

Parallel Hardware

Flynn's Classical Taxonomy (1966)

SISD: Single Instruction Stream Single Data Stream

Uniprocessor

SIMD: Single Instruction Stream Multiple Data Stream

Processors with local memory containing different data execute the same instruction in a synchronized fashion

Array processors, GPU

MISD: Multiple Instruction Stream Single Data Stream

Fault tolerance, but mostly nonsense

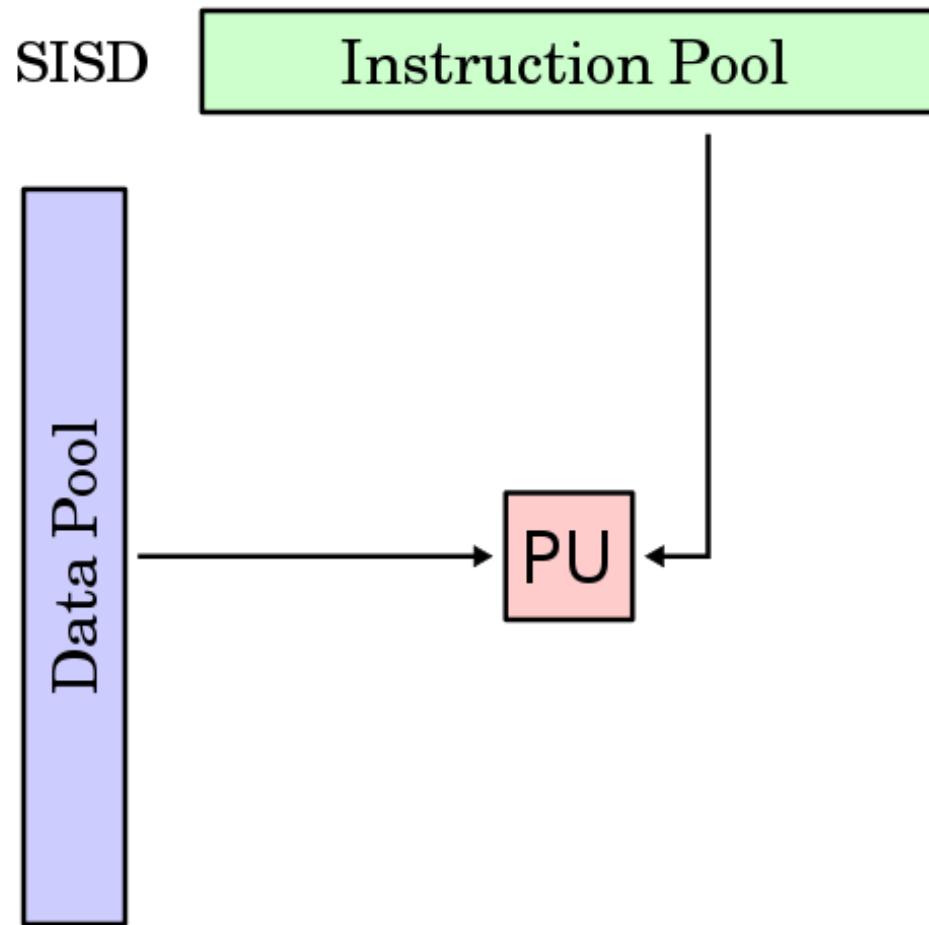
MIMD: Multiple Instruction Stream Multiple Data Stream

The MIMD architecture performs multiple actions simultaneously on numerous data pieces.

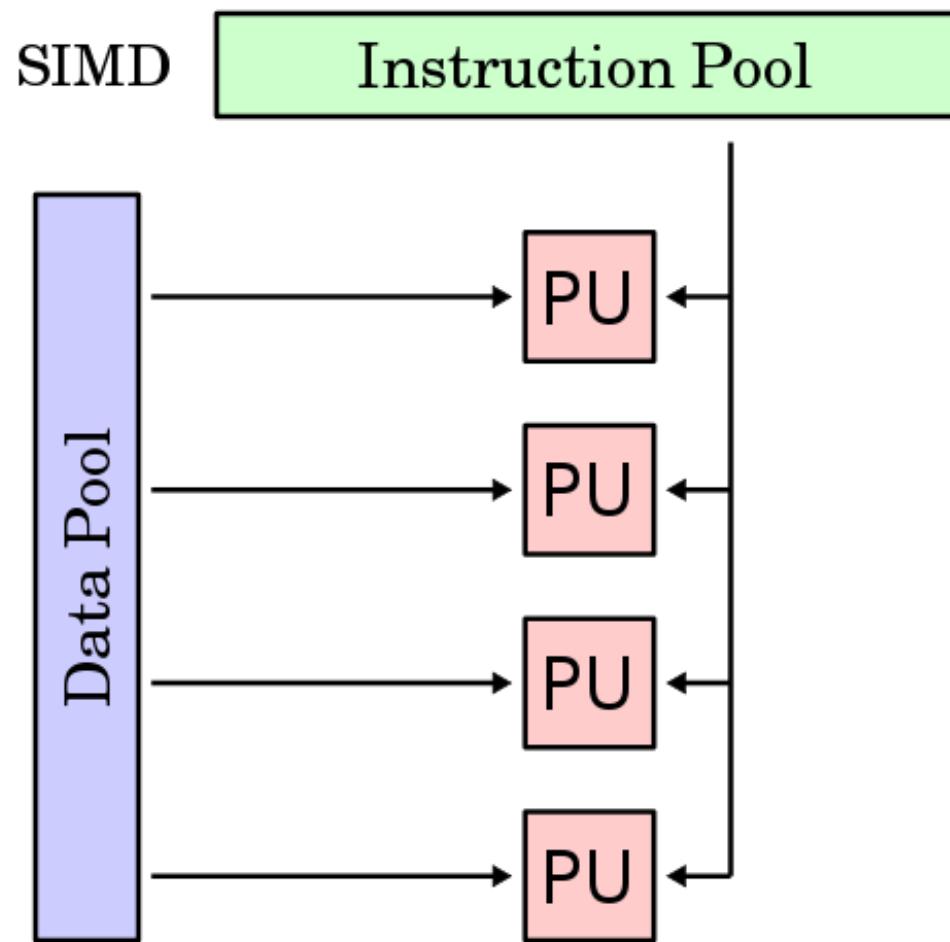
Multi-core processor (most modern PCs)

Computer Clusters, Network of workstations

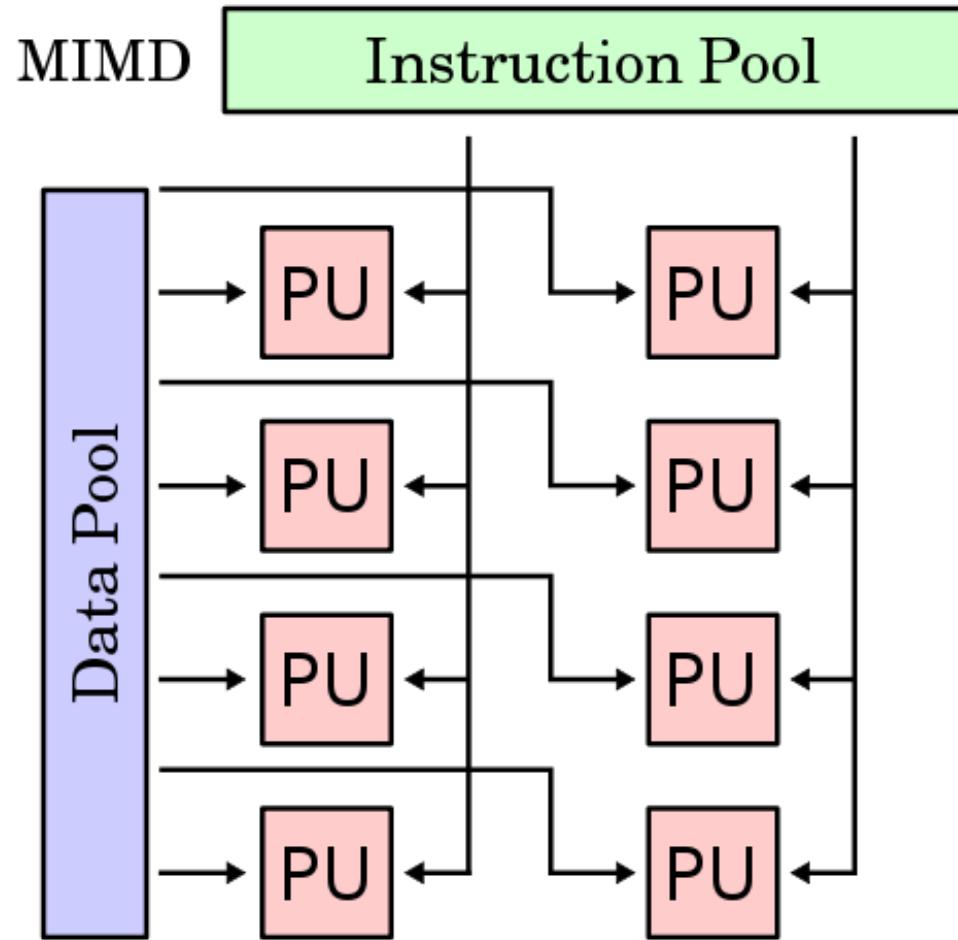
SISD



SIMD

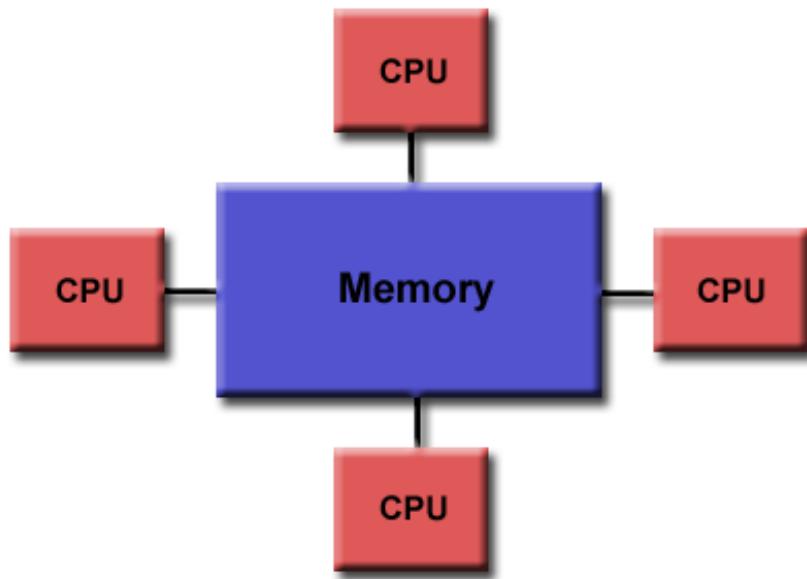


MIMD



MIMD includes both shared-memory and distributed-memory architectures

Shared Memory



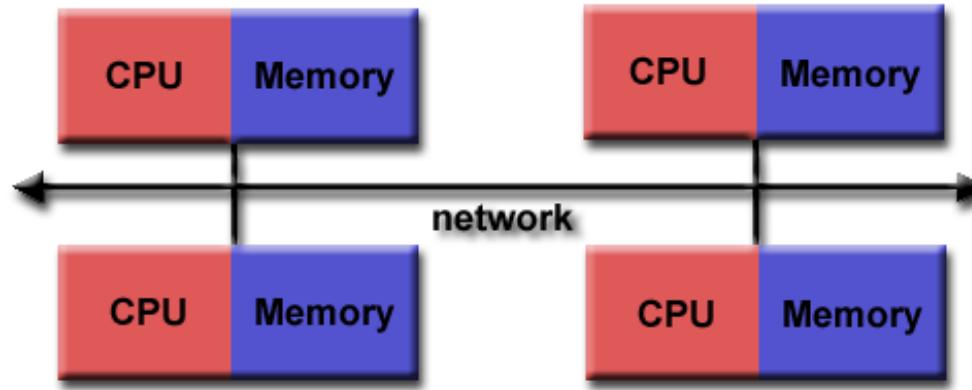
Pros: Provides a user-friendly programming perspective to memory

 Data sharing between tasks is both fast and uniform

Cons: No scalability between memory and CPUs

Keys: correctly accessing shared resources, synchronization

Distributed Memory



Pros: Memory is scalable with the number of processors.

