C++ tutorial

- Assume you've seen the standard C++ class (very similar to Java):

```
class MyCls: public BaseCls {
  private:
    int m_privdata;
    void f1();
  protected:
    int f2();
    char m_procdata;
  public:
    char * pubfunc1();
    int m_pubdata;
};
```

private and protected inheritance?

- Almost never used
- When preceding the name of a base class, the **private** keyword specifies that the public and protected members of the base class are private members of the derived class.
- When preceding the name of a base class, the **protected** keyword specifies that the public and protected members of the base class are protected members of the derived class.

Calling functions in base classes

```
class Window {
  public:
    void draw();
};
class Button: public Window {
  public:
    void draw();
};
How do you call Window's draw() from
  within Button's draw()?
```

Virtual functions

```
class Mammal {
  public:
      void move() { printf("mammal move\n");};
      //don't really inline virtual funcs. Like I have here.
      virtual void speak() { printf("mammal speak\n");};
};
Class Dog: public Mammal {
  public:
      void move() { printf("dog move\n"); };
      void speak() { printf("woof\n"); };
};
int main(void)
{
   Mammal * pDog = new Dog;
   pDog->move();
   pDog->speak();
   return 0;
}
```

How virtual functions (roughly) work

- When a virtual function is created, the compiler builds a *virtual function table* – has pointer to each virtual function
- One virtual function table per type
- Each instance of a class has virtual table pointers (vptr) that point to the vtable
- The vptr is adjusted in the constructors to point to the "correct" function
- Small overhead associated with virtual functions.

Destructors should always be virtual:

```
class A {
  public:
    ~A() {};
    void func();
};
class B: public A {
  private:
    char * data;
  public:
    ~B() { if(data) free(data); };
};
void main(void)
{
  A * a = new B;
  delete a;  //oops... ~B() not called!!
}
```

Pure virtual functions

```
class Mammal {
  public:
    virtual void speak()=0;
};
```

- Speak is a *pure virtual* function
- Can't instantiate a class with pure virtuals
- Derived classes ***must*** implement the pure virtual function or else it can't be instantiated.

The static keyword in classes

- In C++, when modifying a data member in a class declaration, the **static** keyword specifies that one copy of the member is shared by all the instances of the class. When modifying a member function in a class declaration, the **static** keyword specifies that the function accesses only static members.
- static member functions may be called without an instance
- static data needs to be delcared in the implementation like a global.

Concurrency

Haviland – Ch. 8.3.3

Concurrency

- The two key concepts driving computer systems and applications are
  - communication: the conveying of information from one entity to another
  - concurrency: the sharing of resources in the same time frame
- Concurrency can exist in a single processor as well as in a multiprocessor system
- Managing concurrency is difficult, as execution behaviour is not always reproducible.

Concurrency Example

- Program a:

```
#!/usr/bin/sh
count=1
while [ $count -le 20 ]
do
   echo -n "a"
   count=`expr $count + 1`
done
```

- Program b

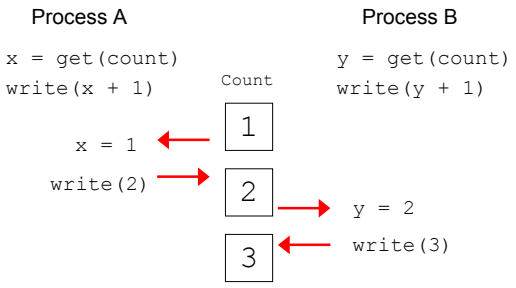
```
#!/usr/bin/sh
count=1
while [ $count -le 20 ]
do
   echo -n "b"
   count=`expr $count + 1`
done
```

- When run sequentially `(a; b)` output is sequential.
- When run concurrently `(a&; b&)` output is interspersed and different from run to run.

Race conditions

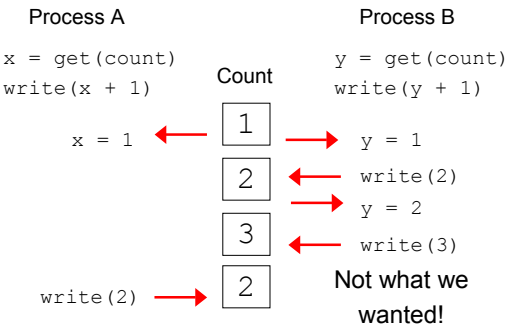
- A race condition occurs when multiple processes are trying to do something with shared data and the final outcome depends on the order in which the processes run.
- E.g., If any code after a fork depends on whether the parent or child runs first.
- A parent process can call wait() to wait for termination (may block)
- A child process can wait for parent to terminate by polling (wasteful)
- Standard solution is to use signals.

