TEXAS STATE UNIVERSITY

The rising STAR of Texas

Task Orchestration: Communication and Synchronization

Module developed Spring 2014

This module created with support form NSF under grant # DUE 1141022

Outline

- Basic concepts of orchestration
- Communication
  - Point-to-point vs. collective
  - Synchronous vs. asynchronous
- Synchronization
  - Barrier
  - Lock / semaphore
  - Monitor
  - Classic synchronization problems
- Shared memory vs. message passing

Questions in Orchestration Phase

- Naming data: What names shall processes use to refer to other processes?
- Structuring communication: How and when shall processes communicate with each other? Small or large messages?
- Synchronization: How and when to synchronize?
- How to organize data structures and schedule tasks temporally to exploit data locality?

Goals of Orchestration

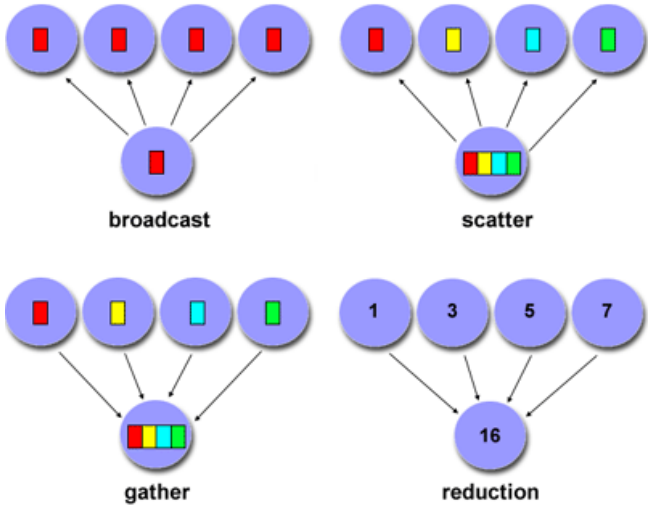
- Reduce cost of communication and synchronization as seen by processors.
- Preserve locality of data reference.
- Schedule tasks early if many other tasks depend on them.
- Reduce the overhead of parallelism management.

Who Needs Communication

- DON'T need communication
  - Types of tasks that can be decomposed and executed in parallel with virtually no need for tasks to share data
  - E.g., an image processing operation where every pixel in a black and white image needs to have its color reversed
- DO need communication
  - Tasks required to share data with each other
  - E.g., a 3-D heat diffusion problem that requires a task to know the temperatures calculated by the tasks that have neighboring data

Types of Communication

- Point-to-point – involves two tasks with one task acting as the sender/producer of data, and the other acting as the receiver/consumer.
- Collective – involves data sharing between more than two tasks which are often specified as being members in a common group, or collective.



Synchronous vs. Asynchronous Communication

- Synchronous communication requires some type of "handshaking" between tasks that are sharing data.
  This can be explicitly structured in code by the programmer, or it may happen at a lower level unknown to the programmer.
- Synchronous communication is often referred to as ***blocking*** communications since other work must wait until the communication has completed.

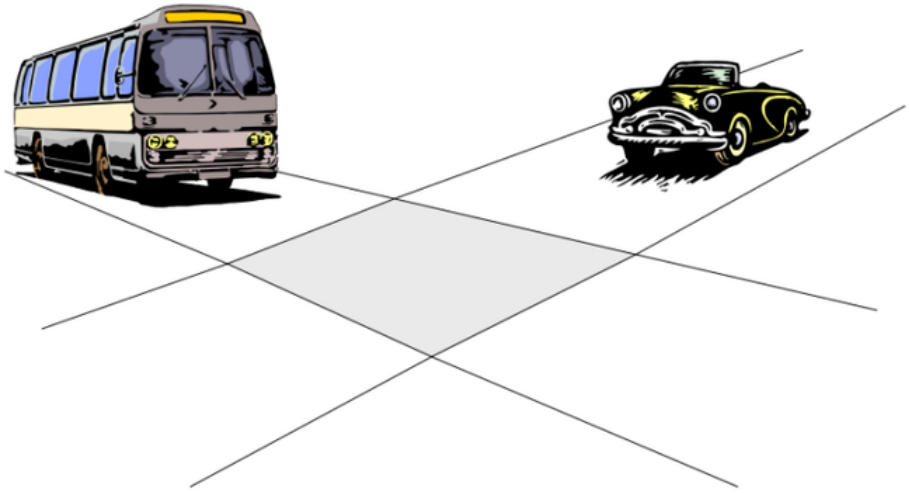
Potential Communication Problems

- Starvation
  - Can occur when multiple processes or threads compete for access to a shared resource and a process is denied access.
- Deadlock
  - Can occur when two processes need multiple resources at the same time in order to continue.

