Synchronization
Synchronization

• Basic Problem:
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.
ATM Example

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

  \[
  \begin{align*}
  \text{current\_balance} &= \text{get\_balance}(\text{acct\_ID}) \\
  \text{curr\_balance} &= \text{curr\_balance} - \text{withdraw\_amt} \\
  \text{put\_balance}(\text{act\_ID},\text{curr\_balance}) \\
  \text{deliver\_bucks}(\text{withdraw\_amt})
  \end{align*}
  \]

- Now, suppose that you and your SO share an account. You each to separate cash machines and withdraw $100 from your balance of $1000.
ATM Example

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)
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?
• Why does it happen?
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.
Critical Sections

• The Too Much Milk or the bank balance problem illustrates the difficulty of coordinating processes
  – Race conditions
  – Deadlock / Livelock
  – Starvation

• Atomic loads and stores make synchronization difficult (but not impossible)
  – For two processes, the simplest correct solution is asymmetric
  – For three or more processes, the bakery (or post office) algorithm requires auxiliary data structures
Criteria for Critical Sections

• A good solution to the critical section problem would have three properties
  – Mutual exclusion
  – Progress
  – Bounded Waiting

• Cannot make any assumptions about the relative speeds of processes
Hardware Primitives

- Modern hardware provides better atomic operations than load/store
  - Test-And-Set (TAS)
  - Swap
  - Compare-And-Swap (CAS)
  - Load-Linked & Store-Conditional (LL/SC)
void TAS(int *location) {
    int oldvalue = *location;
    *location = 1;
    return oldvalue;
}

The entire function is Atomic
You could implement this on hardware by keeping the bus locked for both a load and a store transaction.

• Simple primitive
• Makes programming critical sections easy
Critical Sections with TAS

- While(TAS(&lock) == 1) {
  /* do nothing */
}

  critical section
Lock = 0;

Problem: busy-waiting for the entire duration of the critical section
**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**.
- A **semaphore** is a **variable** that is manipulated **atomically** through operations $V(s)$ (signal) and $P(s)$ (wait).
- To access a critical section, you must:
  
  $P(s)$ ; wait until semaphore is available; also known as wait()

  <critical section code>

  $V(s)$ ; signal others to enter; also known as signal()
Semaphores

- Associated with each semaphore is a queue of waiting processes.
- If you execute `wait(s)` and the semaphore is free, you continue; if not, you block on the waiting queue.
- A `signal(s)` unblocks a process if it’s waiting.
Spinlocks

typedef struct spinlock {
    int lock:
} Spinlock;

void acquire(Spinlock *s) {
    while(test_and_set(s->lock) == 1)
        /* do nothing, or yield */;
}

void release(Spinlock *s) {
    atomicclear(s->lock);
}

Signal and Wait must be atomic
typedef struct semaphore {
    int value:
        ProcessList L;
} Semaphore;

void P(Semaphore *S) {
    S->value = S->value - 1;
    if (S.value < 0) {
        add this process to S.L;
        block(&S->lock);
    }
}

void V(S) {
    S->value = S->value + 1;
    if (S->value <= 0) {
        remove a process P from S.L;
        wakeup P
    }
}
typedef struct semaphore {
    int lock;
    int value;
    ProcessList L;
} Semaphore;

void P(Semaphore *S) {
    while(test_and_set(&S->lock) == 1) /* do nothing */;
    S->value = S->value - 1;
    if (S.value < 0) {
        add this process to S.L;
        atomic_clear_and_block(&S->lock);
    } else
        atomicclear(&S->lock);
}

void V(S) {
    while(test_and_set(&S->lock) == 1) /* do nothing */;
    S->value = S->value + 1;
    if (S->value <= 0) {
        remove a process P from S.L;
        wakeup P
    }
    atomicclear(&S->lock);
}
Semaphore Types

• In general, there are two types of semaphores:
  – a **mutex** semaphore guarantees mutually exclusive access to a resource (only one entry). The mutex sema is initialized to 1.
  – A **counting** semaphore represents a resource with many units available (as indicated by the count to which it is initialized). A counting semaphore lets a process pass as long as more instances are available.
Example: Mutual exclusion with Semaphores

Semaphore *traysema = semacreate(1);

void cook() {
    while(TRUE) {
        Burger *burger = makeburger();
        P(traysema);
        placeItemOnTray(burger);
        V(traysema);
    }
}

void customer() {
    while(TRUE) {
        Burger *burger;

        P(traysema);
        burger = grabItemFromTray(burger);
        V(traysema);
    }
}

2/18/2002
Example: Waiting for a condition

Semaphore *sema = semacreate(0);

void Bob() {
    while(TRUE) {
        /* Block until Abe is done with his construction */
        P(sema);
        removeCarFromAssemblyLine();
        ...
    }
}

void Abe() {
    while(TRUE) {
        /* Prepare a chassis – this will take a while */
        prepareChassis();
        placeChassisOnAssemblyLine();
        V(sema);
    }
}
Example: Mutual exclusion with Semaphores

Semaphore *traysema = semacreate(1);
Semaphore *trayfull = semacreate(0);
Semaphore *trayempty = semacreate(1);

void cook() {
    while(TRUE) {
        Burger *burger;
        P(trayempty);
        burger = makeburger();
        P(traysema);
        placeItemOnTray(burger);
        V(traysema);
        V(trayfull);
    }
}

void customer() {
    while(TRUE) {
        Burger *burger;
        P(trayfull);
        P(traysema);
        burger = grabItemFromTray(burger);
        V(traysema);
        V(trayempty);
    }
}
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.
Bounded Buffer Sema Implementation

```plaintext
var mutex: semaphore = 1 ; mutual exclusion to shared data
  empty: semaphore = n ; count of empty buffers (all empty to start)
  full: semaphore = 0 ; count of full buffers (none full to start)

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

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>
```
Dining Philosophers

Semaphore *chopsticks[NCHOPSTICKS];

Initialize() {
    for(l=0; l<NCHOPSTICKS; ++l) {
        chopstick[l] = semacreate(1);
    }
}

Philosopher() {
    while(TRUE) {
        P(chopstick[i]);
        P(chopstick[(i+1) % NCHOPSTICKS]);
        eat();
        V(chopstick[i]);
        V(chopstick[(i+1) % NCHOPSTICKS]);
        think();
    }
}
Dining Philosophers

• Deadlock!
• Allow at most N-1 philosophers at the table
• Pick up chopsticks in a global critical region
• Odd philosophers pick left, then right, even philosophers pick right, then left, chopstick
Example: Readers/Writers Problem

• Basic Problem:
  – an object is shared among several processes, 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?
Readers/Writers Sema Implementation

var mutex: semaphore ; controls access to readcount
  wrt: semaphore ; control entry to a writer or first reader
  readcount: integer ; number of readers

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

read process:
  wait(mutex) ; ensure exclusion
  readcount = readcount + 1 ; one more reader
  if readcount = 1 then wait(wrt) ; if we’re the first, synch with writers
  signal(mutex)
  <perform reading>
  wait(mutex) ; ensure exclusion
  readcount = readcount - 1 ; one fewer reader
  if readcount = 0 then signal(wrt) ; no more readers, allow a writer
  signal(mutex)
Note that:
1. The first reader blocks if there is a writer; any other readers who try to enter will then block on mutex.
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.