Example 1



The value of count is what we expect.

Example 2



Not what we wanted!

Example: Race Conditions

Try running several instances of this

```
#!/bin/sh
c=1
while [ $c -le 10 ]
do
   sd=`cat sharedData`
   sd=`expr $sd + 1`
   echo $sd > sharedData
   c=`expr $c + 1`
   echo d = $sd
done
#file sharedData must exist and hold
#one integer
```

Producer/Consumer Problem

- Simple example: `who | wc -l`
- Both the writing process (`who`) and the reading process (`wc`) of a pipeline execute concurrently.
- A pipe is usually implemented as an internal OS buffer.
- It is a resource that is concurrently accessed by the reader and the writer, so it must be managed carefully.

Producer/Consumer

- consumer should be blocked when buffer is empty
- producer should be blocked when buffer is full
- producer and consumer should run independently as far as buffer capacity and contents permit
- producer and consumer should never be updating the buffer at the same instant (otherwise data integrity cannot be guaranteed)
- producer/consumer is a harder problem if there are more than one consumer and/or more than one producer.

Protecting shared resources

- Programs that manage shared resources must protect the integrity of the shared resources.
- Operations that modify the shared resource are called critical sections.
- Critical section must be executed in a mutually exclusive manner.
- Semaphores are commonly used to protect critical sections.

Semaphores

- Code that modifies shared data usually has the following parts:
  - **Entry section**: The code that requests permission to modify the shared data.
  - **Critical Section**: The code that modifies the shared variable.
  - **Exit Section**: The code that releases access to the shared data.
  - **Remainder**: The remaining code.

Critical Sections

- Problem: How to execute critical sections in a fair, symmetric manner?
- Solutions must satisfy the following:
  - Mutual Exclusion: At most one process is in its critical section at a time.
  - Progress: If no process is executing its critical section a process that wants to enter can get in
  - Bounded Waiting: No process is postponed indefinitely.
- An atomic operation is an operation that, once started, completes in a logical indivisible way. Most solutions to the critical section problem rely on the existence of atomic operations.

General structure Flawed Solution 1

// process i	// process i

do {
entry section
critical section
exit section
remainder section
} while(1);
	do {
	while(turn != i);
	critical section
	turn = j;
	remainder section
	} while(1);

- **Problems?**
  - strict alternation (violates progress rule)

Flawed Solution 2

```
boolean flag[2];
do {
    flag[i] = true;
    while(flag[ j ]);
    critical section
    flag[i] = false;
    remainder section
} while(1);
```

- Problems?
  - mutual exclusion not satisfied
  - Consider:
  - t0: P0 sets flag[0] to true
  - t1: P1 sets flag[1] to true
  - Now both are looping forever.

A correct solution

```
do{
    flag[i] = true;
    turn = j;
    while(flag[ j] && turn == j);
    critical section
    flag[i] = false;
    remainder section
} while(1);
```

- flag[0] and flag[1] could both be true, but turn can have only one value. Thus only one process can enter critical section
- progress is preserved because process i can only be spinning in the loop if turn == j and flag[j] == true. When process j exits the critical section, it will set flag[j] to false.

Problems with the software solution

- The biggest problem with the software solution to the critical section problem is that a process waiting to get into the critical region is busy-waiting.
- A solution is to take advantage of atomic hardware instructions and use them to build up more general synchronization mechanisms like semaphores.

Semaphores

- A semaphore is an integer variable with two atomic operations: acquire and release.
  - acquire also called wait, down, P, and lock
  - release also called signal, up, V, and unlock
- A process that executes an acquire on a semaphore variable S, cannot proceed until S is positive. It then decrements S.
- The release operation increments the value of the semaphore variable.

Flawed implementation of semaphores

```
void acquire(int *s)
{
  while(*s <= 0);
  (*s)--;
}
```

```
void release(int *s)
{
  (*s)++;
}
```

Problems:

- busy-waiting is inefficient
- does not guarantee bounded waiting
- ++ and -- operations are not necessarily atomic

Unix (System V) semaphores

- The following pseudo-code protects a critical section:

```
acquire(&s);
/* critical section */
release( &s );
/* remainder section */
```

- What happens if S is initially 0? or 8?
- We will use System V semaphores to implement `acquire` and `release`.

Unix System V Semaphores

- Semaphore structures are created and stored in the OS kernel, so unrelated processes can use them.
- A semaphore created and associated with a key, similar to a file handle.
- One System V semaphore is actually an array of semaphores. (We will only use one element of the array in discussions in this class.)