These two pirates are in deadlock because neither is willing to give up what the other pirate needs.

Potential Communication Problems (cont.)

- Data inconsistency
  - Can occur when shared resources are modified at the same time by multiple processes.
  - A section of a program that might cause these problems are called a *critical section*.
  - Failure to coordinate access to a critical section is called a *race condition*.

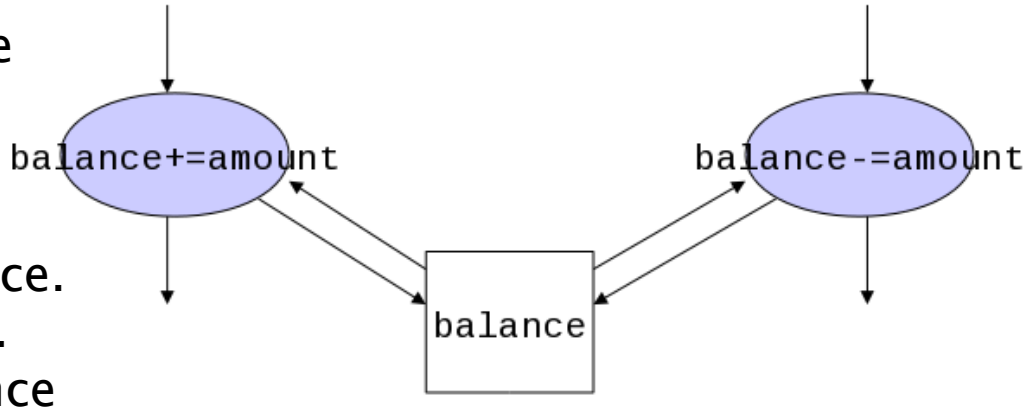


Potential Communication Problems (cont.)

- An example of data inconsistency is the shared buffer problem.

Two processes execute a critical section consisting of three steps:

1. Read account balance.
2. Update the balance.
3. Write the new balance to the account.



One day, Dear Old Dad checks the balance, seeing it is $100, and decides to add $50 to the account. Just then Poor Student withdraws $20 from the account, and the new balance is recorded as $80. After adding the $50, Dear Old Dad records the balance as $150, Rather than $130, as it should be.

Synchronous vs. Asynchronous Communication (cont.)

- Asynchronous communication allows tasks to transfer data independently from one another.
  For example, task 1 can prepare and send a message to task 2, and then immediately begin doing other work. When task 2 actually receives the data doesn't matter.
- Asynchronous communication is often referred to as ***non-blocking*** communication since other work can be done while the communication is taking place.
- Interleaving computation with communication is the single greatest benefit for using asynchronous communication.

Cost of Communication

- Inter-task communication virtually always implies overhead.
- Machine cycles and resources that could be used for computation are instead used to package and transmit data.
- Communication frequently requires some type of synchronization between tasks, resulting in tasks spending time "waiting" instead of doing work.
- Competing communication traffic can saturate the available network bandwidth, further aggravating performance problems.

Latency vs. Bandwidth

- ***Latency*** is the time it takes to send a minimal (0 byte) message from point A to point B. Commonly expressed as microseconds.
- ***Bandwidth*** is the amount of data that can be communicated per unit of time. Commonly expressed as megabytes/sec or gigabytes/sec.
- Sending many small messages can cause latency to dominate communication overheads. Often it is more efficient to package small messages into a larger message, thus increasing the effective communications bandwidth.

Synchronization

- Managing the sequence of work and the tasks performing it is a critical design consideration for most parallel programs.
- Can be a significant factor in program performance (or lack of it)
- Often requires "serialization" of segments of the program.

