Today

- With all these threads running around sharing the same address space (i.e., memory), how do we keep them from mangling each other?
- With specific synchronization primitives
Do We Really Need to Synchronize?

- Yes and no
- You can build a system and run a long long time before hitting a synchronization bug. And maybe your application or user doesn’t care.
- But for truly robust systems you need to synchronize your data structures to ensure their consistency

A very simple example

- Assume a global queue with two fields a flink and a blink
- Here is some sample code to add to the queue
  ```c
  LIST_ENTRY Queue;
  NewEntry = new(…)
  NewEntry->flink = Queue.flink;
  NewEntry->blink = &Queue;
  NewEntry->flink->blink = NewEntry;
  NewEntry->blink->flink = NewEntry;
  ```
- Now let threads execute the above code at the same time
- Where’s the problem?
- The problem goes all the way down to the machine instructions
Synchronization

- Basic Problem Statement:
  If two concurrent processes are accessing a shared variable, and that variable is read, modified, and written by those processes, then the variable must be controlled to avoid erroneous behavior.
- Even simple push and pop stack operations need synchronization

```
Push(s, I) { s->stack[++(s->index)] = I; }
Pop(s) { return (s->stack[(s->index)--]); }
```

Even ignoring stack limit tests these routines need synchronization in a multi-threaded environment

Another Example (the ATM)

- Suppose each cash machine transaction is controlled by a separate process, and the withdraw code is:

```
cur_balance=get_balance (acct_ID)
withdraw_amt=read_amount_from_ATM()
if withdraw_amt>curr_balance then error
curr_balance=curr_balance - withdraw_amt
put_balance (act_ID,curr_balance)
deliver_bucks(withdraw_amt)
```
- Now, suppose that you and your s.o. share an account. You each to to separate cash machines and withdraw $100 from your balance of $1000.
ATM Example Continued

you: curr_balance=get_balance(acct_ID)
you: withdraw_amt=read_amount()
you: curr_balance=curr_balance-withdraw_amt
so: curr_balance=get_balance(acct_ID)  \[\text{context switch}\]
so: withdraw_amt=read-amount()
so: curr_balance=curr_balance-withdraw_amt
so: put_balance(acct_ID,curr_balance)
so: deliver_bucks(withdraw_amt)
you: put_balance(acct_ID,curr_balance)
you: deliver_bucks(withdraw_amt)

• What happens and why?

Problems

• A problem exists because a shared data item (curr_balance) was accessed without control by processes that read, modified, and then rewrote that data.
• We need ways to control access to shared variables.
Ways to Solve The Synchronization Problem

- Only have one thread do everything
- Semaphores (a classic text book solution and the one we cover first)
- Spinlocks
- Interlocked Operations
- Mutexes
- Events
- “EResource” an NT’ism that I’m particularly fond of

Where Can We Actually Use Synchronization?

- Both in the kernel and in user mode
  - A good thing too because we need it in both places
  - In the kernel most any trick is available for us to use
  - In user mode our choices are a bit more limited
- Some synchronization methods are kernel mode only and some can be used in both modes.
- Kernel mode only because of some tricks use the protected instruction set
Semaphores

• Dijkstra, in the THE system, defined a type of variable and two synchronization operations that can be used to control access to critical sections.
• First, what is a critical section?
• Dijkstra defined a semaphore as a synchronization variable that is manipulated atomically through operations signal(s) (a V operation) and wait(s) (a P operation).
• To access a critical section, you must:
  \[\text{wait}(s); \quad // \text{wait until semaphore is available} \]
  \[<\text{critical section code}>\]
  \[\text{signal}(s); \quad // \text{signal others to enter}\]

Semaphore Implementations

• Associated with each semaphore is a count indicating the state of the semaphore
  1. > 0 means the semaphore is free or available
  2. <= 0 means the semaphore is taken or in use
  3. < 0 means there is a thread waiting for the semaphore (its absolute value is the number of waiters)
• Also associated with each semaphore is a queue of waiting threads.
• If you execute \textit{wait} and the semaphore is free, you continue; if not, you block on the waiting queue.
• A \textit{signal} unblocks a thread if it’s waiting.
Semaphore Operations

typedef struct _SEMAPHORE {
    int Value;
    List of waiting threads WaitList;
} SEMAPHORE, *PSEMAPHORE;

VOID Wait( PSEMAHPORE s ) {
    s->Value = s->Value - 1;
    if (s->Value < 0) {
        add this thread to s->WaitList;
        block current thread;
    }
}

VOID Signal( PSEMAPHORE s ) {
    s->Value = s->Value + 1;
    if (s->Value <= 0) {
        remove a thread T from s->WaitList;
        wakeup T;
    }
}

Example: Reader/Writer Problem

• Basic Problem:
  – An object is shared among several threads, some which only read it, and some which write it.
  – We can allow multiple readers at a time, but only one writer at a time.
  – How do we control access to the object to permit this protocol?
A Simplistic Reader/Writer Semaphore Solution

```c
SEMAPHORE wrt;    // control entry to a writer or first reader
SEMAPHORE semap;  // controls access to readcount
int readcount;    // number of active readers

write process:
  wait(wrt);    // any writers or readers?
  <perform write operation>
  signal(wrt);  // allow others

read process:
  wait(semap);  // ensure exclusion
  readcount = readcount + 1;  // one more reader
  if (readcount = 1) { wait(wrt); } // we’re the first
  signal(semap);
  <perform reading>
  wait(semap);  // ensure exclusion
  readcount = readcount – 1;  // one fewer reader
  if (readcount = 0) { signal(wrt); } // no more readers
  signal(semap)
```

Reader/Writer Solution Notes

- Note that:
  1. The first reader blocks if there is a writer; any other readers who try to enter will then block on `semap`.
  2. Once a writer exists, all readers will fall through.
  3. The last reader to exit signals a waiting writer.
  4. When a writer exits, if there is both a reader and writer waiting, which goes next depends on the scheduler.
Semaphore Types

• In general, there are two types of semaphores based on its initial value
  – A binary semaphore guarantees mutually exclusive access to a resource (only one entry). The binary semaphore is initialized to 1. This is also called a mutex semaphore, but not everything you hear called a mutex is implemented as a semaphore
  – A counted semaphore represents a resource with many units available (as indicated by the count to which it is initialized). A counted semaphore lets a thread pass as long as more instances are available.

Example: Bounded Buffer Problem

• The Problem:
  There is a buffer shared by producer processes, which insert into it, and consumer processes, which remove from it.

  The processes are concurrent, so we must control their access to the (shared) variables that describe the state of the buffer.
Simple Bounded Buffer Semaphore Solution

```c
SEMAPHORE mutex;     // mutual exclusion to shared data
SEMAPHORE empty = n; // count of empty buffers
SEMAPHORE full = 0;  // count of full buffers

producer:
    wait(empty);   // one fewer buffer, block if none available
    wait(mutex);   // get access to pointers
    <add item to buffer>
    signal(mutex); // done with pointers
    signal(full);  // note one more full buffer

c consumer:
    wait(full);    // wait until there's a full buffer
    wait(mutex);   // get access to pointers
    <remove item from buffer>
    signal(mutex); // done with pointers
    signal(empty); // note there's an empty buffer
    <use the item>
```

Things to Remember About Semaphores

- A very common synchronization primitive
- Two main elements a count and a list of waiters
- Two types counted and binary semaphore
- Other synchronization operations can be built on top of semaphores
Next Time

- Semaphores are great and used all over the place, but it’s not the only game in town
- Next time we’ll look at a few other useful synchronization primitives