Semaphores

- Three operations on a semaphore
  - `semget` – create or gain access to a semaphore
  - `semctl` – get or set the value of a semaphore, change permissions, or remove a semaphore.
  - `semop` – perform semaphore operation

Initializing Semaphores

- `semget()` initializes or gains access to a semaphore

```
int
semget(key_t key, int nsems, int semflg);
```

- `key` – access value associated with the semaphore ID
- `nsems` – number of elements in a semaphore array
- `semflg` – access permission and creation control flags
- returns the semaphore ID

What's a key?

- *Keys* are used to identify IPC (inter-process communication) objects
- IPC objects vary: message queues, shared memory, semaphores
- Like a file name, but simpler – just a number
- Can just pick one, but note keys can conflict!
- To get a unique key, use:

```
key_t ftok(const char *path,int id)
```

More about the semflg

- Contains the usual octal permissions
- Two additional flags are of relevance:
  - IPC_CREAT – create the IPC object if it doesn't already exist
  - IPC_EXCL – fail if the IPC object already exists (returns –1), sets errno to EEXIST
- Combine everything with bitwise OR

Controlling Semaphores

- `semctl()` changes permission and other characteristics of a semaphore set.

```
int semctl(int semid, int semnum, int
  cmd, union semnum arg);
```

- must be called with a valid `semid`.
- `semnum` selects a semaphore from an array by its index
- `cmd` is one of a number flags to specify an opertion
- `union semun arg` is optional depending on the operation. If required the type must be explicitly declared by the user program.

`semctl()` operations

```
union semun {
  int val;
  struct semid_ds * stat;
  unsigned short * array;
};
```

- GETVAL – return the value of a single semphore
- SETVAL – set value of a single semaphore (arg is taken as arg.val, an int)
- GETPID – return the PID of the process that performed the last operation on the semaphore
- GETNCNT – return the number of processes waiting for the semaphore to go to positive.
- IPC_STAT, IPC_SET – retrieves/sets info in stat
- IPC_RMID – remove the specified semaphore set.

Code thus far…

```
int initsem(key_t semkey)
{
  int status=0, semid;
  if((semid = semget(semkey,1,0600|IPC_CREAT|IPC_EXCL))==-1) {
    if(errno==EEXIST) semid=semget(semkey,1,0);
  } else {
    semun arg;
    arg.val=1;  //the initial value of the semaphore
    status = semctl(semid,0,SETVAL,arg);
    if(status==-1) {
      //handle error...
    }
  }
  return semid;
}
```

Semaphore Operations

```
int semop(int semid, struct sembuf
  *sops, size_t nsops);
```

- `semid` is the semaphore ID returned by `semget()`
- `sops` is a pointer to an array of structures containing information about a semaphore operation:

```
struct sembuf {
  ushort_t sem_num; /*semaphore number */
  short sem_op; /* semaphore operation */
  short sem_flg; /* operation flags */
};
```

- `nsops` specifies the length of the array
- The array of operations execute atomically

semop()

- The operation to be performed is determined by the value of `sem_op(int sembuf)`:
  - `sembuf.sem_op > 0` – increment semaphore by `sembuf.sem_op` immediately
  - `sembuf.sem_op = 0` – wait for the semaphore to reach 0
  - `sembuf.sem_op < 0` to be discussed next

semop() (cont'd)

- Control flags:

  `IPC_NOWAIT` –makes the function return without changing the semaphore if the operation cannot be performed.

  `SEM_UNDO` – Allows operation to be undone when process exits.

semop()

- `sembuf.sem_op < 0` – decrement semaphore by `ABS(sembuf.sem_op)`
- An attempt to set a semaphore to a value less than zero fails or blocks depending on whether `IPC_NOWAIT` is in effect.
- Pseudo code:

```
if(semval >= ABS(sem_op)) {
  semval= semval - ABS(sem_op)
}
else{
  if(IPC_NOWAIT set) return -1
  else wait until semval >= ABS(sem_op) and decrment
}
```

lock()

```
int lock(int semid)
{
  struct sembuf p_buf;
  p_buf.sem_num = 0;
  p_buf.sem_op = -1;
  p_buf.sem_flg = SEM_UNDO;
  return !(semop(semid,&p_buf,1)==-1);
}
```

Cleaning up

- Semaphores are a system-wide resource
- You must delete semaphores when done with them
  - `semctl()` function and `IPC_RMID`
  - `ipcs` and `ipcrm` programs