Types of Synchronization

- **Barrier**
  - Usually implies that all tasks are involved
  - Each task performs its work until it reaches the barrier. It then stops, or "blocks."
  - When the last task reaches the barrier, all tasks are synchronized.
  - What happens from here varies. Often, a serial section of work must be done. In other cases, the tasks are automatically released to continue their work.

Types of Synchronization (cont.)

- **Lock / semaphore**
  - Can involve any number of tasks
  - Typically used to serialize (protect) access to global data or a section of code. Only one task at a time may use (own) the lock / semaphore / flag.
  - The first task to acquire the lock "sets" it. This task can then safely (serially) access the protected data or code.
  - Other tasks can attempt to acquire the lock but must wait until the task that owns the lock releases it.
  - Can be blocking or non-blocking.

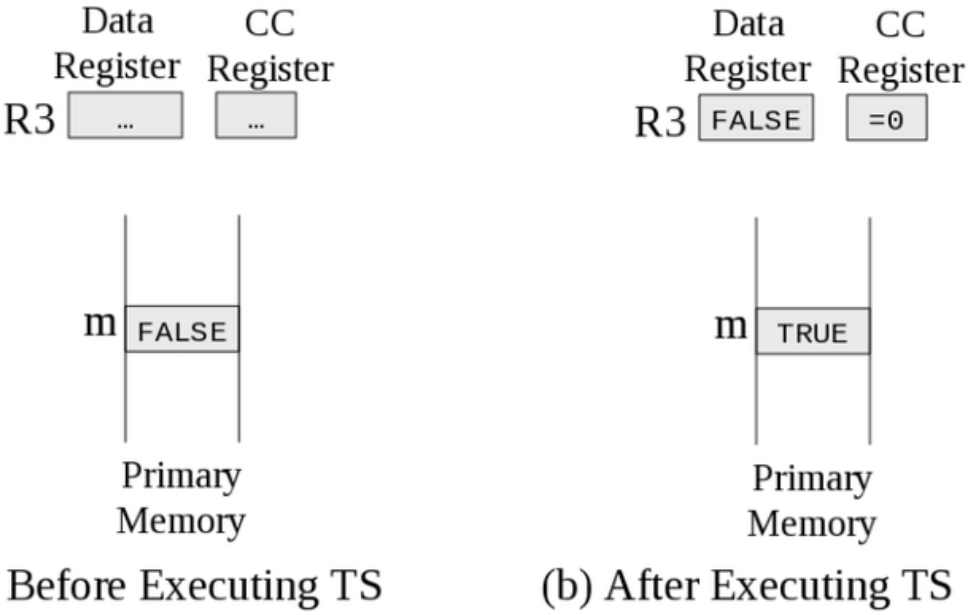
Synchronization for Access to Critical Sections

- Most synchronization problems (e.g., shared buffer problem) require *mutual exclusion*: Only one process can be in the critical section at a time.
- Other problems (such as Readers and Writers) allow more than one read only process in the critical section at the same time.
- Even implementing a simple lock requires mutual exclusion.
- Without mutual exclusion, results of multiple execution are not determinate.

Test and Set Instruction

- Test_and_Set (TS) is a special instruction that does two operations autonomously.
- It is a privileged instruction requiring supervisory mode permission.

TS(m): [Reg_i=memory[m];
memory[m]=TRUE;]



Dijkstra's Semaphore

- A semaphore, s, is a nonnegative integer variable that can only be changed or tested by these two indivisible functions:
  - V(s): [s=s+1;]
  - P(s): [while (s==0) {WAIT()}; s=s-1;]
- V and P are the first letters of Dutch words.
  - P (probern – to test) is used for acquiring the lock.
  - V (verhogen – to increment) is for releasing the lock.
- The initial value of s determines how many simultaneous process may hold the semaphore lock.

