# Before we start

- **Reminder:– Fill out partner evaluations for all Projects**
- **Remember to sign up for demonstrations of Project 1**
  - Essential for getting a grade for this project!
- **In IA:–**
  - Tools > Demonstration Scheduling
  - Select an entry marked “OPEN” at a mutually convenient time

# Before we start (continued)

- …
- **Project 2 checkpoint is SUNDAY, January 28, at 11:59 PM**
- **Don’t delay**
  - No room in calendar for extensions

# Questions?

# Application Design in a Concurrent World

Professor Hugh C. Lauer
CS-3013 — Operating Systems

# Reading Assignment

- **Lampson, B.W., and Redell, D. D., "Experience with Processes and Monitors in Mesa," *Communications of ACM,* vol. 23, pp. 105-117, Feb. 1980. (.pdf)**

- **OSTEP, Part II (§25-34)**
  - A broad view of concurrency, locks, monitors, and condition variable
  - Some of their uses
  - Different order from traditional textbooks

# Challenge

- **In a modern world with many processors, how should multi-threaded applications be designed?**

- ***Not in OS textbooks***
  - Focus on process and synchronization mechanisms
  - Not on so much how they are used
  - Hardly at all about how they *should* be used!
  - See OSTEP, §25-34

- **Reference**
  - Kleiman, Shah, and Smaalders, *Programming with Threads*, SunSoft Press (Prentice Hall), 1996
  - Out of print!

# Three traditional models (plus one new one)

- Data parallelism
- Task parallelism
- Pipelining

- Google-style massive parallelism

# Other Applications

- **Some concurrent applications don't fit any of these models**
- **Some may fit more than one model at the same time.**
  - E.g., Microsoft Word

# Three traditional models (plus one new one)

- **Data parallelism**
- Task parallelism
- Pipelining

- Google-style massive parallelism

# Data Parallel Applications

- **Single problem with large data**
  - Matrices, arrays, etc.
- **Divide up the data into subsets**
  - E.g., Divide a big matrix into quadrants or sub-matrices
  - Generally in an orderly way
- **Assign separate thread (or process) to each subset**
  - Threads execute same program
  - E.g., matrix operation on separate quadrant
  - Separate coordination & synchronization required


IBM

# Data Parallelism (continued)

- **Imagine multiplying two $n \times n$ matrices**
  - Result is $n^2$ elements
  - Each element is $n$-member dot product — i.e., $n$ multiply-and-add operations
  - Total $n^3$ operations (multiplications and additions)
  - If $n = 10^5$, matrix multiply takes $10^{15}$ operations (i.e., ½ week on a 3 GHz Pentium!)

# Matrix Multiply (continued)

$$\left[\begin{array}{cc|cc} A_{1,1} & A_{1,2} & A_{1,3} & A_{1,4} \\ A_{2,1} & A_{2,2} & A_{2,3} & A_{2,4} \\ \hline A_{3,1} & A_{3.2} & A_{3,3} & A_{3,4} \\ A_{4,1} & A_{4,2} & A_{4,3} & A_{4,4} \end{array}\right] \times \left[\begin{array}{cc|cc} B_{1,1} & B_{1,2} & B_{1,3} & B_{1,4} \\ B_{2,1} & B_{2,2} & B_{2,3} & B_{2,4} \\ \hline B_{3,1} & B_{3.2} & B_{3,3} & B_{3,4} \\ B_{4,1} & B_{4,2} & B_{4,3} & B_{4,4} \end{array}\right]$$

# Matrix Multiply (continued)

$$\left[\begin{array}{cc|cc} A_{1,1} & A_{1,2} & A_{1,3} & A_{1,4} \\ A_{2,1} & A_{2,2} & A_{2,3} & A_{2,4} \\ \hline A_{3,1} & A_{3.2} & A_{3,3} & A_{3,4} \\ A_{4,1} & A_{4,2} & A_{4,3} & A_{4,4} \end{array}\right] \times \left[\begin{array}{cc|cc} B_{1,1} & B_{1,2} & B_{1,3} & B_{1,4} \\ B_{2,1} & B_{2,2} & B_{2,3} & B_{2,4} \\ \hline B_{3,1} & B_{3.2} & B_{3,3} & B_{3,4} \\ B_{4,1} & B_{4,2} & B_{4,3} & B_{4,4} \end{array}\right]$$

- **Multiply 4 sub-matrices in parallel (4 threads)**
  - UL×UL, UR×LL, LL×UR, LR×LR
- **Multiply 4 other sub-matrices together (4 threads)**
  - UL×UR, UR×LR, LL×UL, LR×LL
- **Add results together**

# Observation

- **Multiplication of sub-matrices can be done in parallel in separate threads**
    - No data conflict
- **Results must be added together after all four multiplications are finished.**
    - Somewhat parallelizable
    - Only $O(n^2/2)$ additions

# Matrix Multiply (continued)

$$\left[\begin{array}{cc:cc} A_{1,1} & A_{1,2} & A_{1,3} & A_{1,4} \\ A_{2,1} & A_{2,2} & A_{2,3} & A_{2,4} \\ \hdashline A_{3,1} & A_{3.2} & A_{3,3} & A_{3,4} \\ A_{4,1} & A_{4,2} & A_{4,3} & A_{4,4} \end{array}\right] \times \left[\begin{array}{cc:cc} B_{1,1} & B_{1,2} & B_{1,3} & B_{1,4} \\ B_{2,1} & B_{2,2} & B_{2,3} & B_{2,4} \\ \hdashline B_{3,1} & B_{3.2} & B_{3,3} & B_{3,4} \\ B_{4,1} & B_{4,2} & B_{4,3} & B_{4,4} \end{array}\right]$$

$O((n/2)^3)$

$O((n/2)^3)$

- **Multiply 4 sub-matrices in parallel (4 threads)**
  - UL×UL, UR×LL, LL×UR, LR×LR
- **Multiply 4 other sub-matrices together (4 threads)**
  - UL×UR, UR×LR, LL×UL, LR×LL
- **Add results together**

$O(n^2/2)$

# Amdahl's Law

- Let *P* be ratio of time in parallelizable portion of algorithm to total time of algorithm

- I.e.,

$$P = \frac{Exec\ time\ of\ parallelizable\ part}{Total\ execution\ time}$$

# Amdahl's Law (continued)

Very Important!