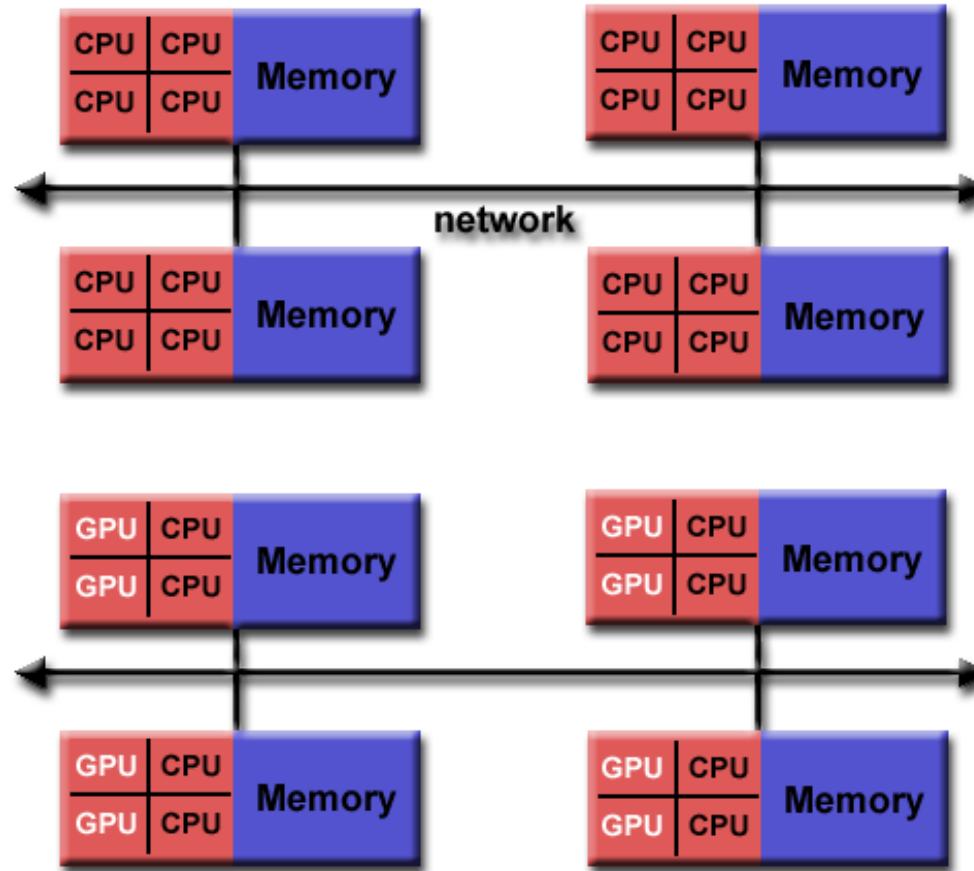
Cost effective, can use commodity, off-the-shelf processors.

Cons: Data communication between processors.

Difficult to map existing data structures to the memory organization.

Non-uniform memory access times: remote vs. local

Hybrid Distributed-Shared Memory



Parallel Software

Parallel Software

Multiple program multiple data (MPMD):

Mainly functional decomposition, e.g., client-server, etc.

Single program multiple data (SPMD):

In a single program, tasks are split up and run simultaneously on multiple processors with different input

Use Branch to implement data-parallelism

```
if (I'm thread/process 0)
    operate on the first half of the array;
else /* I'm thread/process 1 */
    operate on the second half of the array;
```

Parallel Software

Threads

To be used on a shared memory system

Light-weight subroutines in a program

Threads communicate with each other through global memory (by writing and reading address locations) in a program

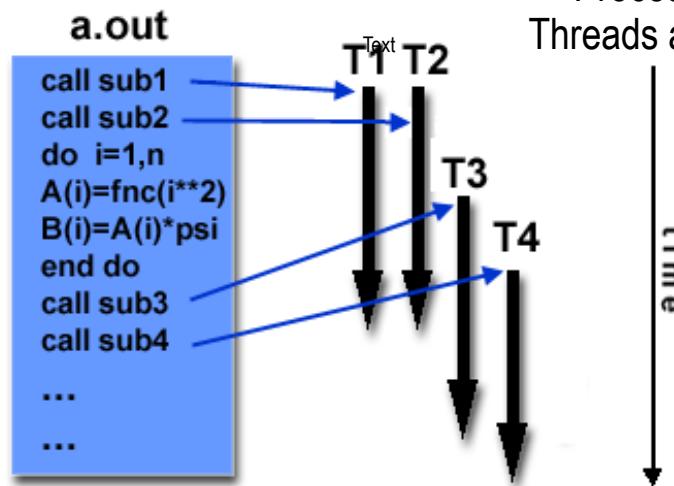
Example: POSIX Threads (Pthreads), OpenMP

Process = Program, although a Program can have multiple processes

Process = multiple Threads

Process has its own (separate) memory space

Threads are sharing the memory space in a process



Parallel Software

Message Passing Model

Processes

A set of tasks that use their own local memory during computation.

Tasks exchange data by sending and receiving messages.

Data transfer usually requires cooperative operations to be performed by each process.

on one process

For example, a *send* operation must have a matching *receive* operation.

on another process

Example: MPI

Can run on either a shared memory system or distributed memory system

Message Passing Program Example

```
// Pseudo Code  
char message[100]; ...  
  
my rank = Get rank();    Get the rank of the process  
  
if (my rank == 1) {  
  
    sprintf(message, "Greetings from process 1");  
  
    Send(message, MSG CHAR, 100, 0);    send to process 0  
  
} else if (my rank == 0) {  
  
    Receive(message, MSG CHAR, 100, 1);    receive from process 1  
  
    printf("Process 0 > Received: %s\n", message);  
}
```

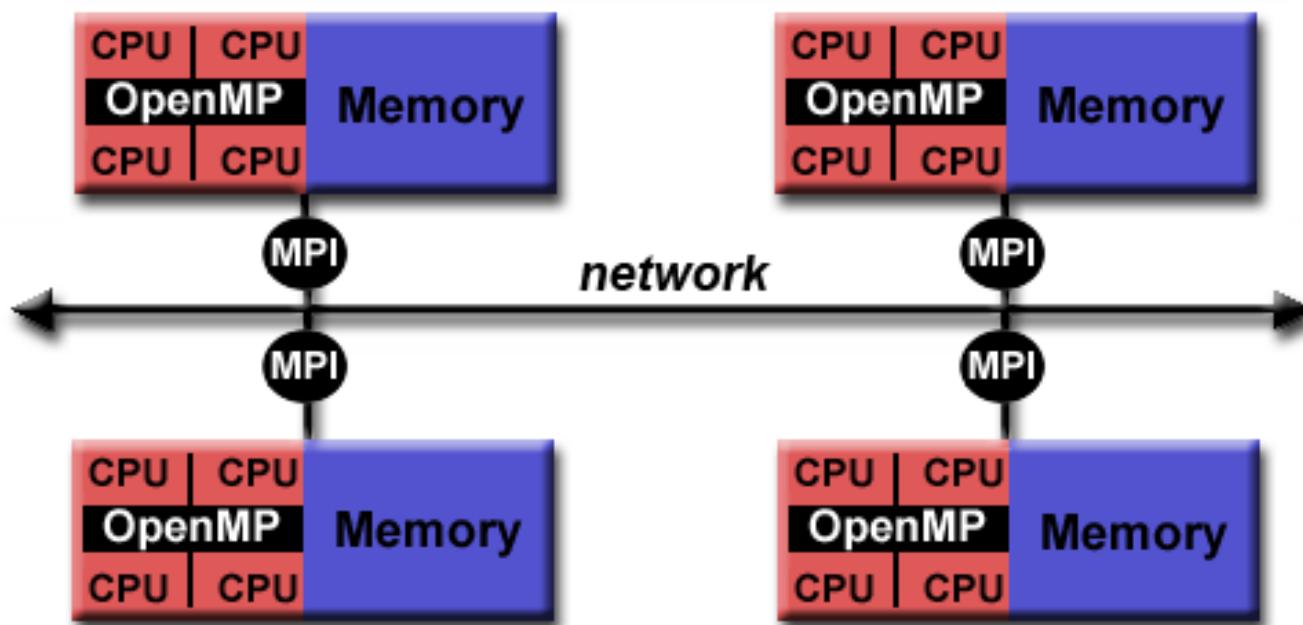
Parallel Software

Hybrid Programming Model

MPI + OpenMP

MPI + GPU

Used to achieve highest performance



Performance and Speedup

Speedup and Efficiency

Let T_1 be the time to solve the problem sequentially.

Let T_p be the time to solve the problem in parallel using p processors

n is the data size

Then, speedup $S(p)$ for problem size n is defined as:

$$S(p) = T_1/T_p \quad (= p)$$

The overhead is defined by

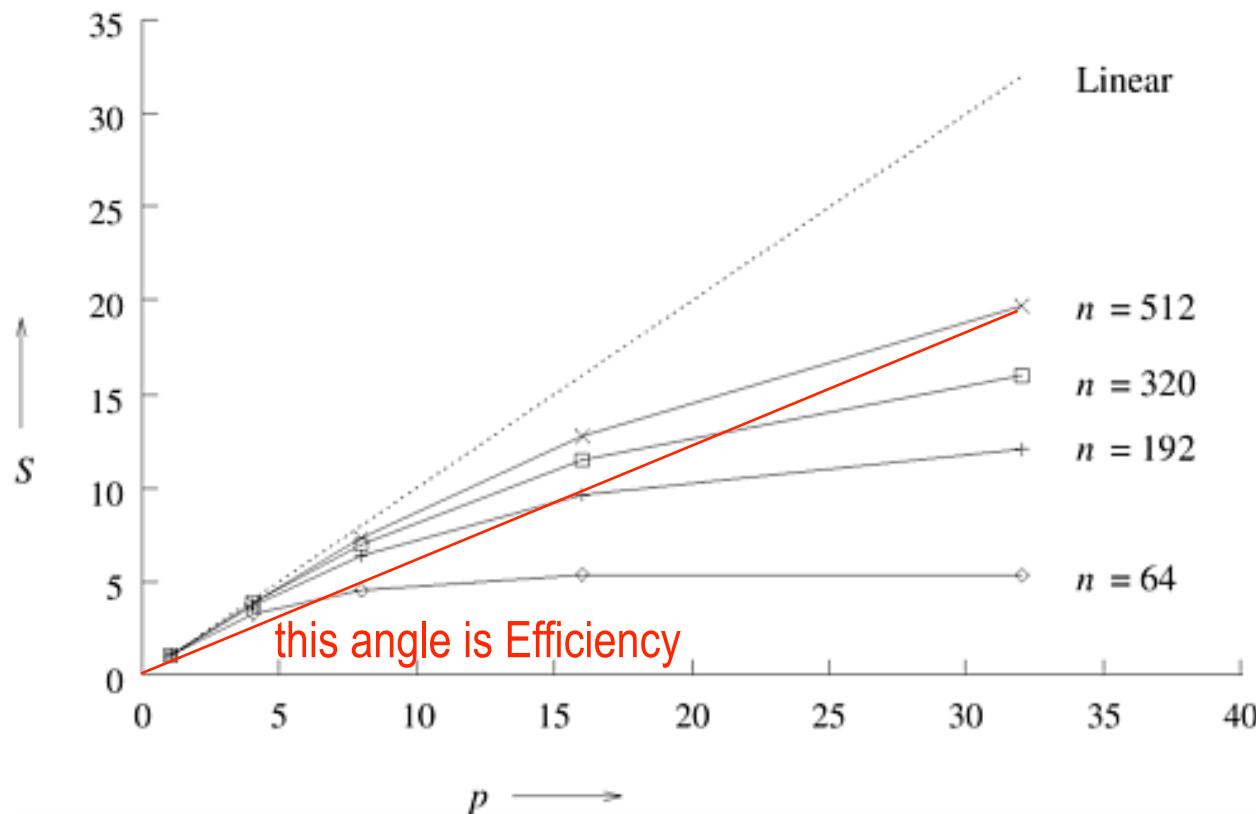
minimum possible parallel time

$$T_p = T_1/p + T_{\text{overhead}}$$

T_1 represents the amount of work will be a function of n

Efficiency: $E(p) = S(p)/p \quad (= 1)$ If $T_{\text{overhead}} > 0$, $E(p) < 1$

A Typical Speedup Curve



Diminishing returns as p increases (efficiency goes down)

T_{overhead} grows slower than T_1 as problem size n increases

Speedup Calculation

$S(p) = p$: linear speedup or ideal speedup

Embarrassingly parallel with no communication overheads

$S(p) < p$: sublinear speedup. Common!

$S(p+1) = S(p) + \delta$, where $\delta < 1$: diminishing returns (efficiency drop)

Idle time due to load unbalance

Overhead due to communication, etc.

Idle time due to synchronization

Extra computation in the algorithm of the parallel program

Many more reasons

$S(p+1) < S(p)$: slowdown (may happen)

Overheads are $O(p)$ but work remains $O(n)$

Contention for a resource depends on p

$S(p) > p$: Fallacy!

Amdahl's Law

In 1960 Gene Amdahl argued

The speedup of a parallel program is inherently limited by its sequential portion (examples?)