Implementing Semaphore with Test_and_Set

```
struct semaphore {
    int value = <initial value>;
    boolean mutex = FALSE;
    boolean hold = TRUE;
    queue m, h
};

shared struct semaphore s;
```

```
/* P() */
acquire_semaphore(struct semaphore s) {
    /* wait for intertal mutex */
    while(TS(s.mutex)) WAIT(s.m);

    s.value--;
    if(s.value < 0) (
        s.mutex = FALSE;
        SIGNAL(s.m);
        /* wait - too many out */
        while(TS(s.hold)) WAIT(s.h);
    } else {
        s.mutex = FALSE;
        SIGNAL(s.m);
    }
}
```

```
/* V() */
release_semaphore(struct semaphore s) {
    /* wait for intertal mutex */
    while(TS(s.mutex)) WAIT(s.m);

    s.value++;
    if(s.value >= 0) (
        s.hold = FALSE;
        SIGNAL(s.h);
    }
    s.mutex = FALSE;
    SIGNAL(s.m);
}
```

Monitor

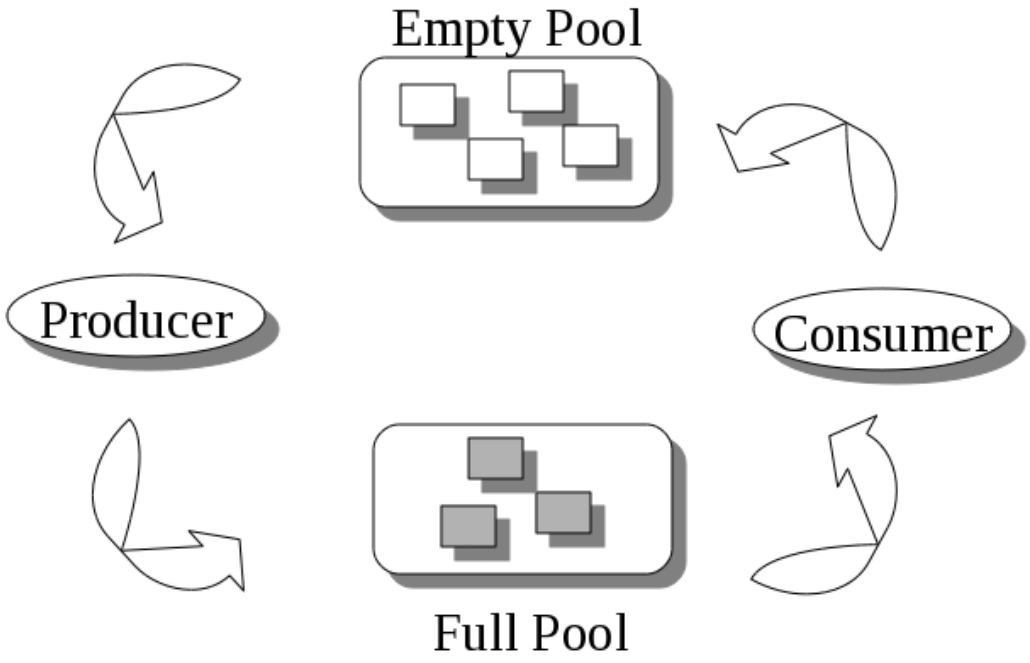
- Semaphore provides a foundation for implementing solutions to synchronization problems, but using it correctly is a challenge.
- An alternative mechanism called a *monitor* can provide a simpler solution for some problems. Monitors are also called conditional waits or conditional monitors.
- A conditional monitor contains an internal mutual exclusion lock and methods for a process to wait() until another process issues a notify() method to awake the process.

Monitor (cont.)

```
/* Code to enter a critical section */
monitor.acquire();
while( not_okay_to_enter() )
    monitor.wait();
keep_others_out();
monitor.release();
/*-----------------------------*/
/*    Critical Section         */
/*-----------------------------*/
/* Exit of critical section code */
monitor.acquire();
okay_others_to_enter();
monitor.notify();
monitor.release();
```

Classic Synchronization Problems

- Shared Buffer
  - A single lock or semaphore can be used to enforce mutual exclusion.
- Bounded Buffer (Producers and Consumers)