- If $T_S$ is execution time in serial environment *then*

$$T_P = T_S \times \left( (1 - P) + \frac{P}{N} \right)$$

is execution time on *N* processors

- I.e., *speedup* factor is

$$S = \frac{T_S}{T_P} = \frac{1}{(1 - P) + \frac{P}{N}}$$

Diminishing returns in N

# More on Data Parallelism

- **Primary focus – big number crunching**
  - Weather forecasting, weapons simulations, gene modeling, drug discovery, finite element analysis, etc.
- **Typical synchronization primitive – *barrier synchronization***
  - I.e., wait until all threads reach a common point
- **Many tools and techniques**
  - E.g., *OpenMP* – a set of tools for parallelizing loops based on compiler directives
  - See www.openmp.org

# Definition — *Barrier Synchronization*

- A synchronization mechanism in which each thread waits at a *barrier* until all *n* have arrived.

- `pthread_barrier_init`
- `pthread_barrier_wait`
- …

- A fundamental tool in data parallelism

# Questions?

# Three traditional models (plus one new one)

- Data parallelism
- **Task parallelism**
- Pipelining

- Google-style massive parallelism

# Task Parallel Applications

- **Many independent tasks**
  - Usually very small
  - E.g., airline reservation request
- **Shared database or resource**
  - E.g., the common airline reservation database
- **Each task assigned to separate thread**
  - No direct interaction among tasks (or very little!)
  - Tasks share access to common data objects

# Task Parallelism (continued)

- **Each task is small, independent**
  - Too small for parallelization within itself
  - Great opportunity to parallelize separate tasks
- **Challenge – access to common resources**
  - Access to independent objects in parallel
  - Serialize accesses to shared objects
- **A “mega” critical section problem**

# Semaphores and Task Parallelism

- ***Semaphores* can theoretically solve critical section issues of many parallel tasks with a lot of parallel data …**

- ***BUT:***
  - No direct relationship to the data being controlled
  - *Very* difficult to use correctly; easily misused
    - Global variables
    - Proper usage requires *superhuman* attention to detail

- **Need another approach**
    - Preferably one with programming language support

# Solution – *Monitors*

- **Programming language construct that supports controlled access to shared data**
  - Compiler adds synchronization automatically
  - Enforced at runtime
- **Encapsulates**
  - Shared data structures
  - Procedures/functions that operate on the data
  - Synchronization between threads calling those procedures
- **Only one thread *active* inside a monitor at any instant**
  - All functions are part of *one* critical section

**Hoare, C.A.R., "Monitors: An Operating System Structuring Concept," *Communications of ACM,* vol. 17, pp. 549-557, Oct. 1974 (.pdf, correction)**

# Monitors

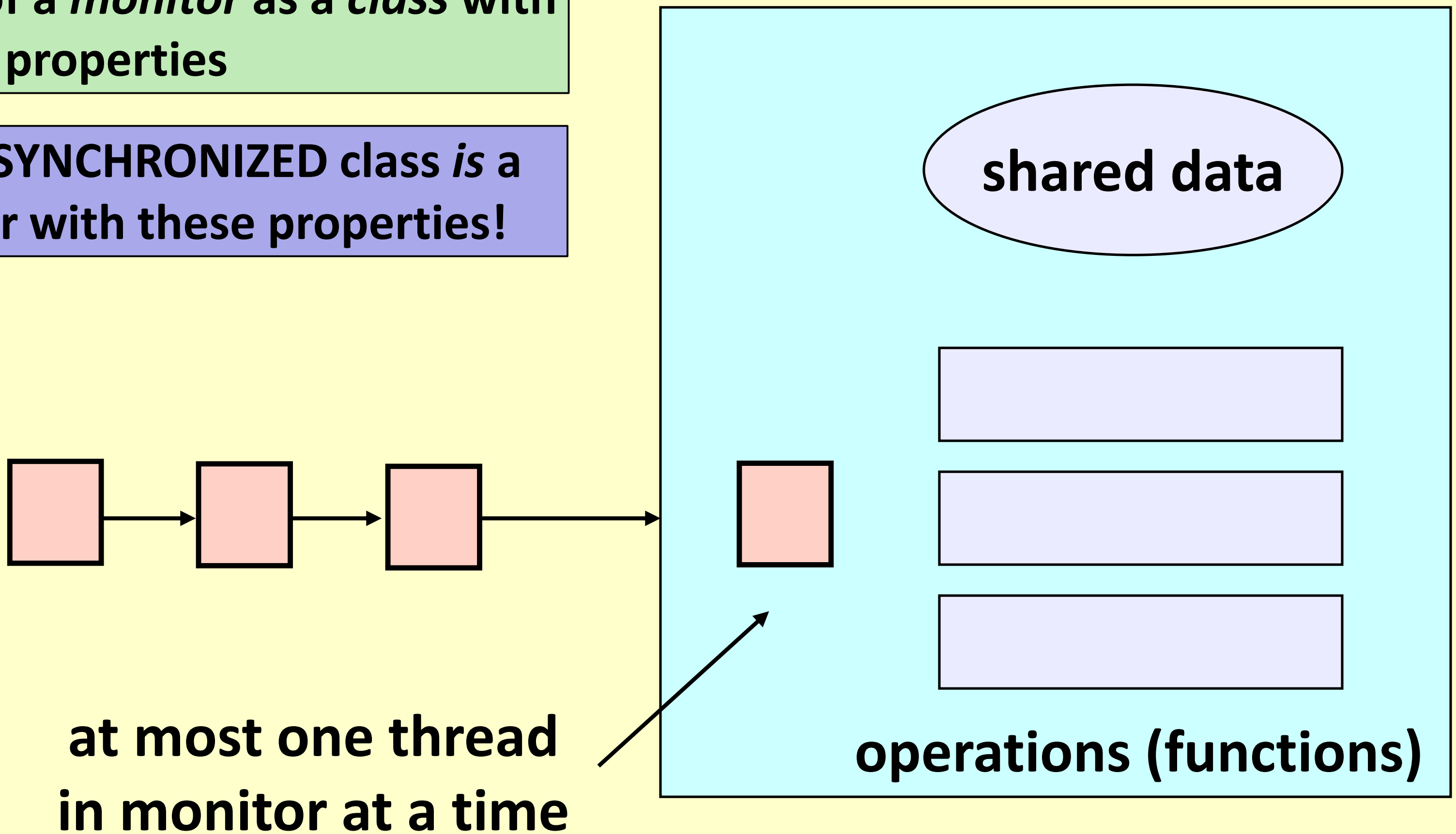
- **High-level synchronization allowing safe sharing of an abstract data type among concurrent threads.**

```
monitor monitor-name
{
     monitor data declarations (shared among
        functions)
     function body F1 (…) {
        . . .
     }
     function body F2 (…) {
        . . .
     }
     function body Fn (…) {
         . . .
     }
     {
        initialization & finalization code
     }
}
```