1-x fraction of the program can NOT be parallelized

If x fraction of a serial program can be parallelized and it takes time T_1 to run the serial program, then the speedup is at most

$$S(p) = \frac{T_1}{T_p} \leq \frac{T_1}{xT_1/p + (1 - x)T_1} \quad = 1/(x/p+1-x) \leq 1/(1-x)$$

min time for parallelizable part

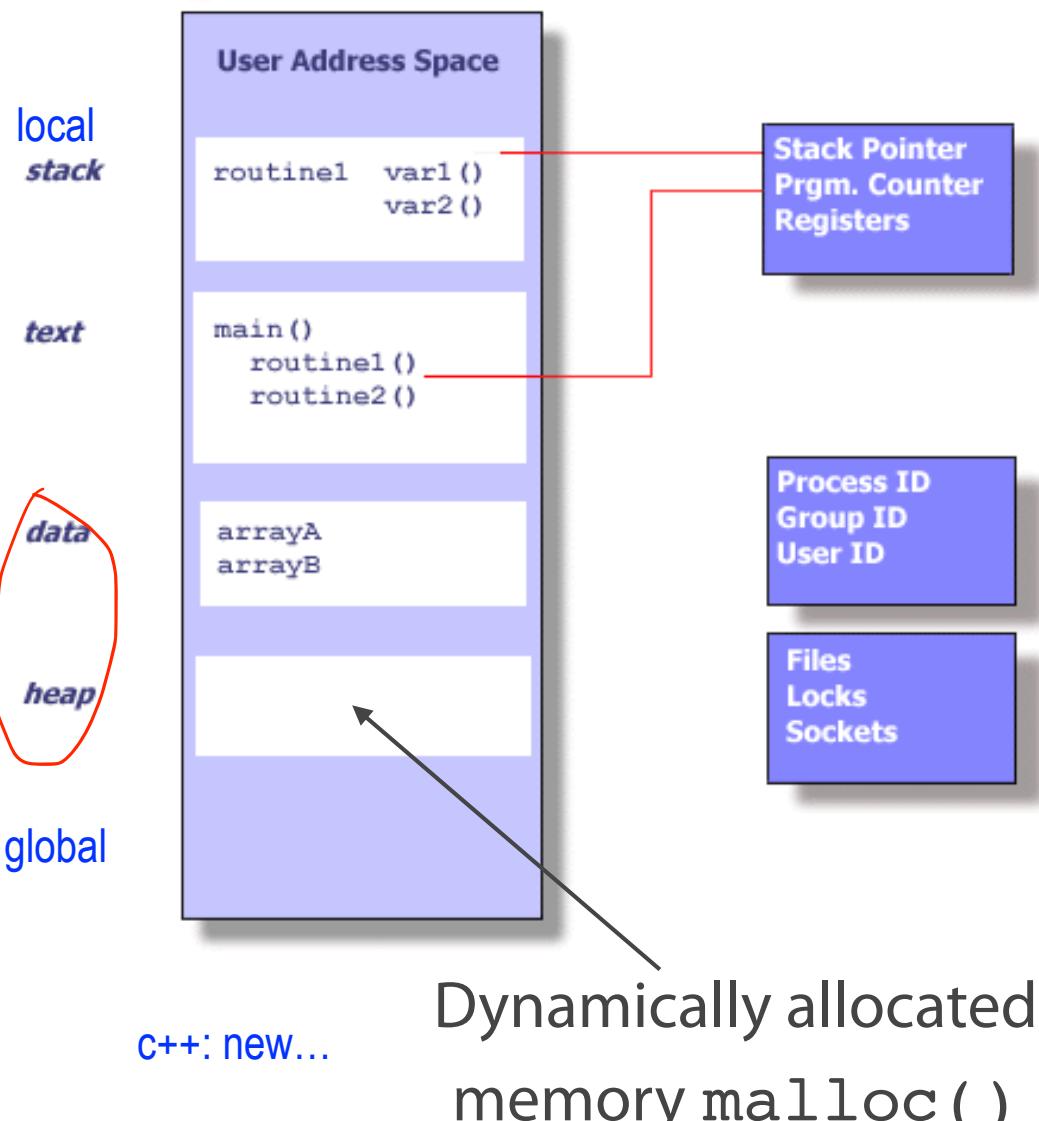
$$\lim_{p \rightarrow \infty} S(p) \leq \frac{1}{1 - x} \quad \text{Does not depend on } p$$

Shared-Memory Programming with Pthreads

Process

Processes contain information about program resources and program execution state, including

Process ID, process group ID, user ID, and group ID, Environment, Working directory, Program instructions, Registers, Stack, Heap, File descriptors, Signal actions, Shared libraries, Inter-process communication tools.



Stack

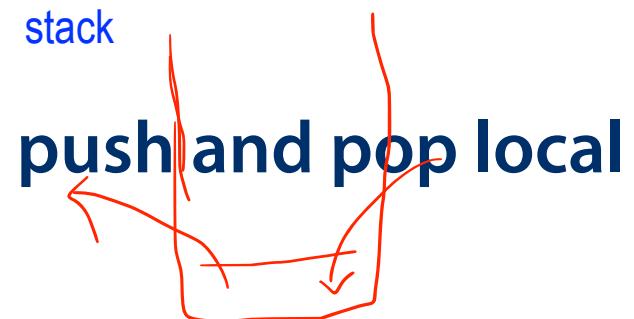
Stack is a special region of your computer's memory that stores temporary variables created by each function (including the main() function).

The stack grows and shrinks as functions push and pop local variables

There is no need to manage the memory yourself, variables are allocated and freed automatically

The stack has size limits

Stack variables only exist while the function that created them, is running



Heap

The heap is a more free-floating region of memory (and is larger).

To allocate memory on the heap, you must use `malloc()` or `calloc()`, which are built-in C functions.

Use `free()` to deallocate that memory once you don't need it any more. If you fail to do this, your program will have memory leak.

Unlike the stack, the heap does not have size restrictions

Heap variables are essentially **global in scope**.

Unlike the stack, variables created on the heap are accessible by any function, anywhere in your program.

Stack vs. Heap

Stack

very fast access

don't have to explicitly de-allocate variables

space is managed efficiently by CPU, memory will not become fragmented

local variables only

limit on stack size (OS-dependent)

variables cannot be resized

Heap

variables can be accessed globally

no limit on memory size

(relatively) slower access

no guaranteed efficient use of space, memory may become fragmented over time as blocks of memory are allocated, then freed

you must manage memory (you're in charge of allocating and freeing variables)

variables can be resized using realloc()

Example: Creating variables on the Stack

```
#include <stdio.h>

double multiplyByTwo (double input) {
    double twice = input * 2.0;    all variables are on the stack
    return twice;
}

int main (int argc, char *argv[ ])
{
    int age = 30;                all variables are on the stack
    double salary = 12345.67;
    double myList[3] = {1.2, 2.3, 3.4};

    printf("double your salary is %.3f\n",
    multiplyByTwo(salary));

    return 0;
}
```

Example: Creating variables on the Heap

```
pointer pointer
double *multiplyByTwo (double *input) {
    double *twice = malloc(sizeof(double));
    *twice = *input * 2.0;
    return twice;
}

int main (int argc, char *argv[ ])
{
    int *age = malloc(sizeof(int));
    *age = 30;

    twiceSalary
    double *salary = malloc(sizeof(double));
    *salary = 12345.67;
    double *myList = malloc(3 * sizeof(double));
    myList[0] = 1.2;
    myList[1] = 2.3;
    myList[2] = 3.4;
}

twice and twiceSalary have the same value,
which is the address
of this memory location
    the pointer (double*) twice

    double *twiceSalary = multiplyByTwo(salary);
    printf("double your salary is %.3f\n", *twiceSalary);

    free(age);
    free(salary);
    free(myList);
    free(twiceSalary);
    return 0;
}
```

twice is Global
even after the function returns, *twice is still in the Heap and thus accessible by *twiceSalary in main()

Diagram: A red box represents memory on the heap. Two arrows point from the variable names 'twice' and 'twiceSalary' to this red box, indicating they both point to the same heap-allocated memory.

frees the memory created by double * twice = malloc(sizeof(double))

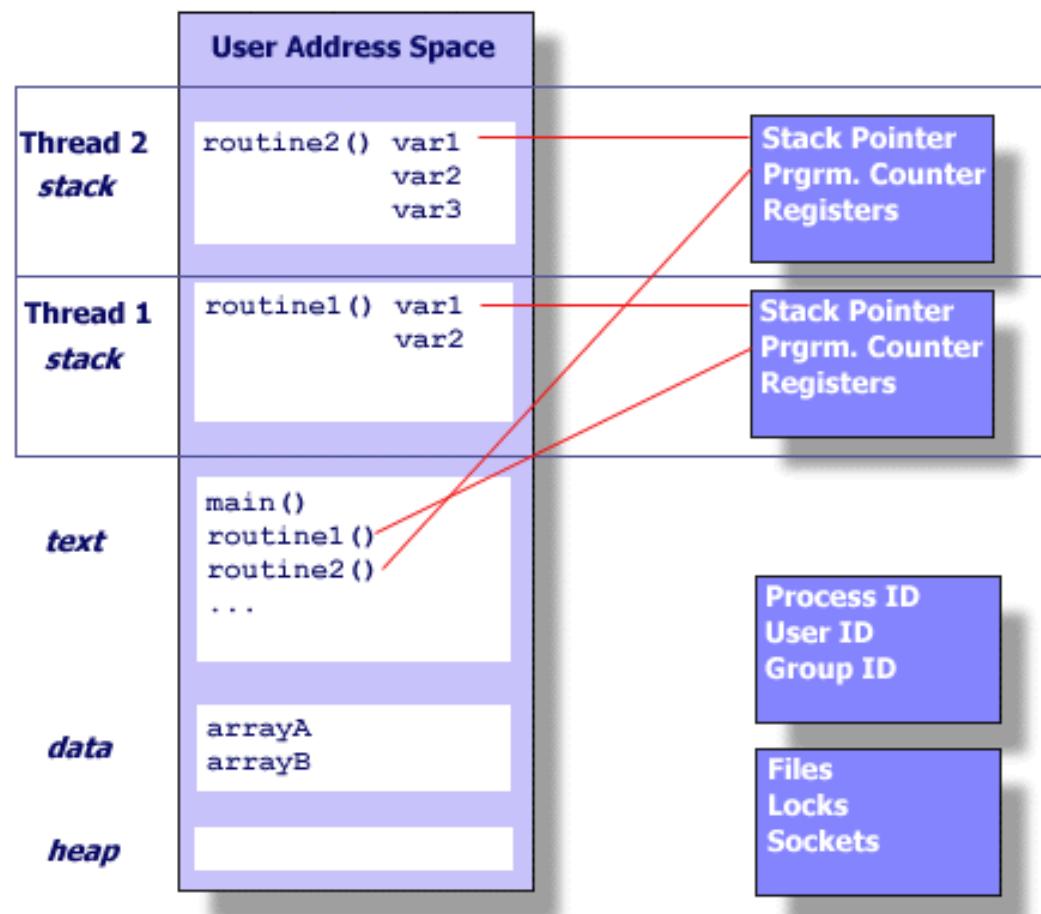
Thread

A "procedure" in a process that runs independently from its main program (can be scheduled by OS).

Threads only maintain essential resources to exist as executable code:

Stack pointer, Registers, Scheduling properties (such as policy or priority), Set of pending and blocked signals, Thread specific data.

Threads are light-weight



POSIX Threads (Pthreads)

For UNIX systems, a standard programming interface has been specified by the IEEE POSIX 1003.1c standard (1995).

Why Pthreads?

Lightweight, faster startup: fork() vs. pthread_create()

Efficient Communication

- No need to copy data from process to process (like in fork())

- No data transfer, because threads share the address space in a process

- Pthreads communications are mostly memory-to-CPU which is faster

Hides memory and I/O latency on a single core processor

Priority/real-time scheduling

Asynchronous event handling

- Examples: a web server, video conferencing, an OS

Design Multi-threaded Programs

Working Modes

Manager/worker: a single thread, the manager assigns work to other threads, the workers. Static/dynamic worker pool.