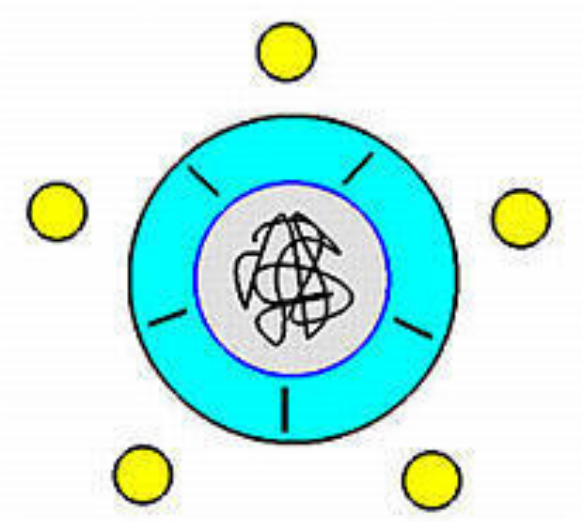
Classic Synchronization Problems (cont.)

- Readers and Writers

Writers have mutual exclusion, but multiple readers are allowed at the same time.

Classic Synchronization Problems (cont.)

- Dinning Philosophers



A philosopher needs both the fork on his left and the fork on his right to eat. The forks are shared with the neighbors on either side.

Shared Memory vs. Message Passing

- Shared memory
  - Efficient, familiar
  - Not always available
  - Potentially insecure

```
global int x

process foo          process bar
begin                begin
  :                    :
 x := ...             y := x
  :                    :
end foo              end bar
```

- Message passing
  - Extensible to communication in distributed systems
  - Canonical syntax:

```
send(process : process_id, message : string)
receive(process : process_id, var message : string)
```

Shared Memory Programming

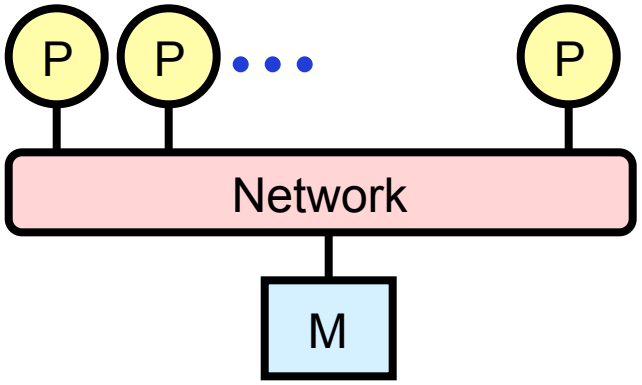
- Programs/threads communicate/cooperate via loads/stores to memory locations they share.
- Communication is therefore at memory access speed (very fast), and is implicit.
- Cooperating pieces must all execute on the same system (computer).
- OS services and/or libraries used for creating tasks (processes/threads) and coordination (semaphores/barriers/locks).

Shared Memory Code

```
fork N processes
each process has a number, p, and computes
   istart[p], iend[p], jstart[p], jend[p]
for(s=0;s<STEPS;s++) {
  k = s&1 ; m = k^1 ;
  forall(i=istart[p];i<=iend[p];i++) {
    forall(j=jstart[p];j<=jend[p];j++) {
       a[k][i][j] = c1*a[m][i][j] + c2*a[m][i-1][j] +
            c3*a[m][i+1][j] + c4*a[m][i][j-1] +
            c5*a[m][i][j+1] ; // implicit comm
    }
  }
   barrier() ;
}
```

Symmetric Multiprocessors

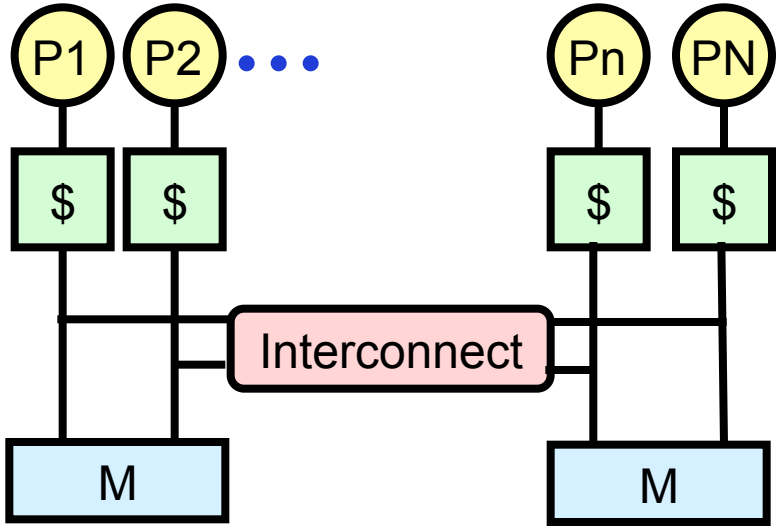
- Several processors share one address space
  - conceptually a shared memory
- Communication is *implicit*
  - read and write accesses to shared memory locations
- Synchronization
  - via shared memory locations
    - spin waiting for non-zero
  - Atomic instructions (Test&set, compare&swap, load linked/store conditional)
  - barriers