# Monitors

Think of a *monitor* as a *class* with special properties

A Java SYNCHRONIZED class *is* a monitor with these properties!



# Synchronization with Monitors

Essentially a semaphore with values limited to zero and one

- **Mutual exclusion**
  - Each monitor has a *built-in* mutual exclusion lock
  - Only one thread can be *executing* inside at any time
  - If another thread tries to enter a monitor procedure, it blocks until the first relinquishes the monitor
- **Once inside a monitor, thread may discover it is not able to continue**
  - Due to some condition or external event that must happen
- ***condition variables* provided within monitor**
  - Threads can **`wait`** for something to happen — i.e., an *Event!*
  - Threads can **`signal`** others that something has happened
  - Condition variable can only be accessed from inside monitor
  - **`wait`**'ing thread relinquishes monitor lock temporaril

Does not resemble a semaphore at all!

Cannot even be implemented with semaphores!

# Waiting within a Monitor

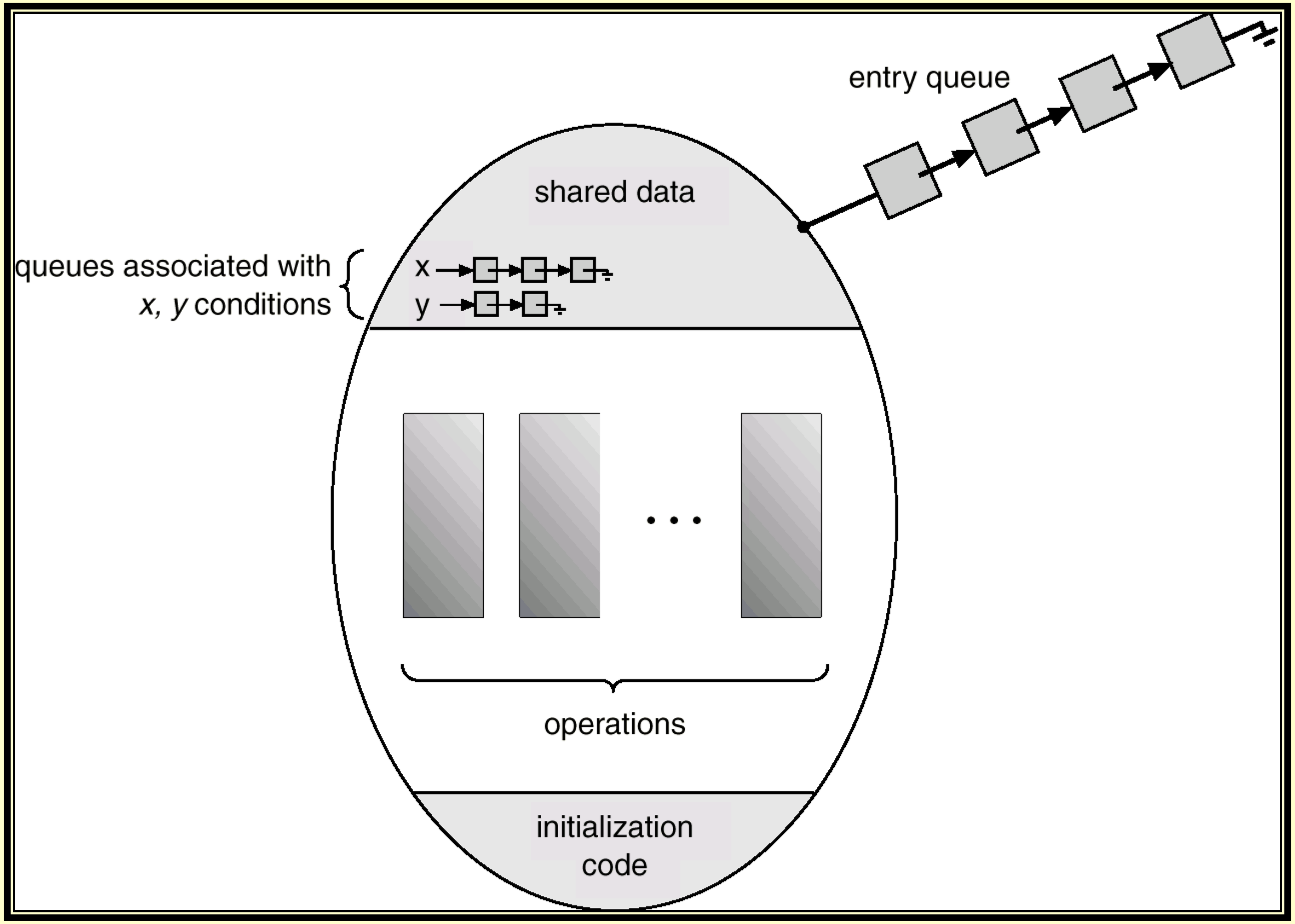
- **To allow a thread to wait *within* the monitor, a *condition* variable must be declared, as**

```
condition x;
```

- ***Condition variable* – a queue of waiting threads inside the monitor**

- **Can only be used with the operations `wait` and `signal`**
  - Operation **`wait(x)`** means that thread invoking this operation is suspended until another thread invokes **`signal(x)`**
  - The **`signal`** operation resumes exactly one suspended thread. If no thread is suspended, then the **`signal`** operation has no effect.