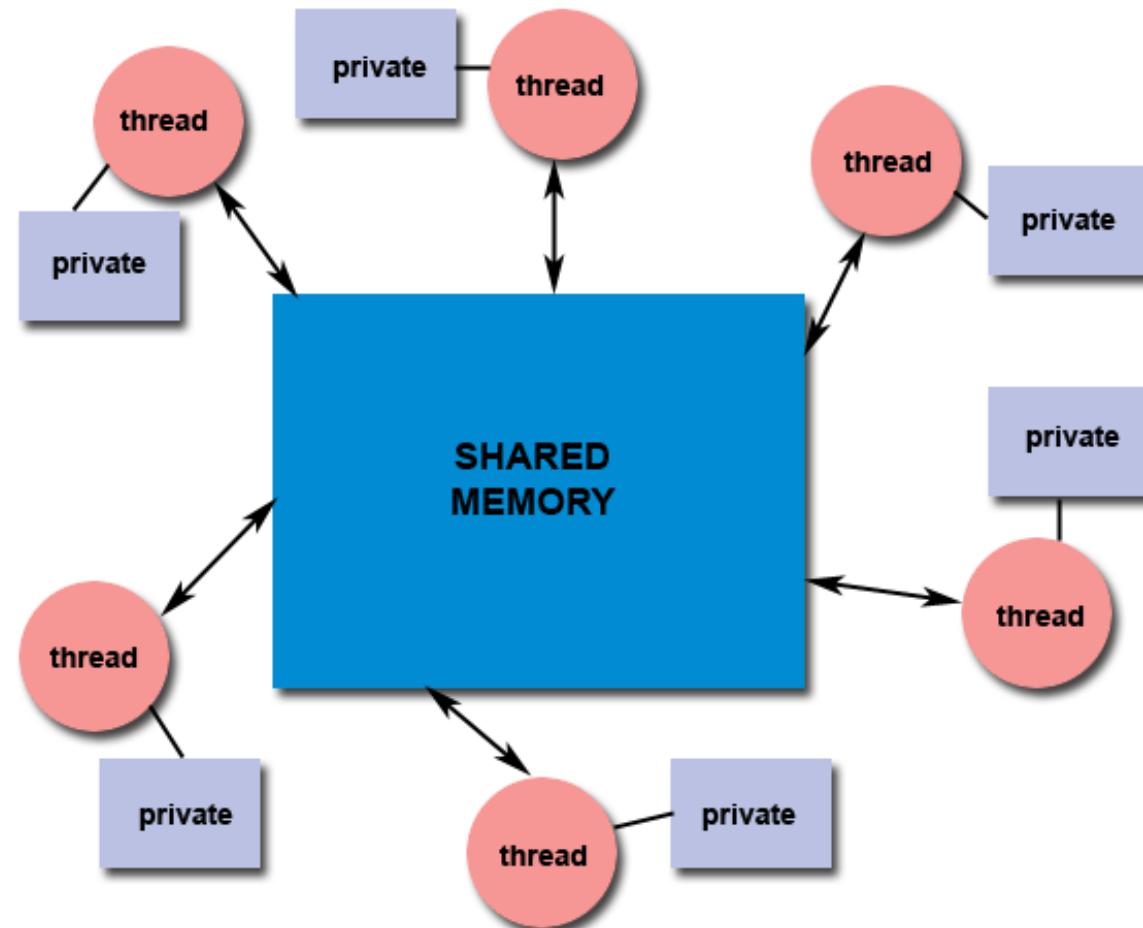
Peer: similar to the manager/worker model, but after the main thread creates other threads, it participates in the work.

Pipeline: a task is broken into a series of suboperations, each of which is handled in series, but concurrently, by a different thread. An automobile assembly line best describes this model.

Shared Memory Model

All threads have access to the same global, shared memory

Threads also have their own private data



Pthreads: Hello, World

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

const int MAX_THREADS = 64;

/* Global variable: accessible to all threads */

int thread_count;

void *Hello(void* rank); /* Thread function */

int main(int argc, char* argv[]) {

    long      thread; /* Use long in case of a 64-bit system */

    pthread_t* thread_handles;

    /* Get number of threads from command line */

    thread_count = strtol(argv[1], NULL, 10);

    thread_handles = malloc (thread_count*sizeof(pthread_t));
}
```

Pthreads: Hello, World

...

```
for (thread = 0; thread < thread_count; thread++)  
    pthread_create(&thread_handles[thread], NULL,  
        Hello, (void*) thread);  
  
printf("Hello from the main thread\n");  
  
for (thread = 0; thread < thread_count; thread++)  
    pthread_join(thread_handles[thread], NULL);  
  
free(thread_handles);  
return 0;  
} /* main */
```

Pthreads: Hello, World

```
void *Hello(void* rank) {  
    long my_rank = (long) rank; /* Use long in case of 64-bit system */  
    printf("Hello from thread %ld of %d\n", my_rank, thread_count);  
    return NULL;  
} /* Hello */
```

Create a Thread

```
int pthread_create(  
    pthread_t*          thread_p           /* out */  
    const pthread_attr_t* attr_p            /* in */  
    void*               (*start_routine)(void*) /* in */  
    void*               arg_p              /* in */);
```

The function started by `pthread_create` should have a prototype like:

```
void* thread_function(void* args_p);
```

`void*` can be cast to any pointer type in C, so `args_p` can point to a list containing one or more values needed by `thread_function`

We have cast the loop variable `thread` to have type `void*`.

Then in the thread function, `hello`, we have cast the argument back to a `long`.

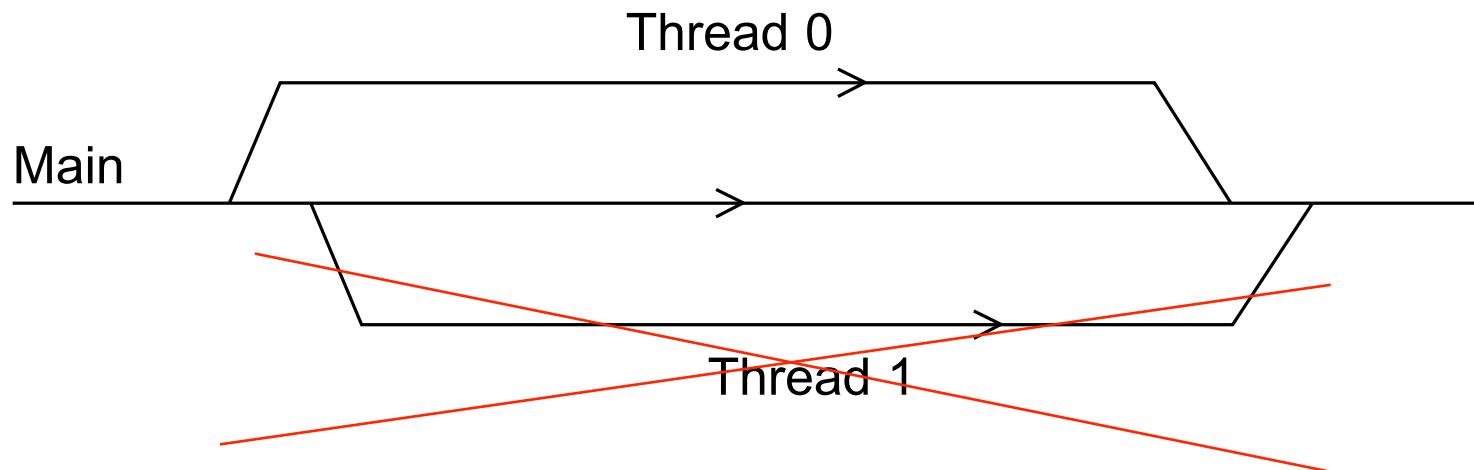
Stopping the Threads

```
int pthread_join(  
    pthread_t           thread      /* in */  
    void**             ret_val_ptr /* out */);
```

A call to `pthread_join` will suspend the calling thread until the thread associated with the `pthread_t` object terminates.

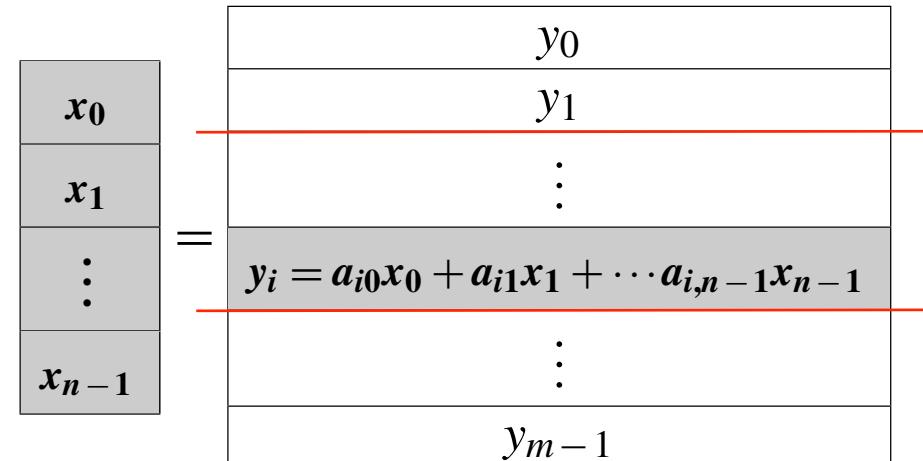
The second argument can be used to receive any return value computed by the thread.

The scheduling of the threads is usually done by the OS.



Example: Matrix-Vector Multiplication

a_{00}	a_{01}	\cdots	$a_{0,n-1}$
a_{10}	a_{11}	\cdots	$a_{1,n-1}$
\vdots	\vdots		\vdots
a_{i0}	a_{i1}	\cdots	$a_{i,n-1}$
\vdots	\vdots		\vdots
$a_{m-1,0}$	$a_{m-1,1}$	\cdots	$a_{m-1,n-1}$



$$y_i = \sum_{j=0}^{n-1} a_{ij}x_j.$$

```

/* For each row of A */
for (i = 0; i < m; i++) {
    y[i] = 0.0;
    /* For each element of the row and each element of x */
    for (j = 0; j < n; j++)
        y[i] += A[i][j]* x[j];
}

```

Example: Matrix Vector Multiplication

Let each thread compute multiple $y[i]$

To compute $y[i]$, the thread that takes care of it needs to run the code

```
y[i] = 0.0;  
for (j = 0; j < n; j++)  
    y[i] += A[i][j]*x[j];
```

Just need to decide which rows to be assigned to which thread

Assume m (# rows) and n (# columns) are divisible by t (# threads).

Each thread handles m/t rows.

Thread q handles the rows with indices $q \times m/t - (q+1) \times m/t - 1$

Matrix-Vector Multiplication

```
/*
 * Function:          Pth_mat_vect
 * Purpose:          Multiply an mxn matrix by an nx1 column vector
 * In arg:           rank
 * Global in vars:  A, x, m, n, thread_count
 * Global out var:  y
 */
void *Pth_mat_vect(void* rank) {
    long my_rank = (long) rank;
    int i, j;
    int local_m = m/thread_count;
    int my_first_row = my_rank*local_m;
    int my_last_row = (my_rank+1)*local_m - 1;

    for (i = my_first_row; i <= my_last_row; i++) {
        y[i] = 0.0;
        for (j = 0; j < n; j++)
            y[i] += A[i*n+j]*x[j];
    }

    return NULL;
} /* Pth_mat_vect */
```

Passing Arguments to the Thread Function

```
long t;

for(t=0; t<NUM_THREADS; t++)
{
    printf("Creating thread %ld\n", t);
    rc = pthread_create(&threads[t], NULL, PrintHello, (void *) t);
    ...
}
```

Correct!

```
int rc;
long t;

for(t=0; t<NUM_THREADS; t++)
{
    printf("Creating thread %ld\n", t);
    rc = pthread_create(&threads[t], NULL, PrintHello, (void *) &t);
    ...
}
```

(not recommended in pthreads, need to watch out for the lifetime of t).
when you try to dereference &t later

Stack vs. Heap

Stack

very fast access

don't have to explicitly de-allocate variables

space is managed efficiently and automatically by CPU

local variables only

limit on stack size (OS-dependent)

variables cannot be resized

Heap

variables can be accessed globally

no limit on memory size

(relatively) slower access

memory may become fragmented over time

you must manage memory (you're in charge of allocating and freeing variables)

variables can be resized using realloc()

Global vs. Local Variables

Global

Global variables (declared outside any function)

Dynamically allocated variables on the heap [using malloc](#)

Local

Variables on stack (declared in each function)

Watch out for the pointer and dereferencing!

Where are variables stored?

Global variables -----> data

Static variables -----> data

Constant data types -----> code and/or data.

Consider string literals for a situation when a constant itself would be stored in the data segment, and references to it would be embedded in the code

Local variables(declared and defined in functions) -----> stack

Variables declared and defined in main function -----> also stack

Pointers (ex: char *arr, int *arr) -----> data or stack

depending on the context. C lets you declare a global or a static pointer, in which case the pointer itself would end up in the data segment.

Dynamically allocated space(using malloc, calloc, realloc) -----> heap

<https://stackoverflow.com/questions/14588767/where-in-memory-are-my-variables-stored-in-c>

Passing Multiple Arguments (via struct)