Conceptual Model

Non-Uniform Memory Access - NUMA

- CPU/Memory busses cannot support more than ~4-8 CPUs before bus bandwidth is exceeded (the SMP “sweet spot”).
- To provide shared-memory MPs beyond these limits requires some memory to be “closer to” some processors than to others.



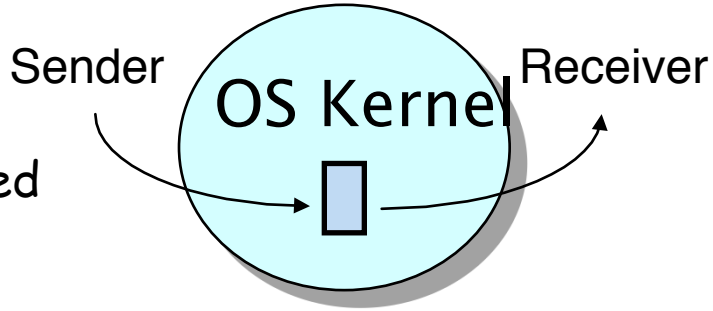
Message Passing Programming

- “Shared” data is communicated using “send”/”receive” services (across an external network).
- Unlike Shared Model, shared data must be formatted into message chunks for distribution (shared model works no matter how the data is intermixed).
- Coordination is via sending/receiving messages.
- Program components can be run on the same or different systems, so can use 1,000s of processors.
- “Standard” libraries exist to encapsulate messages:
  - Parasoft's Express (commercial)
  - PVM (standing for Parallel Virtual Machine, non-commercial)
  - MPI (Message Passing Interface, also non-commercial).

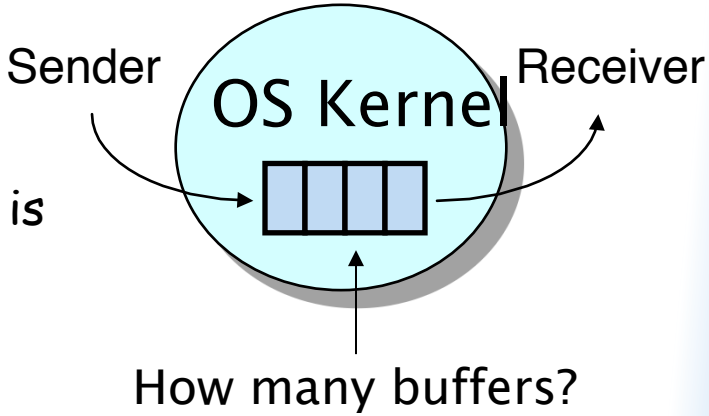
Message Passing Issues

When does a *send* /*receive* operation terminate?

Blocking (aka Synchronous):
- Sender waits until its message is received
- Receiver waits if no message is available



Non-blocking (aka Asynchronous):
- Send operation "immediately" returns
- Receive operation returns if no message is available (polling)



Partially blocking/non-blocking:
- send()/receive() with *timeout*

Clustered Computers Designed for Message Passing

- A collection of computers (nodes) connected by a network
  - computers augmented with fast network interface
    - send, receive, barrier
    - user-level, memory mapped
  - otherwise indistinguishable from conventional PC or workstation
- One approach is to network workstations with a very fast network
  - Often called ‘cluster computers’
  - Berkeley NOW
  - IBM SP2 (remember Deep Blue?)