ECE 454
Computer Systems Programming

Threads and Synchronization

Ashvin Goel, Ding Yuan
ECE Dept, University of Toronto

Overall Progress of the Course

- What we have learnt so far: improving sequential performance
  - CPU architectures
  - Compiler optimizations
  - Optimizations for processor caches
  - Virtual memory, dynamic memory performance
- Next: improving performance by parallelization
  - Single machine parallelization
    - Using threads and processes on a single machine
  - Multi-machine parallelization
    - Using modern, data-intensive distributed computing

Contents

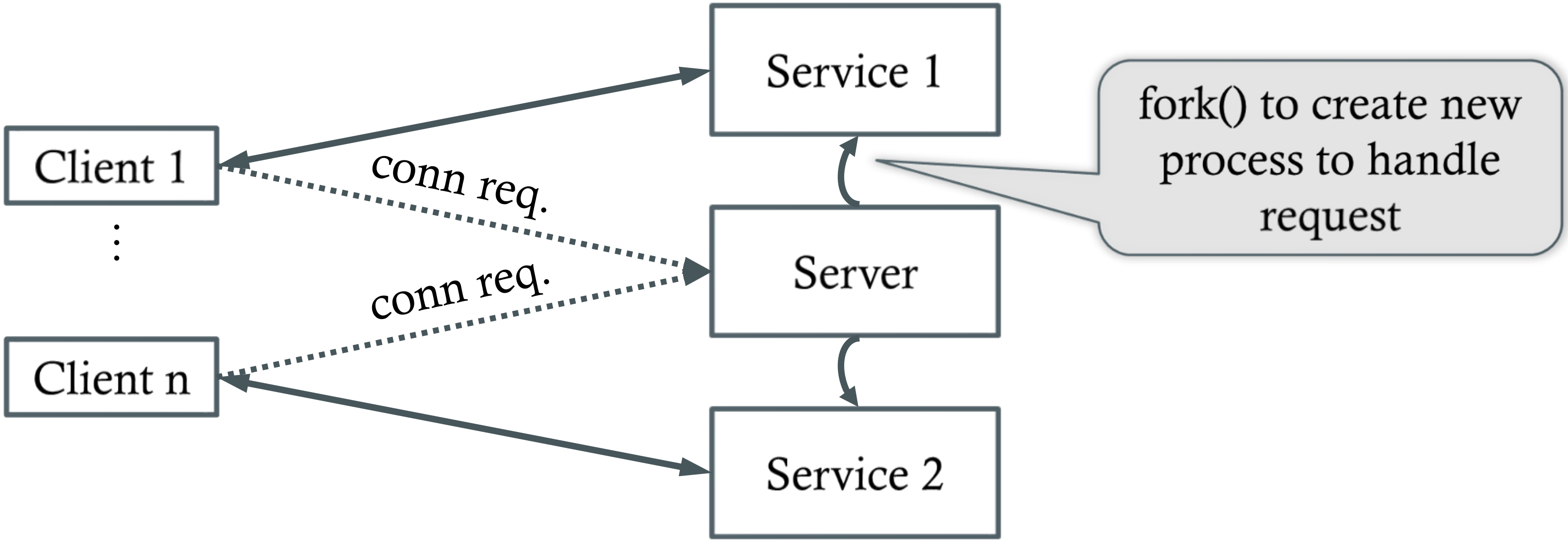
- Threads and processes
- Posix threads
- Mutual exclusion
- Synchronization

Threads and Processes

Parallel Processing with Processes

- E.g., web server or any other online service
- Using just one process is problematic
  - Client request arrives
  - Servicing request may require reading data from disk
    - Process blocks waiting for IO
  - Other requests cannot be serviced while blocked
- Idea: use multiple processes to service requests
  - Concurrently
    - When a process blocks, another process can run on the same core
  - Parallel
    - Multiple processes can run simultaneously on different cores