```
struct thread_data{
    int thread_id;
    int sum;
    char *message;
};

struct thread_data thread_data_array[NUM_THREADS];

void *PrintHello(void *threadarg)
{
    struct thread_data *my_data;
    ...
    my_data = (struct thread_data *) threadarg;
    taskid = my_data->thread_id;
    sum = my_data->sum;
    hello_msg = my_data->message;
    ...
}

int main (int argc, char *argv[])
{
    ...
    thread_data_array[t].thread_id = t;
    thread_data_array[t].sum = sum;
    thread_data_array[t].message = messages[t];
    rc = pthread_create(&threads[t], NULL, PrintHello,
        (void *) &thread_data_array[t]);
    ...
}
```

not very common!
more like Message Passing

Terminating Threads

Several ways in which a thread may be terminated:

The thread returns normally from its starting routine.

The thread makes a call to the `pthread_exit(status)`

The entire process is terminated due to making a call to either the `return` or `exit()`

If `main()` finishes first, without calling `pthread_exit()` explicitly itself (but `return` is called implicitly to terminate the process).

In subroutines that execute to completion normally, you can often omit calling `pthread_exit()`

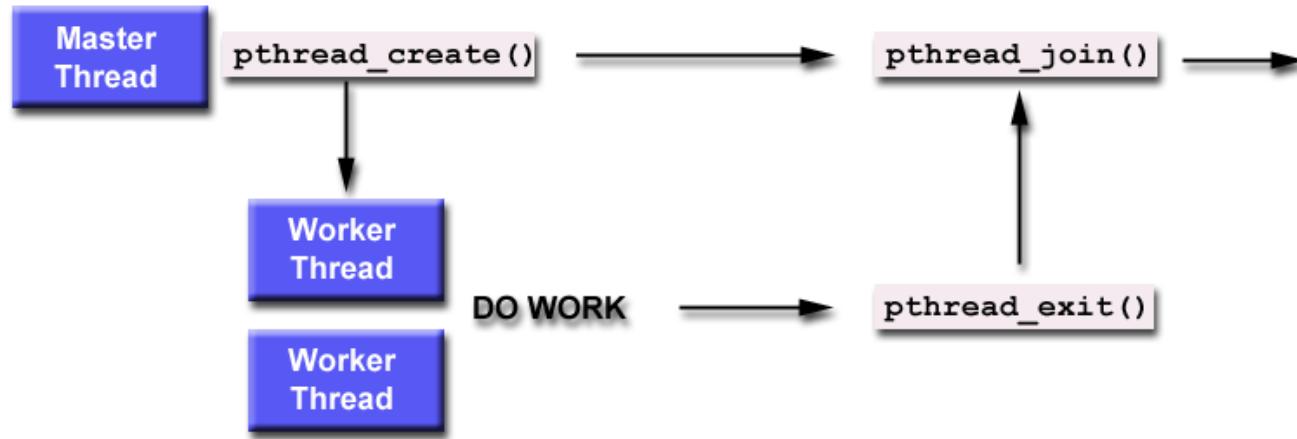
By having `main()` explicitly call `pthread_exit()` as the last thing it does, the process will be kept alive to support the threads it created until they are all done.

Terminating Threads

```
void *PrintHello(void *threadid)
{
    long tid;
    tid = (long)threadid;
    printf("Hello World! It's me, thread #%ld!\n", tid);
    pthread_exit(NULL); "return" is not required in a subroutine
}

int main (int argc, char *argv[])
{
    pthread_t threads[NUM_THREADS];
    int rc;
    long t;
    for(t=0; t<NUM_THREADS; t++){
        printf("In main: creating thread %ld\n", t);
        rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
        if (rc){
            printf("ERROR; return code from pthread_create() is %d\n", rc);
            exit(-1);
        }
    } Join is not required
        Join is a way to synchronize, not a way to terminate a thread
/* Last thing that main() should do */
pthread_exit(NULL);
}
```

Joining - A way of synchronization



The `pthread_join()` subroutine blocks the calling thread until the specified thread terminates.

```
int pthread_join(pthread_t thread, &var  
void *value_ptr);  
void pthread_exit(void *value_ptr);  
value_ptr in pthread_join() will obtain value from the argument  
passed to pthread_exit()
```

`&var`
`void **value_ptr` ;
call by reference for `(void*)`
used as an output instead of input

Joining - A way of synchronization

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define NUM_THREADS 4

void *BusyWork(void *t)
{
    int i;
    long tid;
    double result=0.0;
    tid = (long)t;
    printf("Thread %ld starting...\n",tid);
    for (i=0; i<1000000; i++)
    {
        result = result + sin(i) * tan(i);
    }
    printf("Thread %ld done. Result = %e\n",tid, result);
    pthread_exit((void*) t);
}
```

passing (void*) threadID into pthread_exit()

Joining - A way of synchronization

```
int main (int argc, char *argv[ ])
{
    pthread_t thread[NUM_THREADS];
    pthread_attr_t attr;
    int rc;
    long t;
    void *status;

    /* Initialize and set thread detached attribute */
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

    for(t=0; t<NUM_THREADS; t++) {
        printf("Main: creating thread %ld\n", t);
        rc = pthread_create(&thread[t], &attr, BusyWork, (void *)t);
        if (rc) {
            printf("ERROR; return code from pthread_create()
                   is %d\n", rc);
            exit(-1);
        }
    }
}
```

Joining - A way of synchronization

```
/* Free attribute and wait for the other threads */
pthread_attr_destroy(&attr);
for(t=0; t<NUM_THREADS; t++) {
    rc = pthread_join(thread[t], &status);      status will be updated by (void*) t
    if (rc) {                                &(void *) -> void **
        printf("ERROR; return code from pthread_join()
                is %d\n", rc);
        exit(-1);
    }
    printf("Main: completed join with thread %ld having a status
           of %ld\n",t,(long)status);
}

printf("Main: program completed. Exiting.\n");
pthread_exit(NULL);
}
```

Critical Section and Race Condition

Example of Race Condition

expecting: $\text{Balance} = \text{Balance} + 40$

Two threads both want to execute: $\text{Balance} = \text{Balance} + 20$.

A possible sequence of instructions

Thread 1 reads Balance of 1000

Thread 2 reads Balance of 1000

Thread 1 adds 20 to Balance and gets a new Balance of 1020

Thread 2 adds 20 to Balance and gets a new Balance of 1020 (Wrong!)

The code “ $\text{Balance} = \text{Balance} + 20$ ” is a critical section

When multiple threads access a shared resource such as a shared variable or a file, at least one of the accesses is a write, and the accesses can result in an error, we have a race condition.

Mutex (Mutual Exclusion)

Guarantee that one thread “excludes” all other threads while it executes the critical section.

Mutex: a variable of type

`pthread_mutex_t`

Initialize a mutex variable

```
int pthread_mutex_init(  
    pthread_mutex_t*           mutex_p /* out */,  
    const pthread_mutexattr_t* attr_p  /* in */);
```

Destroy a mutex variable

```
int pthread_mutex_destroy(pthread_mutex_t* mutex_p);
```

Mutex (Mutual Exclusion)

Before entering a critical section, a thread should call

```
int pthread_mutex_lock(pthread_mutex_t* mutex_p);
```

Done with the code in a critical section, a thread should call

```
int pthread_mutex_unlock(pthread_mutex_t* mutex_p);
```

Sequence of actions

Several threads attempt to lock the mutex they may arrive at the same time

Only one succeeds and owns the mutex

(When several threads try to “lock”, the losers block at that call)

The owner thread performs some set of actions

The owner unlocks the mutex

Another thread acquires the mutex and repeats the process

Finally the mutex is destroyed

Mutex Example: Calculating Pi

$$\pi = 4 \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots + (-1)^n \frac{1}{2n+1} + \cdots \right).$$

Serial Code

```
double factor = 1.0;
double sum = 0.0;
for (i = 0; i < n; i++, factor = -factor)
{
    sum += factor / (2*i+1);
}
pi = 4.0 * sum;
```

Mutex Example: Calculating Pi

Pthreads code

```
/*-----*/  
void* Thread_sum(void* rank) {  
    long my_rank = (long) rank;  
    double factor;  
    long long i;  
    long long my_n = n/thread_count;  
    long long my_first_i = my_n*my_rank;  
    long long my_last_i = my_first_i + my_n;  
    double my_sum = 0.0;  
  
    if (my_first_i % 2 == 0)  
        factor = 1.0;  
    else  
        factor = -1.0;  
  
    for (i = my_first_i; i < my_last_i; i++, factor = -factor) {  
        my_sum += factor/(2*i+1);  
    }  
    pthread_mutex_lock(&mutex);  
    sum += my_sum;  
    pthread_mutex_unlock(&mutex);  
  
    return NULL;  
} /* Thread_sum */
```

Semaphores

Used for Producer-Consumer Synchronization

Suppose we have t threads and we want thread 0 to send a message to thread 1, thread 1 to send a message to thread 2, ..., thread $t-1$ to send a message to thread 0.

To simulate the consumer by printing

After a thread has received the message, it prints the message.

Implementation

Allocate a global array of char^*



Only one commodity
to consume at a time

Each thread initializes the message locally and set a pointer in the global to refer to it.

Need to make sure that each thread prints its message **after** it has received it (not before).

Busy-waiting is a solution, but consumes CPU cycle.

Semaphore Usage

Semaphores can be deemed as a special type of `unsigned int`

```
#include <semaphore.h>
```

Initialize

```
int sem_init(sem_t* semaphore_p, int shared, unsigned init_val);  
//shared==0: shared between threads, init_val: the initial value
```

Destroy

```
int sem_destroy(sem_t* semaphore_p);
```

Wait

```
int sem_wait(sem_t* semaphore_p);  
//Block if the semaphore is 0; otherwise, decrement the semaphore and proceed
```

Unlock

```
int sem_post(sem_t* semaphore_p);  
//Just increment the semaphore
```

Semaphore Examples

Solving the message sending task (synchronization) using

Busy Waiting

Semaphores

See the 2 sample programs on eClass

Pthreads: Sending messages (via busy waiting)

Pthreads: Sending messages (via semaphores)

Mutex vs. Semaphores

Mutex

A binary mutual exclusion semaphore (with a count of one)

Can only be released by the same thread that locked it

Usually used to avoid race conditions

Semaphores

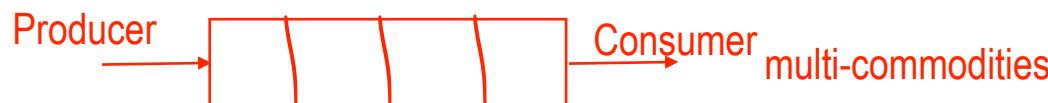
Has a count, which could be more than one

Can be locked by *count* number of threads concurrently

You have a number of resource instances to lock (e.g., 3 drives)

Can be unlocked by any thread (doesn't have to be the thread that locked it). posted
count of semaphore depends on how many resources to consume

Usually used for producer-consumer synchronization



Condition Variables

A condition variable is a data object that allows a thread to suspend execution (block-waiting) until a certain event or condition occurs.

signaling mechanism

When the event or condition occurs, another thread can signal the thread to “wake up.”

A condition variable is always used with a mutex to protect the “condition”.

blocking thread reads the condition
the other thread alters the condition

Initialization and Destroy

```
int pthread_cond_init(pthread_cond_t *cond,  
                      const pthread_condattr_t *attr);
```

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

Condition Variables

Wait: blocks the calling thread until condition is signalled

Shall be called with mutex locked by the calling thread

```
int pthread_cond_wait(pthread_cond_t *cond,  
                      pthread_mutex_t *mutex);
```

Upon successful completion, a value of zero shall be returned

Signal: unblock one of the threads blocked on the condition variable

Shall be called with mutex locked by the calling thread

```
int pthread_cond_signal(pthread_cond_t *cond);
```

Broadcast: unblock all threads blocked on the condition variable

Shall be called with mutex locked by the calling thread

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Condition Variables Standard Usage

The blocking thread typically looks like

```
// safely examine the condition, prevent other threads from altering it
pthread_mutex_lock (&lock);
while (SOME-CONDITION is false) // to avoid spurious wakeup
    pthread_cond_wait (&cond, &lock);

// Do whatever you need to do when condition becomes true
do_stuff();
pthread_mutex_unlock (&lock); As long as the mutex protects the Condition
```

A thread, notifying the condition variable, typically looks like

```
// ensure we have exclusive access to whatever comprises the condition
pthread_mutex_lock (&lock);

ALTER-CONDITION

// Wakeup any threads that may be waiting on the condition
pthread_cond_signal (&cond);

// allow others to proceed
pthread_mutex_unlock (&lock)
```

Condition Variables

Wait: blocks the calling thread until condition is signalled

Shall be called with mutex locked by the calling thread

```
int pthread_cond_wait(pthread_cond_t *cond,  
                      pthread_mutex_t *mutex);
```

Actually this “wait” automatically does the following:

```
pthread_mutex_unlock(&mutex);
```

```
block waiting on the signal (&cond);
```

```
pthread_mutex_lock(&mutex);
```

See the sample code on eClass

“Pthreads: A condition variable example”

Barriers

Sometimes, threads need to “meet up” (make sure they are all at the same point.)

