MPI: Message Passing application level IPC and synchronization library

- Originated at Argonne National Lab.
- Used in “Beowulf” GNU/Linux PC/network computing clusters.
- Has functions to transmit and receive arrays of data (of specified types and sizes) between specified processes, often on different hosts.
- Has barrier synchronization functions.
- Has bindings for high level languages: C, Fortran, etc.
Message Passing entails a Buffer

- Buffer: In mechanical engineering, a buffer provides force or power coupling between things that move at different speeds. A clutch, using friction, is a kind of buffer.

- Visualize a buffer of messages. Adding to a full buffer or reading from an empty buffer blocks.

- All sleep/wait IPC and synchronization operations (semaphores, monitor/wait/signal like facilities, message passing) have complex implementations with internal queues of processes waiting to be awakened.
Rendezvous

- Tanenbaum mentions special case of message passing in which the buffer has size zero. The sending process blocks until the receiving process tries to read the message, and vice versa.
Internals of Message Passing

- A review of the (Java) monitor-based solution to the producer-consumer problem shows the buffer explicitly.

- In Tanenbaum's message passing solution, a specific number of messages (N=100) from the consumer to the producer, called “empties” is used to limit the buffer size to N.
public class ProducerConsumer {
    static final int N = 100; // constant giving the buffer size
    static producer p = new producer(); // instantiate a new producer thread
    static consumer c = new consumer(); // instantiate a new consumer thread
    static our_monitor mon = new our_monitor(); // instantiate a new monitor

    public static void main(String args[]) {
        p.start(); // start the producer thread
        c.start(); // start the consumer thread
    }

    static class producer extends Thread {
        public void run() { // run method contains the thread code
            int item;
            while (true) { // producer loop
                item = produce_item();
                mon.insert(item);
            }
        }

        private int produce_item() { ... } // actually produce
    }

    . . .

Figure 2-35. A solution to the producer-consumer problem in Java.
static class consumer extends Thread {
    public void run() { run method contains the thread code
        int item;
        while (true) { // consumer loop
            item = mon.remove();
            consume_item(item);
        }
    }
}

private void consume_item(int item) { ... } // actually consume

static class our_monitor { // this is a monitor
    private int buffer[] = new int[N];
    private int count = 0, lo = 0, hi = 0; // counters and indices
    public synchronized void insert(int val) {
        if (count == N) go_to_sleep(); // if the buffer is full, go to sleep
        buffer[hi] = val; // insert an item into the buffer
        hi = (hi + 1) % N; // slot to place next item in
        count = count + 1; // one more item in the buffer now
        if (count == 1) notify(); // if consumer was sleeping, wake it up
    }

Figure 2-35. A solution to the producer-consumer problem in Java. (Java docs. specify that the condition waited for must be tested again: The if(count==N) above should be while(count==N) etc.)
public synchronized int remove() {
    int val;
    if (count == 0) go_to_sleep(); // if the buffer is empty, go to sleep
    val = buffer [lo]; // fetch an item from the buffer
    lo = (lo + 1) % N; // slot to fetch next item from
    count = count - 1; // one few items in the buffer
    if (count == N - 1) notify(); // if producer was sleeping, wake it up
    return val;
}

private void go_to_sleep() { try{wait();} catch(InterruptedException exc) {}};

Figure 2-35. A solution to the producer-consumer problem in Java. (Please replace if() go_to_sleep();s by whiles)
```
#define N 100
/* number of slots in the buffer */

void producer(void)
{
    int item;
    message m;
    /* message buffer */

    while (TRUE) {
        item = produce_item();
        receive(consumer, &m);
        build_message(&m, item);
        send(consumer, &m);
        /* generate something to put in buffer */
        /* wait for an empty to arrive */
        /* construct a message to send */
        /* send item to consumer */
    }
}

...
```

Figure 2-36. The producer-consumer problem with N messages.
void consumer(void)
{
    int item, i;
    message m;

    for (i = 0; i < N; i++) send(producer, &m); /* send N empties */
    while (TRUE) {
        receive(producer, &m); /* get message containing item */
        item = extract_item(&m); /* extract item from message */
        send(producer, &m); /* send back empty reply */
        consume_item(item); /* do something with the item */
    }
}

Figure 2-36. The producer-consumer problem with N messages.
Figure 2-37. Use of a barrier. (a) Processes approaching a barrier. (b) All processes but one blocked at the barrier. (c) When the last process arrives at the barrier, all of them are let through.

A Barrier is one object referred to by many processes or threads. They all sleep until every one arrives, then they all become ready.
Figure 2-38. Bursts of CPU usage alternate with periods of waiting for I/O. (a) A CPU-bound process. (b) An I/O-bound process. Tanenbaum notes an asymmetry: How much CPU time is used between I/O operations is what matters in the differentiation. How much I/O time DOESN'T because all I/O operations take about the same CPU time to initiate. A process waiting for I/O doesn't require CPU time. (I misspoke in the lecture..sorry!)
Categories of Scheduling Algorithms

The main distinction is preemptive versus non-preemptive scheduling. Preemptive means the OS kernel reschedules (calls the scheduler again) periodically during timer interrupts. Both preemptive and non-preemptive systems respond to I/O interrupts and they reschedule when a process makes a blocking system call.

- Batch
- Interactive
- Real time
Scheduling Algorithm Goals

All systems
Fairness - giving each process a fair share of the CPU
Policy enforcement - seeing that stated policy is carried out
Balance - keeping all parts of the system busy

Batch systems
Throughput - maximize jobs per hour
Turnaround time - minimize time between submission and termination
CPU utilization - keep the CPU busy all the time

Interactive systems
Response time - respond to requests quickly
Proportionality - meet users’ expectations

Real-time systems
Meeting deadlines - avoid losing data
Predictability - avoid quality degradation in multimedia systems

Figure 2-39. Some goals of the scheduling algorithm under different circumstances. Many are quantitative.
Quantities of Merit: DEFINED

- **Throughput** is number of jobs completed per unit time. (say jobs/hour or jobs/second)

- **Turnaround time** is the average time from the moment a job is submitted to the moment it completes. (say seconds)

- **Response time** is the time between issuing a command and getting the result. (seconds)

- **CPU utilization** is the proportion of time the CPU is doing useful work. (unit-less fraction or %)