Sample Server



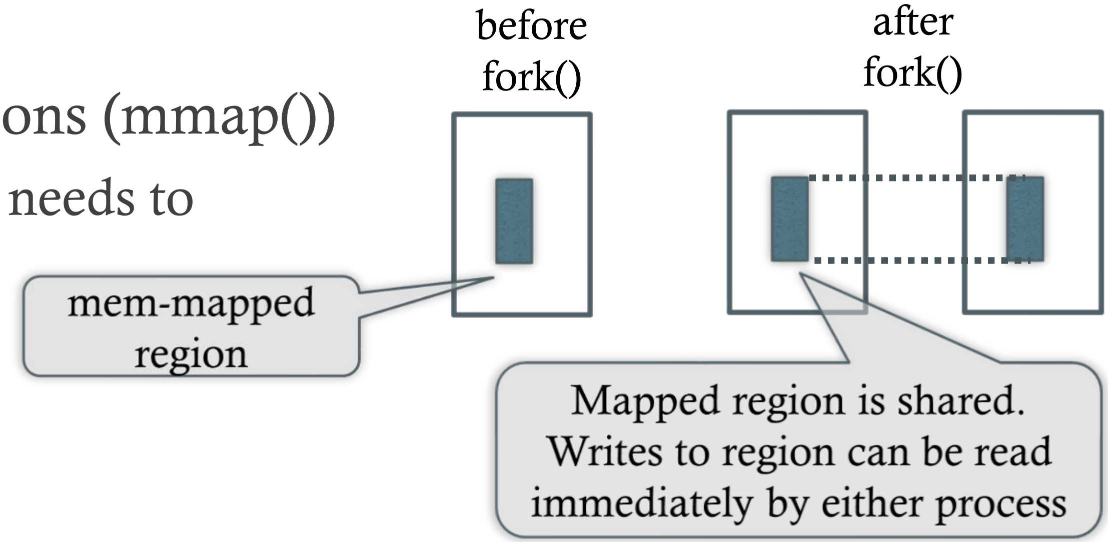
fork(): OS sys call to create new process. Address space of child is an exact copy of parent (same data, same code, same PC, except fork() of child returns 0; fork() of parent returns pid of child).

Sample Server

```
int main(int argc, char **argv) {
  int listenfd, connfd;
  stuct sock_addr_storage client_addr;

  listenfd = open_listenfd(argv[1]);
  while (1) {
    socklen_t  clientlen = sizeof(struct sock_addr_storage);
    connfd = Accept(listenfd, &client_addr, &clientlen);
    if (Fork() == 0) { // I am a child process
      close(listenfd);
      Read_and_Process_Request(connfd);
      close(connfd);
      exit();
    }
    close(connfd);
  }
}
```

Interprocess Communications (IPC)

- Above program requires no communication between processes
- But what if we do require communication?
  - Send signals
  - Sockets (TCP/IP connections)
  - Pipes
  - Memory mapped regions (mmap())
    - Access to shared data needs to be synchronized…
      - How?


before fork()
after fork()
mem-mapped region
Mapped region is shared. Writes to region can be read immediately by either process

Process

- Process = process context (CPU state) +
  address space (code, data, and stack) + kernel state

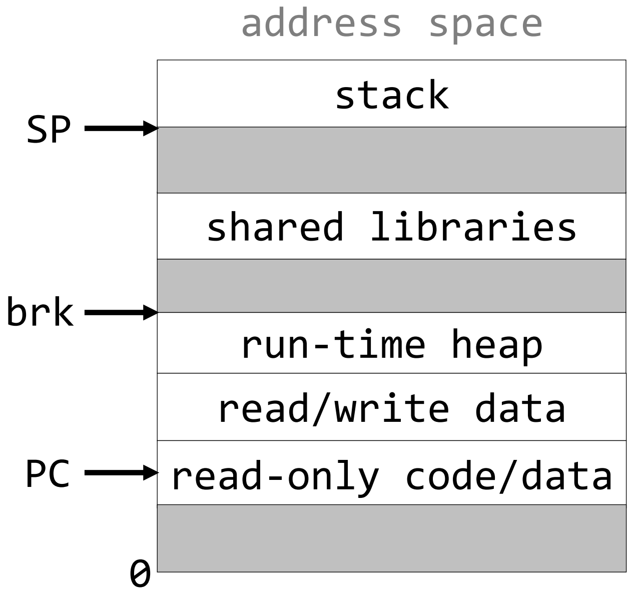
Process context

Data registers
Condition codes
Stack pointer (SP)
Program counter (PC)