# Monitors – Condition Variables



# `wait` and `signal` (continued)

- **The thread invoking `wait` *automatically relinquishes* monitor lock to allow other threads in.**

- **When a thread invokes `signal`, any resumed thread must *reacquire monitor lock* before proceeding**
  - Program counter is still inside the monitor
  - Resumed thread cannot proceed until it actually *acquires* the monitor lock

# Signal Semantics

- **`signal(c)` means**
  - Waiting thread is made ready, but signaler continues
    - Waiting thread competes for monitor lock whenever signaler leaves monitor or executes **`wait`** itself
    - condition not necessarily true when waiting thread runs again
  - being signaled is only a *hint* of something changed
    - must recheck conditional case

**`pthread_cond_signal` and Java `Notify()` method use these semantics!**

**Called *Mesa Semantics* in OSTEP**

# Monitor Example*

```
monitor FIFOMessageQueue {
▪ struct qItem {
    struct qItem *next,*prev;
    msg_t msg;
  };
▪ /* internal data of queue*/
▪ struct qItem *head, *tail;
  conditi
▪ /* fun
▪ void addMsg(msg_t newMsg);
  msg_t removeMsg(void);
▪ /* constructor/destructor */
▪
};
```

Only one thread allowed inside monitor at a time!

Let any waiting threads know that message queue is non-empty!

```
/* function implementations */
FIFOMessageQueue(void) {
▪ /* constructor*/
  head = tail = NULL;
};
void addMsg(msg_t newMsg) {
▪ qItem *new = malloc(qItem);
  new→prev = tail;
  new→next = NULL;
▪ if (tail==NULL) head = new;
  else tail→next = new;
▪ tail = new;
▪ signal nonEmpty;
};
```

* Adapted from Kleiman, Shah, and Smaalders

# Monitor Example (continued)

```
/* function implementations
        continued*/

msg_t removeMsg(void) {
■ while (head == NULL)
    wait(nonEmpty);
■ struct qItem *old = head;
■ if (old→next == NULL)
    tail = NULL; /*last
  element*/
  else
    old→next→prev = NULL;
■ head = old→next;
■ msg_t msg = old→msg;
■ free(old);
■ return(msg);
};
```

Even after returning from wait, we are not 100% sure queue non-empty — Why?

```
/* function implementations
    concluded*/

~FIFOMessageQueue(void) {
    head = top→next;
    free(top);
};

/* what is missing here? */

/* answer:- need to unblock
waiting threads in destructor!
*/
```

# Invariants

- **Monitors lend themselves naturally to *programming invariants***
  - I.e., logical statements or assertions about what is true when no thread holds the monitor lock
  - Similar to *loop invariant* in sequential programming
  - All monitor operations must preserve invariants
  - All functions must restore invariants before **`wait`**ing

- **Easier to explain & document**
  - Especially during code reviews with co-workers

# Invariants of Example

- Element are stored in order of arrival!

- `head` points to first element
  - or `NULL` if no elements
- `tail` points to last element
  - or `NULL` if no elements)
- Each element except head has a non-null `prev`
  - Points to element insert just prior to this one
- Each element except tail has a non-null `next`
  - Points to element insert just after to this one
- `head` has a null `prev`; `tail` has a null `next`

# Personal Experience

- During design of *Pilot* operating system …
- Prior to introduction of monitors, it took an advanced degree in CS and a *lot* of work to design and debug critical sections
- Afterward, a new team member with BS and ordinary programming skills could design and debug monitor as first project
  - And get it right the first time!

# Modern world

- **More advanced techniques available and appropriate**
- **Many monitor implementations involve system calls**
  - Expensive in time and performance!
- **Wait-free implementations for simple shared data structures**
  - Java
- **Comprehensive text on theory and practice:–**

***The Art of Multiprocessor Programming*, by Maurice Herlihy & Nir Shavit, Morgan Kaufman, 2012**

# Monitors – Summary

- ***Much* easier to use than semaphores**
  - Especially to get it right
  - Helps to have language support
- **Available in Java – `SYNCHRONIZED CLASS`**
- **Can be simulated in C or C++ using**
  - **`pthreads`**, **`conditions`**, **`mutexes`**, etc.

- **Highly adaptable to *object-oriented programming***
  - Each separate object can be its own monitor!
- **Monitors may have their own threads inside!**

# Monitors in *C*

- **Must be programmed in "long-hand"**
    - No language support!
- **Monitor data implemented as a `struct`**
    - Include one `pthread_mutex_t` object
        - I.e., the "monitor lock"
    - One or more `pthread_cond_t` objects
        - I.e., the condition variables (representing events or conditions to wait for)
- **Many implementations available in *C++***

# Monitors in C or C++ (*continued*)

- **Initialization & destruction of mutexes:–**
  - `pthread_mutex_init(lock)  /*dynamic*/`
  - `pthread_mutex_t lock =    /*static*/`
    `PTHREAD_MUTEX_INITIALIZER;`
  - `pthread_mutex_destroy(lock)`
- **Conditions variables:–**
  - `pthread_cond_init(event)  /*dynamic*/`
  - `pthread_cont_t event =    /*static*/`
    `PTHREAD_COND_INITIALIZER;`
  - `pthread_cond_destroy(event)`

# Monitors in C or C++ (*continued*)

- For protecting monitor data, every function must be surrounded by:–
  - `pthread_mutex_lock(lock)` or `pthread_mutex_trylock(lock)`
  - `pthread_mutex_unlock(lock)`

Only the thread holding the lock can unlock it

- For conditions and events:–
  - `pthread_cond_wait(event, lock)`
  - `pthread_cond_signal(event)`
  - `pthread_cond_broadcast(event)`

Same here!

