved (if there are more CPUs/cores than threads).

Thus, the use of multi-CPU/core systems greatly increases the likelihood that an error-producing order of operations will occur in concurrent programs that contain a race condition bug.

Other Sources: Multiple CPUs (contd.)

Note that in modern multi-CPU/core computers (as apposed to **clusters**), the CPUs/cores all share the same memory (RAM).

While it appears that this architecture could lead to race conditions when multiple units all try to access memory at once, each unit will lock the **memory bus** when accessing memory, eliminating this possibility.

Programmers do not have to worry about race conditions occurring below the level of control they have over program operations.

Signals and Asynchrony/Atomicity

While we are focusing on issues in concurrent programming here, similar problems involving *asynchrony and atomicity* of operations can arise even in *single-thread programs*.

In such programs, the source of these issues is **signals**.

Signals are a type of **software interrupt** that informs a process that some **event** has occurred.

Signals that are generated by the process itself may be received **synchronously**—e.g., a `SIGPFE` due to an instruction that attempts *division by zero*.

Most signals, though, can be received by a program at any point during its execution—i.e., they can be received **asynchronously**.

Signals and Asynchrony/Atomicity (contd.)

Examples of signals that can be *received asynchronously* even in a *single process* program include those from:

- alarms/timers
- the user (e.g., `^-c` on keyboard or `kill` command)
- **signal-driven I/O**

When a program has **registered** a **handler** for one or more signals, the primary program can be interrupted at any point, the handler run, and then the primary program resumed.

Such programs must be designed so that interruptions can occur at any point (while the handler is registered) without affecting the correctness of the results.

Handlers and Reentrant Functions

To be safe and reliable, signal handlers must be **reentrant**.

A handler is **nonreentrant** if it accesses/modifies data used by other program components or other invocations of itself.

This can result in *race conditions*: incorrect results can be produced if the handler is invoked at particular points in the primary code or is invoked multiple times.

To be **reentrant**, a function must modify only data that is *local* (unique) to each *execution instance* of the function.

Reentrant functions cannot modify (or return pointers to) any data that is shared among multiple calls to the function, such as global or `static` variables, files, and so forth.

Handlers and Reentrant Functions

Signal handlers may be nonreentrant because the they call *library functions* that are nonreentrant.

SUS/POSIX refer to *reentrant* functions as **async-signal-safe functions**.

SUS/POSIX define a set of functions that must be safe to call in signal handlers, known as the **async-signal-safe functions** (see "`man 7 signal`").

Some *C library functions* are *not reentrant* because they modify statically allocated data (e.g., `fgetc()`, `malloc()`) or because they return pointers to statically allocated memory (e.g., `strtok()`, `gethostbyname()`).

Most of the *C standard I/O library* functions are *not async-signal-safe functions!*

Reentrant vs. Thread-Safe Functions (contd.)

The concept of *reentrant functions* is different from that of **thread-safe functions** (see "`man 7 pthreads`").

Many reentrant functions are also **thread-safe function**, but this is not necessarily the case.

E.g., if a reentrant function reads from a file that may be modified by another thread, the reentrant function is *not* thread-safe.

Being thread-safe does *not imply anything* about a function being reentrant!

A *nonreentrant* function can be made *thread-safe* by using a *mutex* to limit simultaneous execution of the function or limit simultaneous access to shared resources (like global variables).

Reentrant vs. Thread-Safe Functions (contd.)

SUS/POSIX library functions/syscalls that are *not thread-safe* are also *not async-signal-safe*, such as: `asctime()`, `ftw()`, `gethostbyname()`, `rand()`, `strtok()`, etc.

However, there are a number of *thread-safe functions* that are *not async-signal-safe*, such as: `fgetc()`, `fopen()`, `malloc()`, `mount()`, etc.

Most of the latter are C library functions, but there are a few system calls as well.

Concurrent Computing 6: Event-Driven Prog.

1. Introduction
2. Terminology
3. Synchronization
4. C and Concurrency
5. More Asynchrony
6. **Event-Driven Programming**

Event-Driven Programming

Event-driven (or **event-based**) **programming** is a *programming paradigm* (style) where program functionality is structured in terms of a set of **event handler** routines and **control flow** is determined by the sequence in which the relevant **events** occur.

Events can include a wide variety of things: keystrokes, mouse clicks, "messages" becoming available on a pipe/socket, changes in variable values, changes in the state of files in a directory, outputs from sensors, etc.

Event handlers are *subroutines/functions/methods* that perform the appropriate functionality whenever their associated event occurs.

Event-driven programming is widely used for **graphical user interfaces** (**GUI**s) as well as in **computer gaming** (multiple game AIs must respond to player and other AIs actions).

Event-Driven Programming (contd.)

The reason for discussing event-driven programming here is that it can be advantageous to use concurrent programming techniques to implement event-driven programs.

Conceptually, an event-driven program can be described as:

```
while (1) {
  //Wait for some event(s) to occur:
  while ((events = check_for_events()) == NULL);

  //If multiple events, decide which to handle next:
  event = select_event_to_handle(events);

  //Invoke the correct event handler for the selected event:
  dispatch_to_handler(event);
}
```


Event-Driven Programming (contd.)

The above code has the check for whether an event(s) has occurred inside a `while` loop that will repeat continuously until an event occurs.

While this is conceptually what happens in event-driven programming, a direct implementation of this sort of **busy waiting** or **polling** will be used only if absolutely necessary, since it wastes CPU cycles.

Instead, waiting/checking for an event to occur will typically be implemented using one or more the following OS capabilities:

- **I/O blocking**
- **I/O multiplexing** (using `select()`, `poll()`, etc.)
- **suspending** a process until a **signal** is delivered

Event-Driven Programming (contd.)

Perhaps the most obvious advantage of a concurrent program implementation of an event-driven program is that it could take advantage of a computer with multiple CPUs/cores by allowing event checking to continue to run in parallel with event handler calls.

This is particularly useful for maintaining fast *response time* when events may take signficant time to handle (limiting the time a handler hogs the CPU would be quite complicated in a single-thread program).

A straightforward way to add concurrency to the basic event-driven program logic would be by having each handler call get run in a new OS thread or process.

Event-Driven Programming (contd.)

However, since there is some *time cost* to creating new threads (especially new processes), it can make more sense to create a separate thread for each event handler when the program begins running, and simply *suspend* each thread until there is a relevant event that needs to be handled.

In fact, instead of continuing to have a central thread that is responsible for detecting events and doing handler dispatching, the logic for detecting the relevant event(s) can be distributed to each handler thread.

The event-driven program is now a set of threads each of which has the following logic:

```
while (1) {
  while ((event = check_for_handler_event()) == NULL);
  handler(event);
}
```


Event-Driven Programming (contd.)

As noted above, each thread will generally make use of OS capabilities to avoid busy-waiting for their event to occur.

In fact, doing so can be more critical with this design, since there would be multiple threads wasting CPU cycles, which could delay other threads from being scheduled to be run as soon as their event has occurred.