Multiplication of multiple matrices

Time measurement for a certain phase of code

We'll look at barrier implementation through

Barriers (pthread_barrier_t)

Busy waiting and a mutex

Semaphores

Condition Variables (*the best way*)

Barrier through pthread_barrier_t

```
double initial_matrix[ROWS][COLS];
double final_matrix[ROWS][COLS];
pthread_barrier_t barr; // Barrier variable

extern void DotProduct(int row, int col,
                      double source[ROWS][COLS],
                      double destination[ROWS][COLS]);
extern double determinant(double matrix[ROWS][COLS]);

void * entry_point(void *arg)
{
    int rank = (int)arg;
    for(int row = rank * ROWS / THREADS; row < (rank + 1) * THREADS; ++row)
        for(int col = 0; col < COLS; ++col)
            DotProduct(row, col, initial_matrix, final_matrix);

    // Synchronization point
    int rc = pthread_barrier_wait(&barr);

    if(rc != 0 && rc != PTHREAD_BARRIER_SERIAL_THREAD)
    {
        printf("Could not wait on barrier\n");
        exit(-1);
    }
    for(int row = rank * ROWS / THREADS; row < (rank + 1) * THREADS; ++row)
        for(int col = 0; col < COLS; ++col)
            DotProduct(row, col, final_matrix, initial_matrix);
}
```

Barrier through pthread_barrier_t

```
int main(int argc, char **argv)
{
    pthread_t thr[THREADS];

    // Barrier initialization
    if(pthread_barrier_init(&barr, NULL, THREADS)) // THREADS is the number of threads to be synched on barrier
    {
        printf("Could not create a barrier\n");
        return -1;
    }

    for(int i = 0; i < THREADS; ++i)
    {
        if(pthread_create(&thr[i], NULL, &entry_point, (void*)i))
        {
            printf("Could not create thread %d\n", i);
            return -1;
        }
    }

    for(int i = 0; i < THREADS; ++i)
    {
        if(pthread_join(thr[i], NULL))
        {
            printf("Could not join thread %d\n", i);
            return -1;
        }
    }

    double det = Determinant(initial_matrix);
    printf("The determinant of M^4 = %f\n", det);

    return 0;
}
```

Barriers via Busy Waiting and a Mutex

```
/* Shared and initialized by the main thread */

int count;      // Initialize to 0

int thread_count;

pthread_mutex_t barrier_mutex;

...

void* Thread_func(void* ...){

    ...

    /* Barrier */

    pthread_mutex_lock(&barrier_mutex);

    count++;

    pthread_mutex_unlock(&barrier_mutex);

    while (count<thread_count);      // Busy waiting

    ...

}
```

Not efficient and hard to reuse the same count for a 2nd barrier

Barrier through Semaphores

```
/* Shared variables */
int counter;      // Initialize to 0
sem_t count_sem; // Initialize to 1      serves as a mutex
sem_t barrier_sem; // Initialize to 0      serves as a barrier
...
void* Thread_func(void* ...){
    ...
    /*Barrier*/
    sem_wait(&count_sem);
    if (counter == thread_count-1){
        counter = 0;
        sem_post(&count_sem);
        for (j = 0; j < thread_count-1; j++)      last thread
            sem_post(&barrier_sem);
    } else {
        counter++;
        sem_post(&count_sem);
        sem_wait(&barrier_sem);      release the ones blocked at sem_wait()
    }
}
```

Hard to reuse the same variables for a 2nd barrier

Barrier via a Condition Variable

```
/* Shared */
int counter = 0;
pthread_mutex_t mutex;
pthread_cond_t cond_var;

...
void* Thread_func(void* arg){
    ...
    /* Barrier */
    pthread_mutex_lock(&mutex);
    counter++;
    if (counter == thread_count){
        counter = 0;
        pthread_cond_broadcast(&cond_var);          to wake up everything
                                                    blocked on cond_var
    } else {
        while (pthread_cond_wait(&cond_var, &mutex) != 0);  another way to call pthread_cond_wait()
    }
    pthread_mutex_unlock(&mutex);
    ...
}
```

The most important implementation of barrier

Composite Synchronization Constructs

Barrier as a Composite Construct

```
typedef struct {
    pthread_mutex_t count_lock;
    pthread_cond_t ok_to_proceed;
    int count;
} mylib_barrier_t;

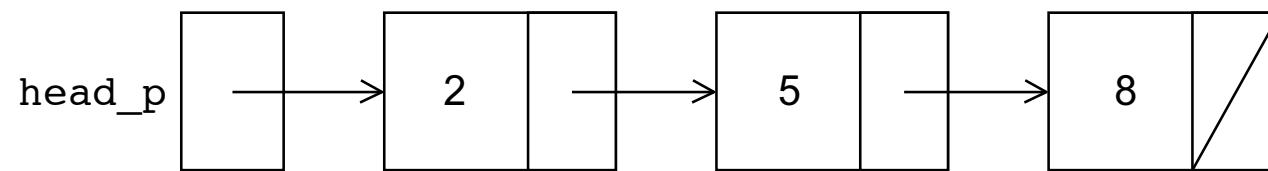
void mylib_init_barrier(mylib_barrier_t *b) {
    b -> count = 0;
    pthread_mutex_init(&(b -> count_lock), NULL);
    pthread_cond_init(&(b -> ok_to_proceed), NULL);
}

void mylib_barrier (mylib_barrier_t *b, int num_threads) {
    pthread_mutex_lock(&(b -> count_lock));
    b -> count++;
    if (b -> count == num_threads) {
        b -> count = 0;
        pthread_cond_broadcast(&(b -> ok_to_proceed));
    }
    else
        while (pthread_cond_wait(&(b -> ok_to_proceed),
            &(b -> count_lock)) != 0);
    pthread_mutex_unlock(&(b -> count_lock));
}
```

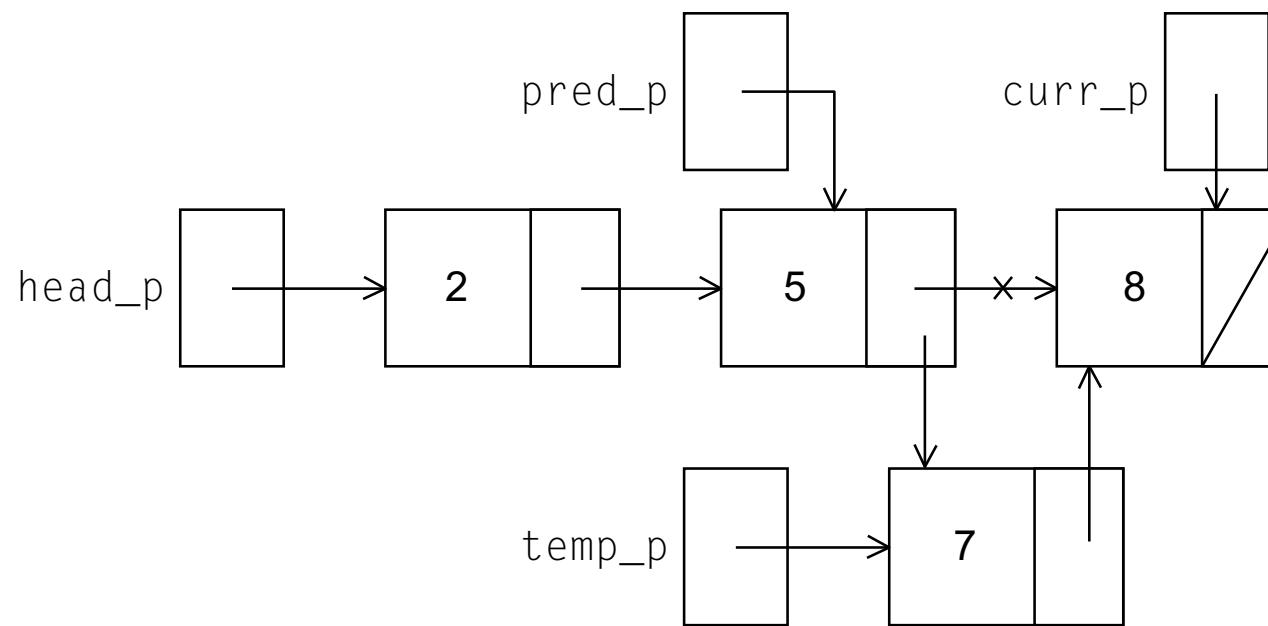
Read-Write Locks

In many applications, a data structure is read frequently but written infrequently. For such applications, we should use read-write locks.

A Linked List

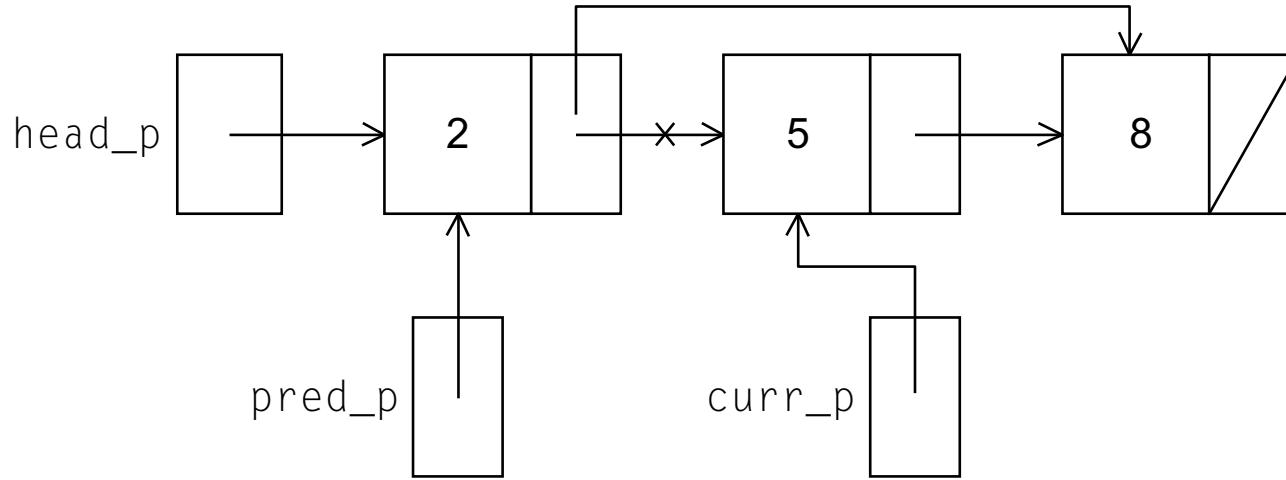


Insert

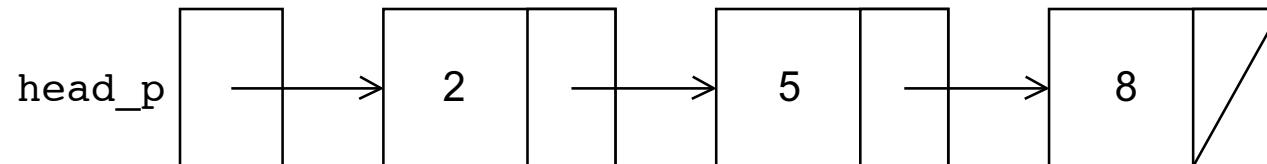


Read-Write Locks

Delete



Member



Read-Write Locks

Implementation Requirements

1. A read lock can be granted when there are other threads that already have read locks.
2. If there is a write lock on the data (or if there are queued write locks), the thread performs a condition wait (no matter whether it's a read or write).
3. If there are multiple threads requesting a write lock, they must perform a condition wait. and one of them acquires the lock if there is no reader

With this description, we can design functions for **read locks** `mylib_rwlock_rlock`, **write locks** `mylib_rwlock_wlock`, and **unlocking** `mylib_rwlock_unlock`. unlock is shared between read lock and write lock

Read-Write Locks vs. Mutex Locks

Run times Comparison

slowest

Table 4.3 Linked List Times: 1000 Initial Keys, 100,000 ops,
99.9% Member, 0.05% Insert, 0.05% Delete

Implementation	Number of Threads			
	1	2	4	8
Read-Write Locks	0.213	0.123	0.098	0.115
One Mutex for Entire List	0.211	0.450	0.385	0.457
One Mutex per Node	1.680	5.700	3.450	2.700

slowest

Table 4.4 Linked List Times: 1000 Initial Keys, 100,000 ops,
80% Member, 10% Insert, 10% Delete

Implementation	Number of Threads			
	1	2	4	8
Read-Write Locks	2.48	4.97	4.69	4.71
One Mutex for Entire List	2.50	5.13	5.04	5.11
One Mutex per Node	12.00	29.60	17.00	12.00

Read-Write Locks

Application: computing the minimum of a list of numbers

```
void *find_min_rw(void *list_ptr) {
    int *partial_list_pointer, my_min, i;
    my_min = MIN_INT;
    partial_list_pointer = (int *) list_ptr;
    for (i = 0; i < partial_list_size; i++)
        if (partial_list_pointer[i] < my_min)
            my_min = partial_list_pointer[i];
    /* lock the mutex associated with minimum_value and update the
variable as required */
    mylib_rwlock_rlock(&read_write_lock);
    if (my_min < minimum_value) {  minimum_value is the critical section
        mylib_rwlock_unlock(&read_write_lock);  shared unlock
        mylib_rwlock_wlock(&read_write_lock);
        if (my_min < minimum_value)  always need another if check here
            minimum_value = my_min;
    }
    /* and unlock the mutex */
    mylib_rwlock_unlock(&read_write_lock);
    pthread_exit(0);
}
```

find my_min from the thread's local list

Read-Write Locks

The lock data type `mylib_rwlock_t` holds the following:

a count of the number of readers,

the `writer` (a 0/1 integer specifying whether a writer is present)

a condition variable `readers_proceed` that is signaled when readers can proceed,

a condition variable `writer_proceed` that is signaled when ***one of the writers*** can proceed,

a count `pending_writers` of pending writers, and

`pending_readers` is not necessary

a mutex `read_write_lock` associated with the shared data structure.