# Monitors – References

- **OSTEP, Part II (§25-34)**

- **See also**
  - Lampson, B.W., and Redell, D. D., "Experience with Processes and Monitors in Mesa," *Communications of ACM,* vol. 23, pp. 105-117, Feb. 1980. (.pdf)
  - Redell, D. D. *et al.* "Pilot: An Operating System for a Personal Computer," *Communications of ACM,* vol. 23, pp. 81-91, Feb. 1980. (.pdf)

- ***We will use or simulate monitors in Project 3***

# Message-oriented Design

## (Another variant of Task Parallelism)

- **Shared resources managed by separate processes**
  - Typically in separate address spaces
- **Independent task threads send messages requesting service**
  - Task state encoded in message and responses
- **Manager does work and sends reply messages to tasks**
- **Synchronization & critical sections**
  - Inherent in message queues and process main loop
  - Explicit queues for internal waiting

# Message-Oriented Design (continued)

- **Message-oriented and monitor-based designs are equivalent!**
  - Including structure of source code
  - Performance
  - Parallelizability
  - Shades of *Remote Procedure Call* (RPC)!
  - However, not so amenable to *object-oriented* design
- **See**
  - Lauer, H.C. and Needham, R.M., "On the Duality of Operating System Structures," *Operating Systems Review,* vol 13, #2, April 1979, pp. 3-19. (.pdf)

# Questions?

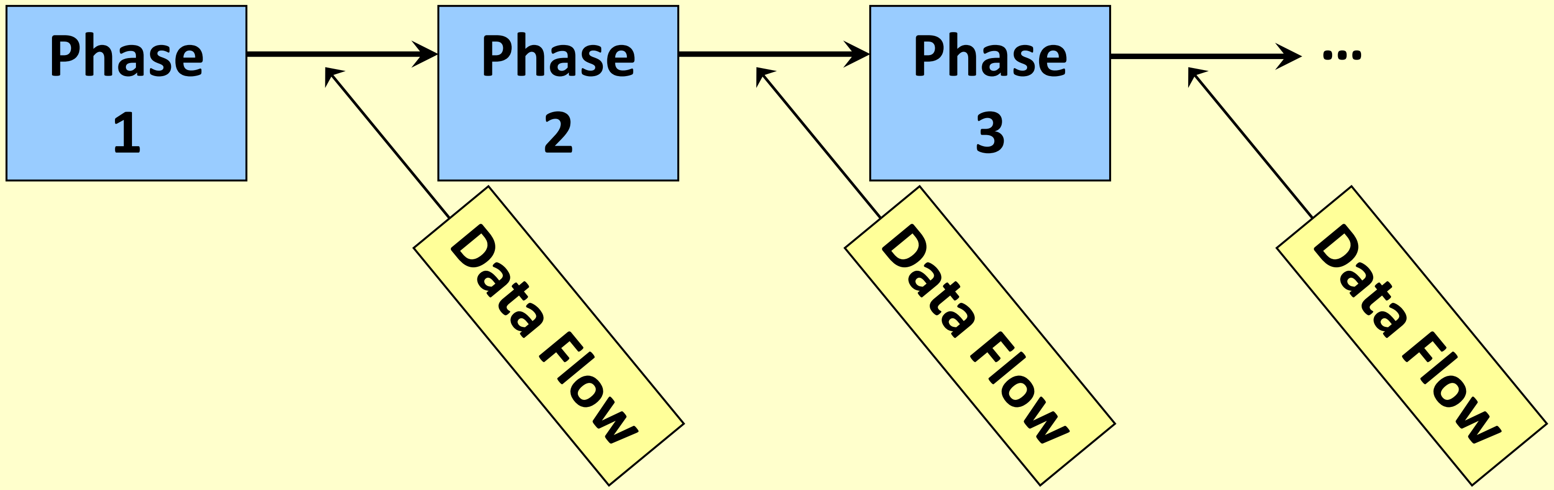
# Three traditional models (plus one new one)

- Data parallelism
- Task parallelism
- Pipelining
- Google-style massive parallelism

# Pipelined Applications

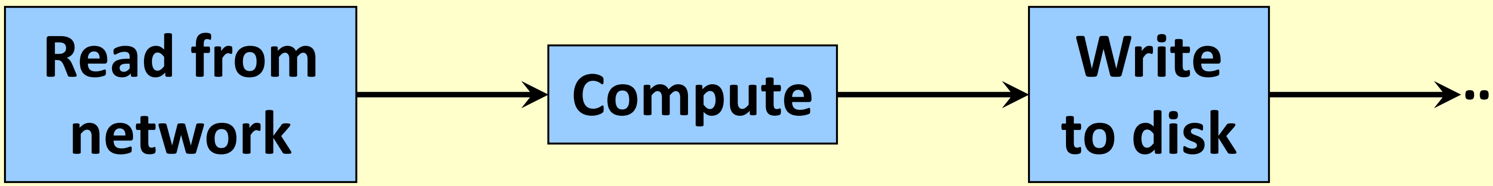
- **Application can be partitioned into phases**
    - Output of each phase is input to next
- **Separate threads or processes assigned to separate phases**
    - Data flows through phases from start to finish, pipeline style
- **Buffering and synchronization needed to**
    - Keep phases from getting ahead of adjacent phases
    - Keep buffers from overflowing or underflowing

# Pipelined Parallelism



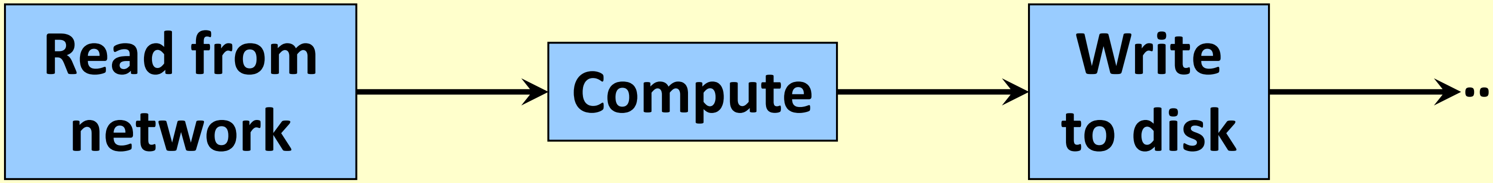

- **Assume phases do not share resources**
  - Except data flow between them
- **Phases can execute in separate threads in parallel**
  - I.e., *Phase 1* works on item *i*, which *Phase 2* works on item *i-1*, while *Phase 3* works on item *i-2*, etc.

# Example




- Reading from network involves long waits for each item
- Computing is non-trivial
- Writing to disk involves waiting for disk arm, rotational delay, etc.

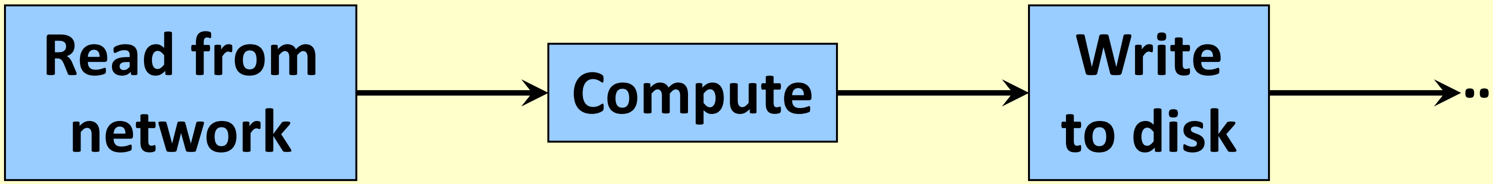
# Example Time Line



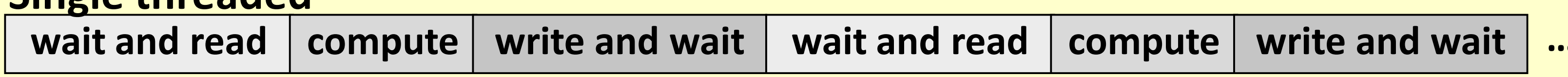

**Single threaded**

| | | | | | | |
|---|---|---|---|---|---|---|
| wait and read | compute | write and wait | wait and read | compute | write and wait | … |