Read-Write Locks

```
typedef struct {
    int readers;
    int writer;
    pthread_cond_t readers_proceed;
    pthread_cond_t writer_proceed;
    int pending_writers;
    pthread_mutex_t read_write_lock;
} mylib_rwlock_t;

void mylib_rwlock_init (mylib_rwlock_t *l) {
    l -> readers = l -> writer = l -> pending_writers = 0;
    pthread_mutex_init(&(l -> read_write_lock), NULL);
    pthread_cond_init(&(l -> readers_proceed), NULL);
    pthread_cond_init(&(l -> writer_proceed), NULL);
}
```

Read-Write Locks

```
void mylib_rwlock_rlock(mylib_rwlock_t *l) {
    /* if there is a write lock or pending writers, perform
condition wait, else increment count of readers and grant
read lock */
    mutex
    pthread_mutex_lock(&(l -> read_write_lock));
    while (((l -> pending_writers > 0) || (l -> writer > 0))
        pthread_cond_wait(&(l -> readers_proceed),
        &(l -> read_write_lock));
    l -> readers++;
    pthread_mutex_unlock(&(l -> read_write_lock));
}
```

Priority of pending_writer is greater than reader
Reader will block if there is a pending writer

Read-Write Locks

```
void mylib_rwlock_wlock(mylib_rwlock_t *l) {  
    /* if there are readers or writers, increment pending  
writers count and wait. On being woken, decrement pending  
writers count and increment writer count */  
  
    pthread_mutex_lock(&(l -> read_write_lock));  
    while (((l -> writer > 0) || (l -> readers > 0)) {  
        l -> pending_writers++;  
        pthread_cond_wait(&(l -> writer_proceed),  
                          &(l -> read_write_lock));  
        l -> pending_writers--;  
    } // writer = 0 before the increment  
    l -> writer++;  
    pthread_mutex_unlock(&(l -> read_write_lock));  
}
```

a pair

Read-Write Locks

```
void mylib_rwlock_unlock(mylib_rwlock_t *l) {
    /* if there is a write lock then unlock, else if there are
     * read locks, decrement count of read locks. If the count is 0
     * and there is a pending writer, let it through, else if there
     * are pending readers, let them all go through */
unlock is shared by both reader and writer    Whenever there is a pending writer, its priority is higher than reader
                                                unless there are some reads in the critical section.
    pthread_mutex_lock(&(l -> read_write_lock));
    if (l -> writer > 0)
        l -> writer = 0;
    else if (l -> readers > 0)
        l -> readers--;
    if (l -> readers > 0)
        pthread_cond_broadcast(&(l -> readers_proceed));
    else if ((l -> readers == 0) && (l -> pending_writers > 0))
        pthread_cond_signal(&(l -> writer_proceed));
    else if ((l -> readers == 0) && (l -> pending_writers==0))
        pthread_cond_broadcast(&(l -> readers_proceed));
    pthread_mutex_unlock(&(l -> read_write_lock));
}
```

How Caches Affect Shared-Memory Programming?

Memory Hierarchy Review

Block (aka cache line): unit of copying

minimum unit for data movement

May be multiple words
(a word = 64 bits or 32 bytes) 4 bytes

If accessed data is present in upper level

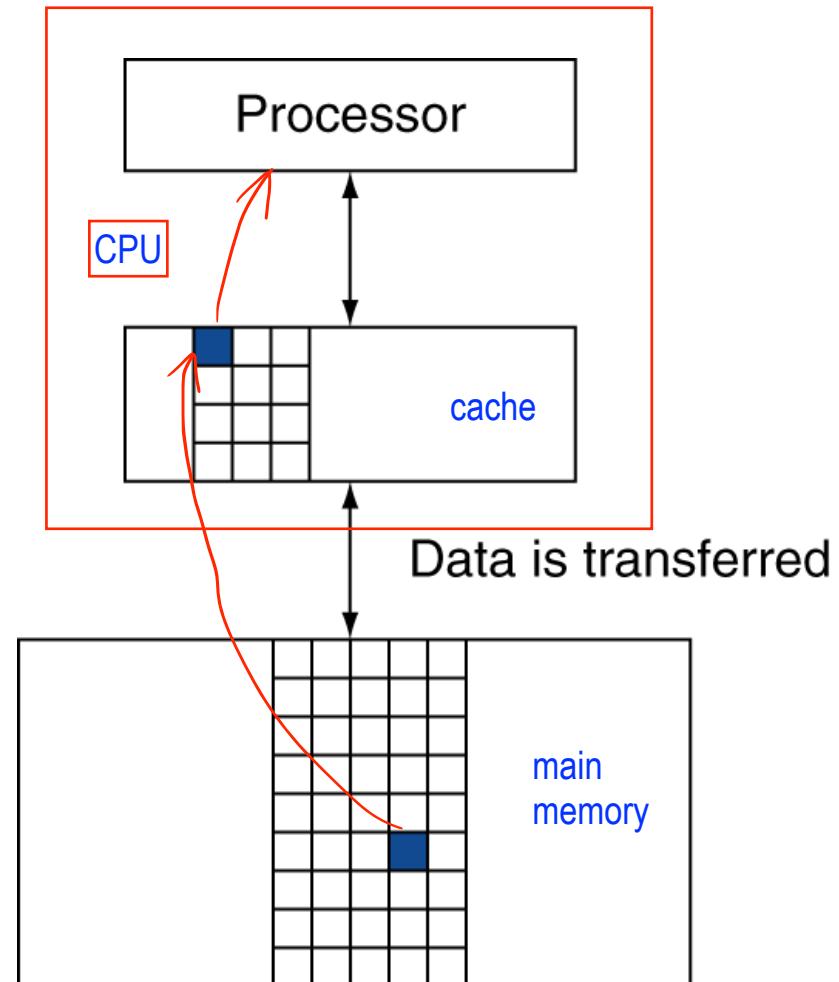
Hit: access satisfied by upper level

Hit ratio: hits/accesses

If accessed data is absent

Miss: block copied from lower level

Then accessed data supplied from upper level

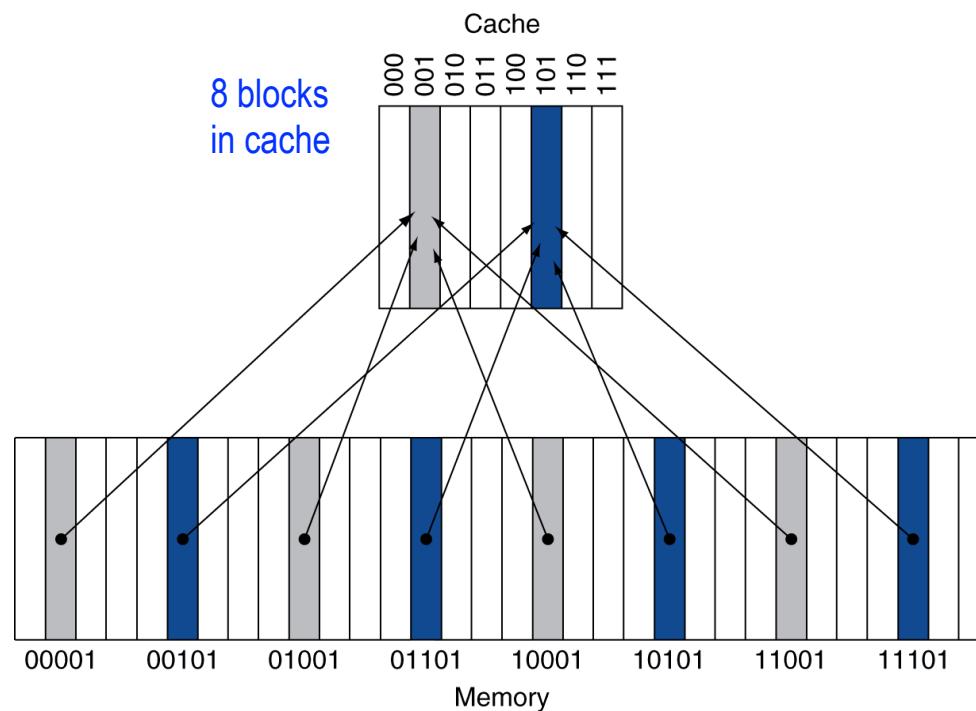


Memory Hierarchy Review

Cache position determined by address

Direct mapped: multiple memory cells share a certain cache line

Cache line position = (Block address) modulo (#Blocks in cache)



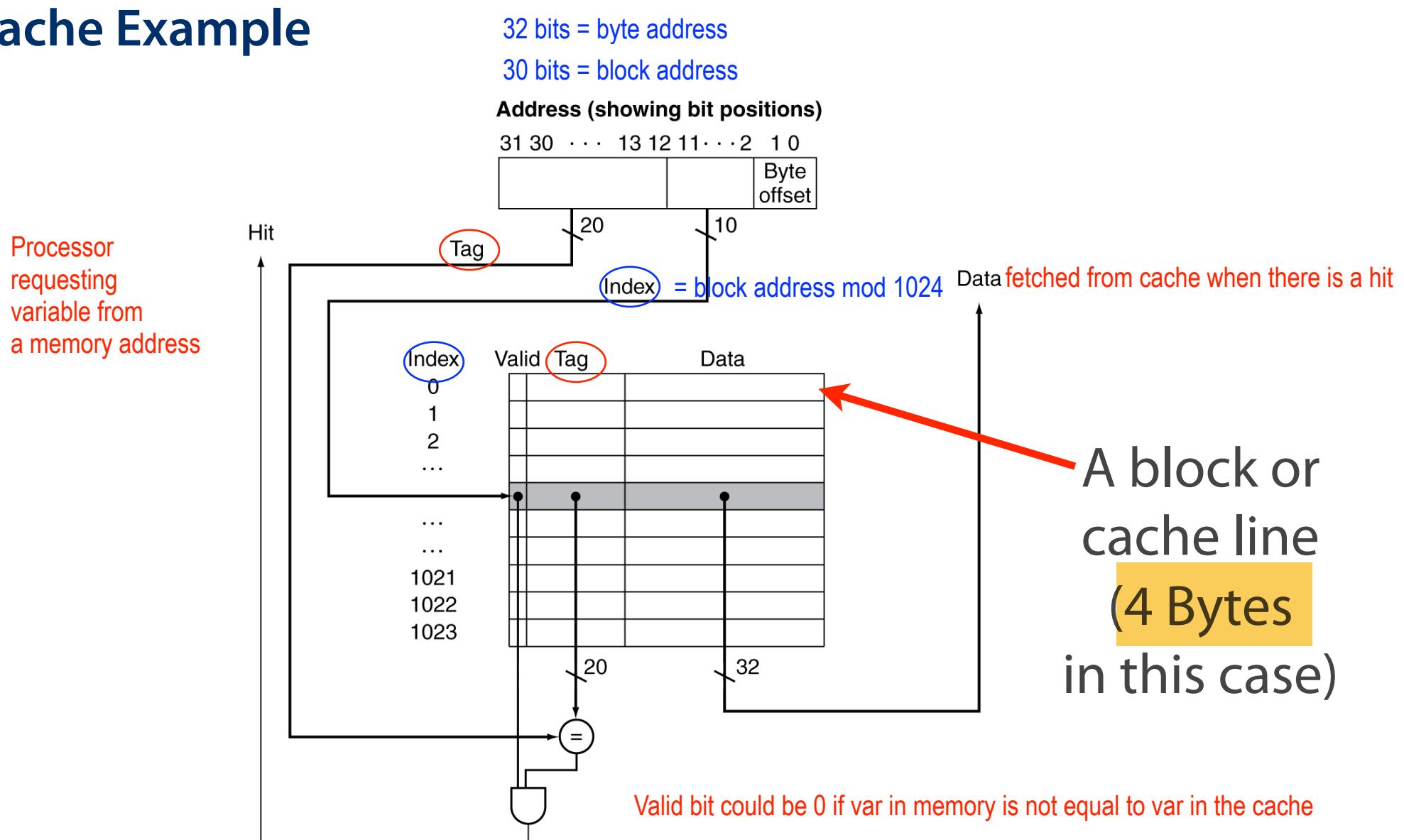
Memory Hierarchy Review

Cache Example

32 bits = byte address

30 bits = block address

Address (showing bit positions)



Cache Coherence

Cache Coherence in Shared Memory Systems

A certain level of consistency must be maintained for multiple copies of the same data

Required to ensure proper semantics and correct program execution

Serializability: there exists some serial order of instruction execution that corresponds to the parallel schedule