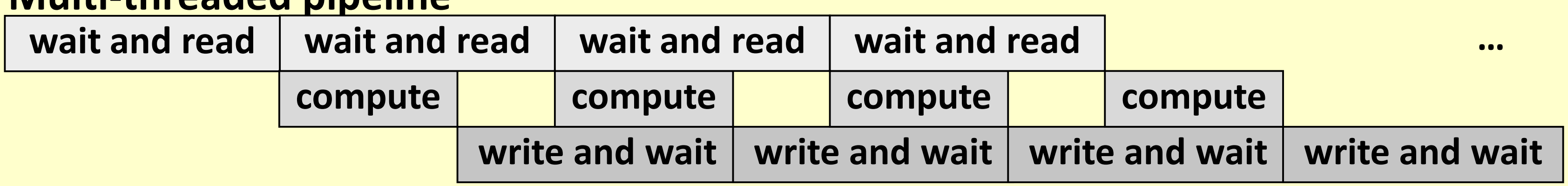
# Example Time Line




**Single threaded**




**Multi-threaded pipeline**



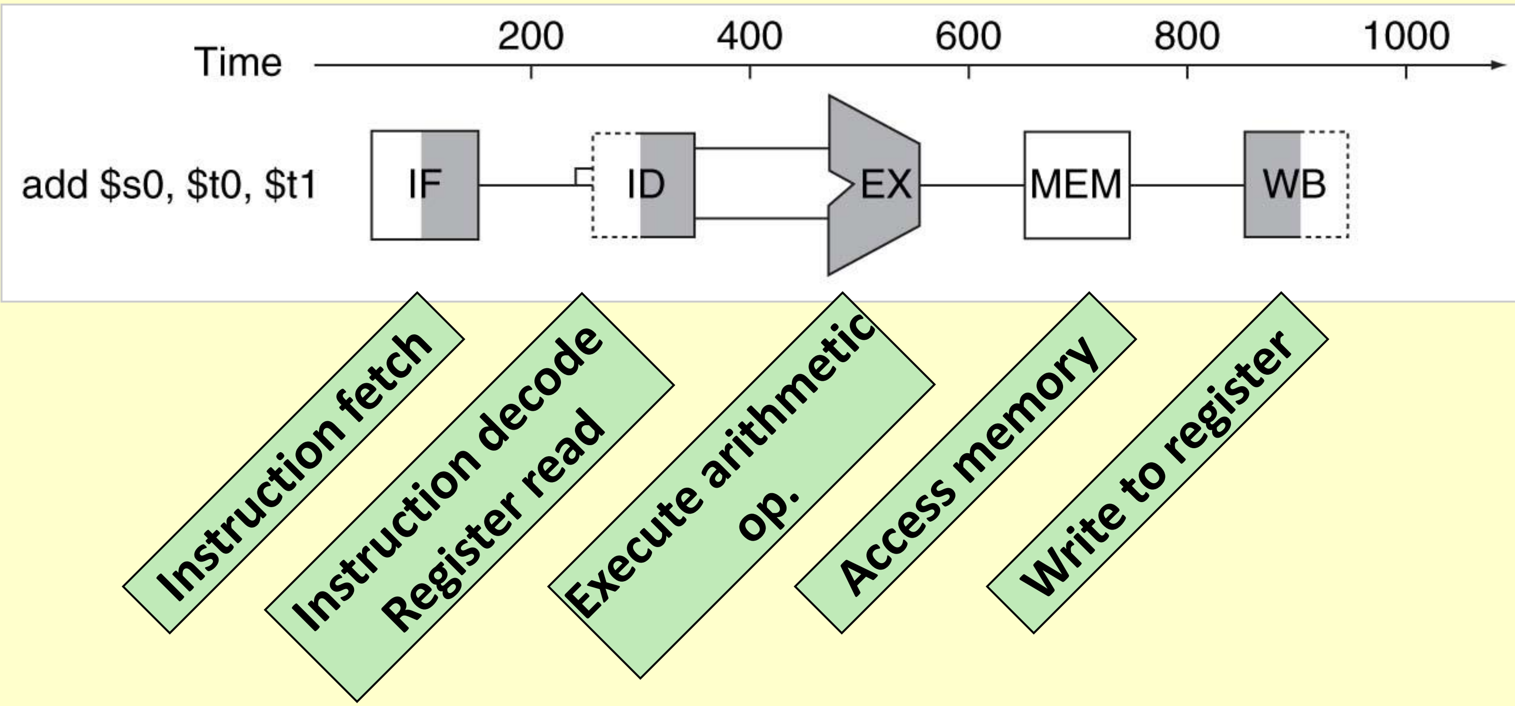
# Example

- **Unix/Linux/Windows *pipes***
  - `read xx | compute yy | write zz`
- **Execute in separate processes**
  - Data flow passed between them via OS *pipe* abstraction
  - I.e., **`stdout`** of one process becomes **`stdin`** of next process

# Another Example

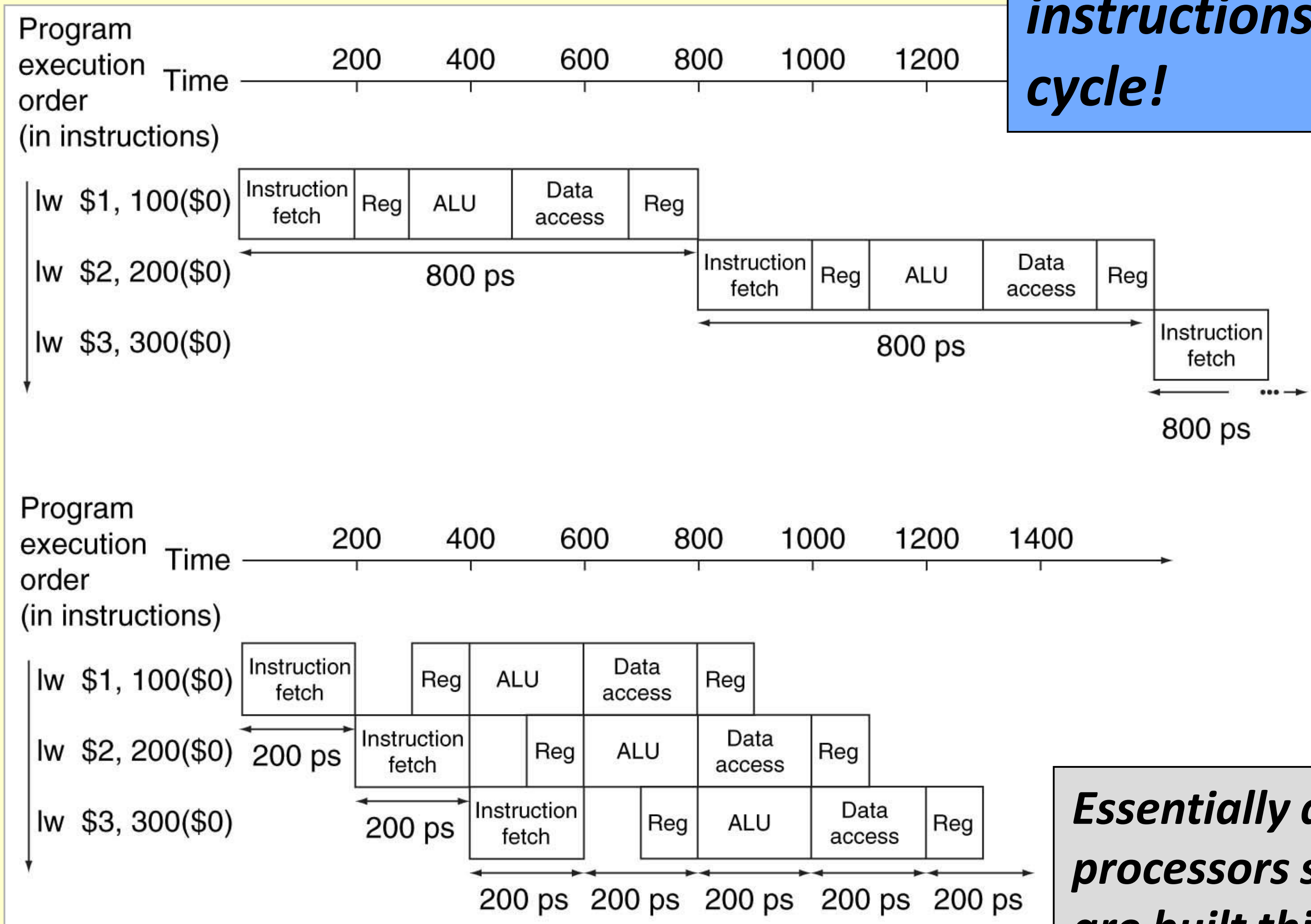
- **PowerPoint presentations**
  - One thread manages display of current slide
  - A separate thread “reads ahead” and formats the next slide
  - Instantaneous progression from one slide to the next

# Third Example – Pipelined Processor



# Pipelined Processor (continued)

***Modern processors have 12-30 pipeline stages and issue 4+ instructions in each cycle!***




***Essentially all micro-processors since 1985 are built this way***

# Producer-Consumer

- **Fundamental synchronization mechanism for decoupling the flow between parallel phases**

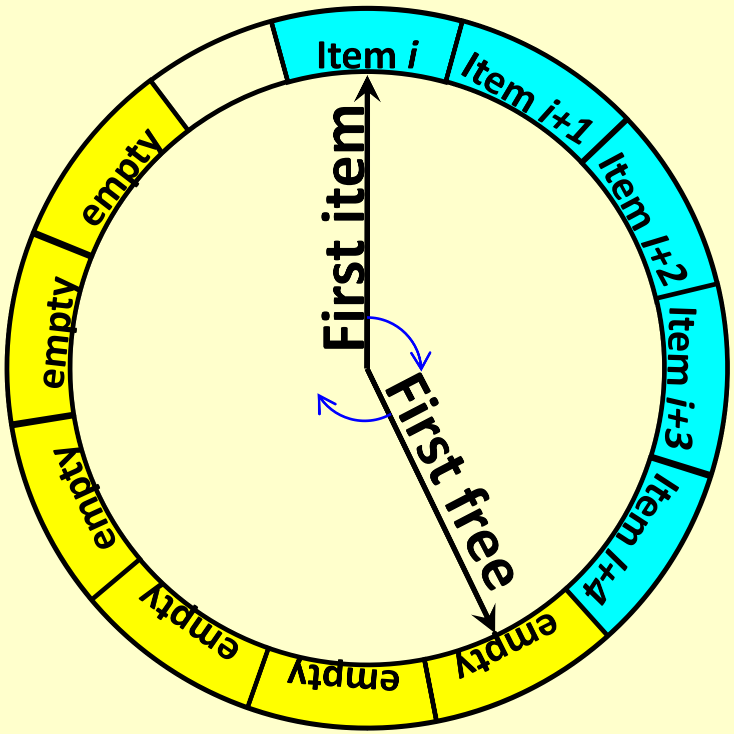
- **One of the few areas where *semaphores* are natural tool**

# Definition — *Producer-Consumer*

- **A method by which one process or thread communicates an *unbounded* stream of data through a *finite buffer* to another.**

- ***Buffer:*– a temporary storage area for data**
  - *Esp.* an area by which two processes (or computational activities) at different speeds can be *decoupled* from each other

# Example – Ring Buffer




**Consumer empties items, starting with first full item**

**Producer fills items, starting with first free slot**

# Implementation with Semaphores

```
struct Item { …
};
Item buffer[n];
semaphore empty = n, full = 0;
```

**Producer:–**

```
int j = 0;
    while (true) {
        wait_s(empty);
        produce(buffer[j]);
        post_s(full);
        j = (j+1) mod n;
    }
```

**Consumer:–**

```
int k = 0;
    while (true) {
        wait_s(full);
        consume(buffer[k]);
        post_s(empty);
         k = (k+1) mod n;
    }
```

# Implementation with Semaphores

```
struct Item { …
};
Item buffer[n];
semaphore empty = n, full = 0;
```

**Producer:–**

```
int j = 0;
   while (true) {
      wait_s(empty);
      produce(buffer[j]);
      post_s(full);
      j = (j+1) mod n;
   }
```

**Consumer:–**

```
int k = 0;
   while (true) {
      wait_s(full);
      consume(buffer[k]);
      post_s(empty);
       k = (k+1) mod n;
   }
```

**Note: critical section may be needed to protect buffer!**

# Real-world example — I/O overlapped with computing

- **Producer:– the input-reading process**
  - Reads data as fast as device allows
  - Waits for physical device to transmit records
  - Unblocks and stores data into ring buffer, one record per slot

- **Consumer**
  - Computes on each record in turn
  - Is freed from the details of waiting and unblocking physical input

# Double Buffering

- A *producer-consumer* application with *n=2*
- Widely used for many years
- Most modern operating systems provide this in I/O and file *read* and *write* functions

# Summary: Producer-Consumer

- Occurs frequently throughout computing
- Needed for *decoupling* the timing of two activities
- Especially useful in Pipelined parallelism
- Uses whatever synchronization mechanism is available

# Questions?

# A final note (for all three models)

- **Multi-threaded applications require *thread safe* libraries**
  - I.e., so that system library functions may be called concurrently from multiple threads at the same time
  - E.g., *malloc*(), *free*() for allocating from heap and returning storage to heap
- **Most modern Linux & Windows libraries are thread safe**

# Thread-safe Libraries

- **May implemented as monitors**
  - Using `pthread_mutex`, `pthread_cond`
- **Only one thread in “module” at a time**
  - File writes or reads
  - …
- **Some may be implemented by wait-free operations**
  - `malloc()`, `free()`

# Questions?

# Three traditional models (plus one new one)

- Data parallelism
- Task parallelism
- Pipelining

- Google-style massive parallelism

# Google & Massive Parallelism

- **Exciting topic of research in recent years**
- **1000s, 10000s, or more threads/processes**
- **Primary function – *Map/Reduce***
  - Dispatches 1000s of tasks that search on multiple machines in parallel
  - Collects results together
- **Topic for another time/course**

# Reading Assignment

- **OSTEP, Part II (§25-34)**
  - More than you can read in one sitting …
  - … or even in one week!

- **Lampson, B.W., and Redell, D. D., "Experience with Processes and Monitors in Mesa," *Communications of ACM,* vol. 23, pp. 105-117, Feb. 1980.**

http://www.cs.wpi.edu/~cs3013/c14/Papers/LampsonRedell_Monitors.pdf

# Questions?