Cache coherence is enforced at the “cache-line level”

minimum unit to be moved around is block or cache line

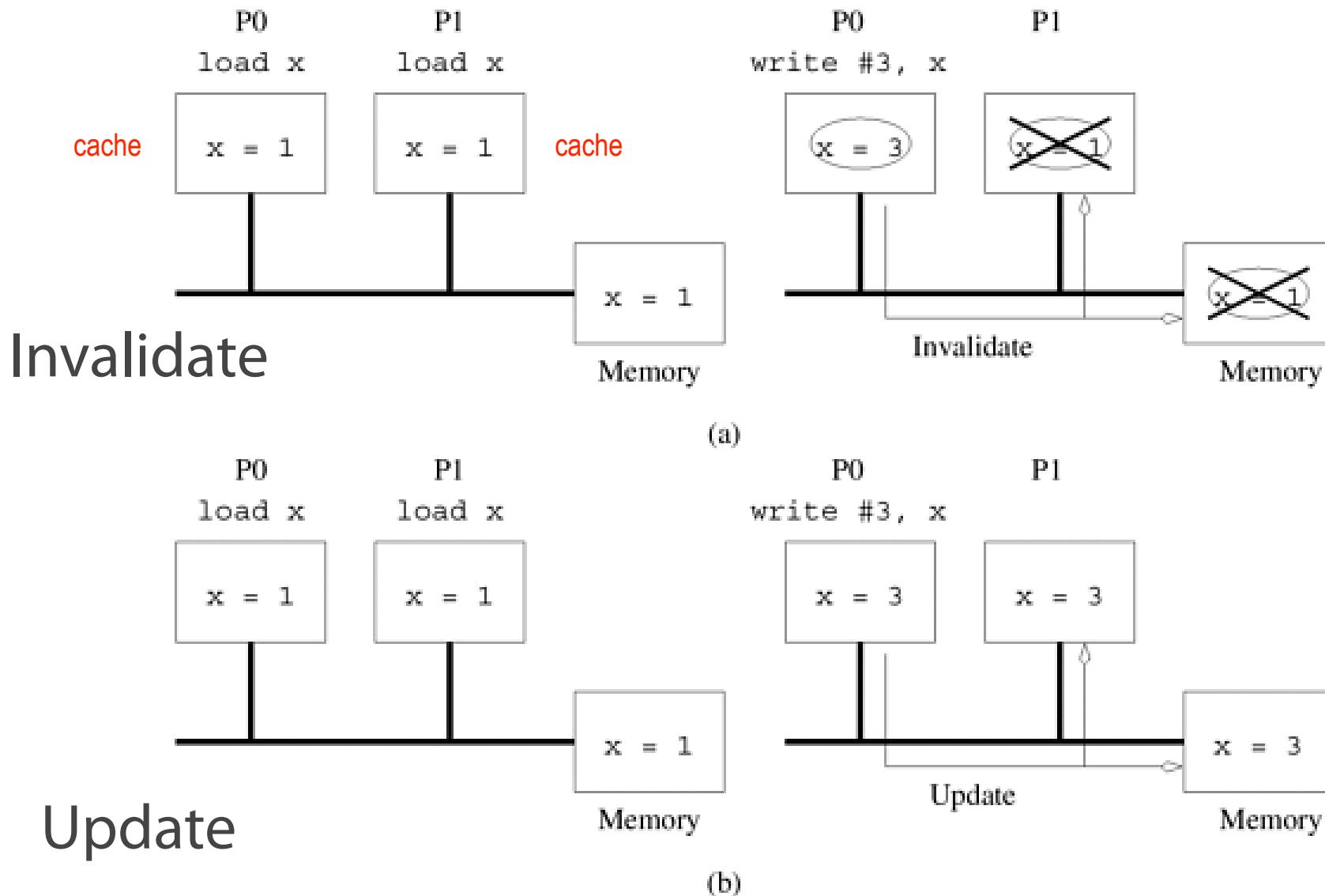
Two general protocols for dealing with it

Invalidate vs. Update

Classical trade-off between communication overhead (updates) and idling (stalling in invalidates)

Most existing schemes are based on the **invalidate protocol**

Cache Coherence Mechanisms



Cache Invalidating Protocol

Steps:

- Cache gets exclusive access to a block when it is to be written
- Broadcasts an invalidate message on the bus
- Subsequent read for the block from another cache misses
- Owning cache supplies updated value

CPU activity	Bus activity	CPU A's cache	CPU B's cache	Memory
				0
CPU A reads X	Cache miss for X	0	0	0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes 1 to X	Invalidate for X	1	invalidated	invalidated
CPU B read X	Cache miss for X	1	1	1

False Sharing

Suppose multiple threads with separate caches access different variables that belong to the same cache line.

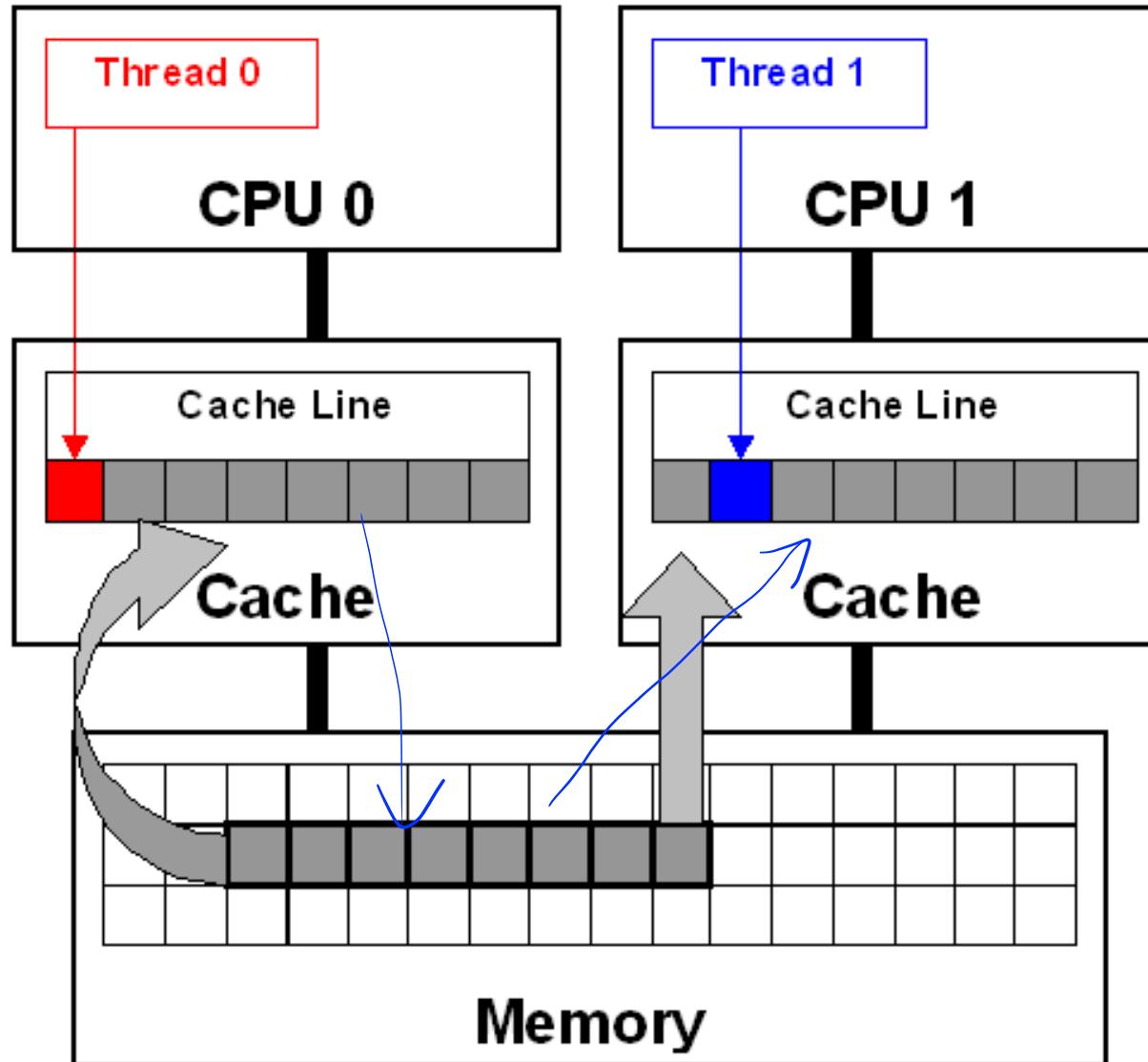
At least one of the threads updates its variable.

Even though neither thread has written to a variable that the other thread is accessing, the cache controller invalidates the *entire cache line* and forces the threads to get the values of the variables from main memory.

The behavior of the threads with respect to memory access is the same as if they were sharing a variable, hence the name *false sharing*.

False Sharing

Write
Invalidates
the entire
cache line



Read
has to
fetch from
main
memory

This miss is
caused by
False Sharing

False Sharing: Matrix-Vector Multiplication

```
1 void *Pth_mat_vect(void* rank) {  
2     long my_rank = (long) rank;  
3     int i, j;  
4     int local_m = m/thread_count;  
5     int my_first_row = my_rank*local_m;  
6     int my_last_row = (my_rank+1)*local_m - 1;  
7  
8     for (i = my_first_row; i <= my_last_row; i++) {  
9         y[i] = 0.0;  
10        for (j = 0; j < n; j++)  
11            y[i] += A[i][j]*x[j];  
12    }  
13  
14    return NULL;  
15 } /* Pth_mat_vect */
```

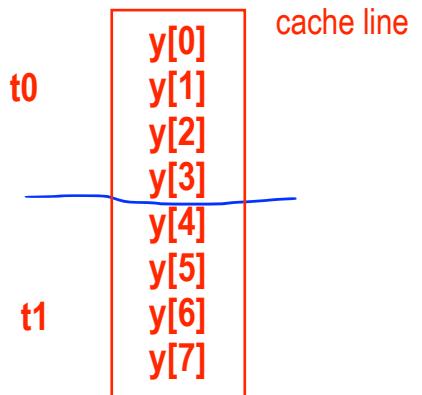


Table 4.5 Run-Times and Efficiencies of Matrix-Vector Multiplication (times are in seconds)

Threads	Matrix Dimension					
	8,000,000 × 8		8000 × 8000		8 × 8,000,000	
	Time	Eff.	Time	Eff.	Time	Eff.
1	0.393	1.000	0.345	1.000	0.441	1.000
2	0.217	0.906	0.188	0.918	0.300	0.735
4	0.139	0.707	0.115	0.750	0.388	0.290

False Sharing: Matrix-Vector Multiplication

Write-miss
for y: Line 9

Read-miss in x, A:
Line 11

Table 4.5 Run-Times and Efficiencies of Matrix-Vector Multiplication (times are in seconds)

Threads	Matrix Dimension					
	8,000,000 × 8		8000 × 8000		8 × 8,000,000	
	Time	Eff.	Time	Eff.	Time	Eff.
1	0.393	1.000	0.345	1.000	0.441	1.000
2	0.217	0.906	0.188	0.918	0.300	0.735
4	0.139	0.707	0.115	0.750	0.388	0.290

Unlikely to have false sharing. Why?

False Sharing
when updating
 $y[i]$: Line 11

Tips to Avoid False Sharing

1. Pad the y vector with dummy elements in order to ensure that any update by one thread won't affect another thread's cache line.
2. **Avoid frequent writing to global data that is accessed from multiple threads!**
Frequently used method. Reduce the updates to global data.
Have each thread use its own private storage during the multiplication loop, and then update the shared storage when they're done.
3. False sharing may be less visible when dealing with larger problem sizes, as there might be less sharing.
4. Other methods: align shared global data to cache line boundaries using compiler directives

When parallelizing an algorithm, partition data sets along cache lines, not across cache lines.

Tips to Avoid False Sharing

Padding a data structure to a cache line boundary and ensuring the array is also aligned using the compiler `__declspec(align(n))` statement (not required in this course), where n equals 64 (64 byte boundary).

```
struct ThreadParams
{
    // For the following 4 variables: 4*4 = 16 bytes
    unsigned long thread_id;
    unsigned long v; // Frequent read/write access variable
    unsigned long start;
    unsigned long end;

    // expand to 64 bytes to avoid false-sharing
    // (4 unsigned long variables + 12 padding)*4 = 64 Block size
    int padding[12];
};

Array[i] doesn't span two blocks. Each Array[i] occupies an entire block after padding and alignment.

__declspec(align(64)) struct ThreadParams Array[10];
```

Tips to Avoid False Sharing

Using local copies to avoid the frequency of writing and thus alleviate the impact of false sharing

```
struct ThreadParams
{
    // For the following 4 variables: 4*4 = 16 bytes
    unsigned long thread_id;
    unsigned long v; //Frequent read/write access variable
    unsigned long start;
    unsigned long end;
};

void threadFunc(void *parameter)
{
    ThreadParams *p = (ThreadParams*) parameter;
    // local copy for read/write access variable
    unsigned long local_v = p->v;

    for(int i = p->start; i < p->end; i++)
    {
        // update local_v multiple times
    }
    Only update the global variable once, thus the false sharing reduced to minimum
    p->v = local_v; // Update shared data structure only once
}
```

Thread Safety

Some C functions cache data between calls by declaring variables to be static, e.g., `strtok()`

This can cause errors when multiple threads call the function;

Since static storage is shared among the threads, one thread can overwrite another thread's data. Such a function is not thread-safe,

There are several such functions in the C library. Sometimes, however, there is a thread-safe variant.

`rand()` is not thread-safe, use `rand_r` (or `drand48_r` on Linux)

`strtok()` is not thread-safe, use `strtok_r()`

Thread Safety

```
#include <string.h>
#include <stdio.h>

int main()
{
    const char str[80] = "This is - ECE - 420";
    const char s[2] = "-";
    char *token;

    /* get the first token */
    token = strtok(str, s);

    /* walk through other tokens */
    while( token != NULL )
    {
        printf( " %s\n", token );

        token = strtok(NULL, s);
    }

    return(0);
}
```

=====

Output:

```
This is
ECE
420
```

Thread-Safety

See the multi-threaded thread-safety examples:

Pthreads: Tokenize

Pthreads: Tokenize (thread-safe)

Wrong programs may produce right results!