Kernel state

virt. mem. structures
brk pointer
file descriptor table

address space



stack
SP
shared libraries
brk
run-time heap
read/write data
PC
read-only code/data
0

Performance Overhead in Process Management

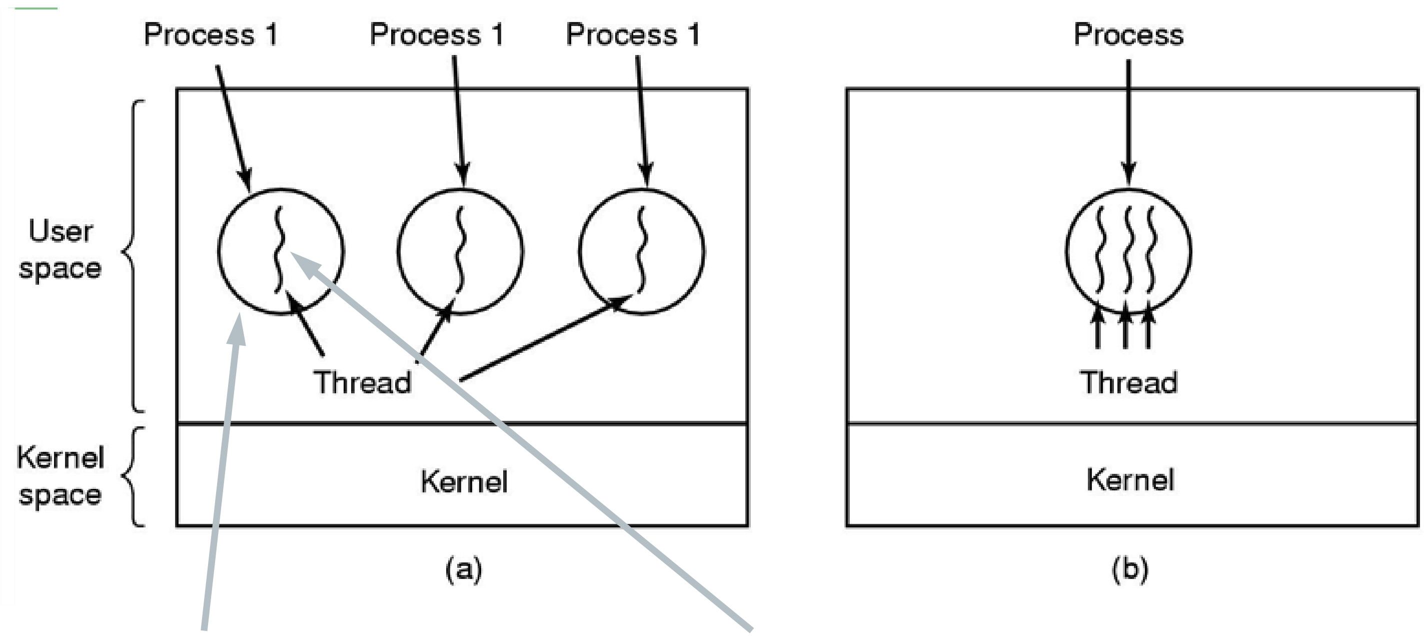
- Creating a new process is costly because of all the data structures (process context, address space, kernel state) that must be allocated and initialized

- Communicating between processes is costly because communication goes through the OS
  - Overhead of system calls and copying data
  - Switching between processes is also expensive
    - Why?

Rethinking Processes

- What do cooperating processes share?
  - They share some code and data (address space)
  - They share some privileges, files, sockets, etc. (kernel state)
- What don’t they share?
  - Each process has its own execution state: PC, SP, and registers
- Key idea: Why don’t we separate the concept of a process from its execution state?
  - Process: address space, kernel state
  - Execution state: PC, SP, registers
- Execution state also called thread of control, or thread

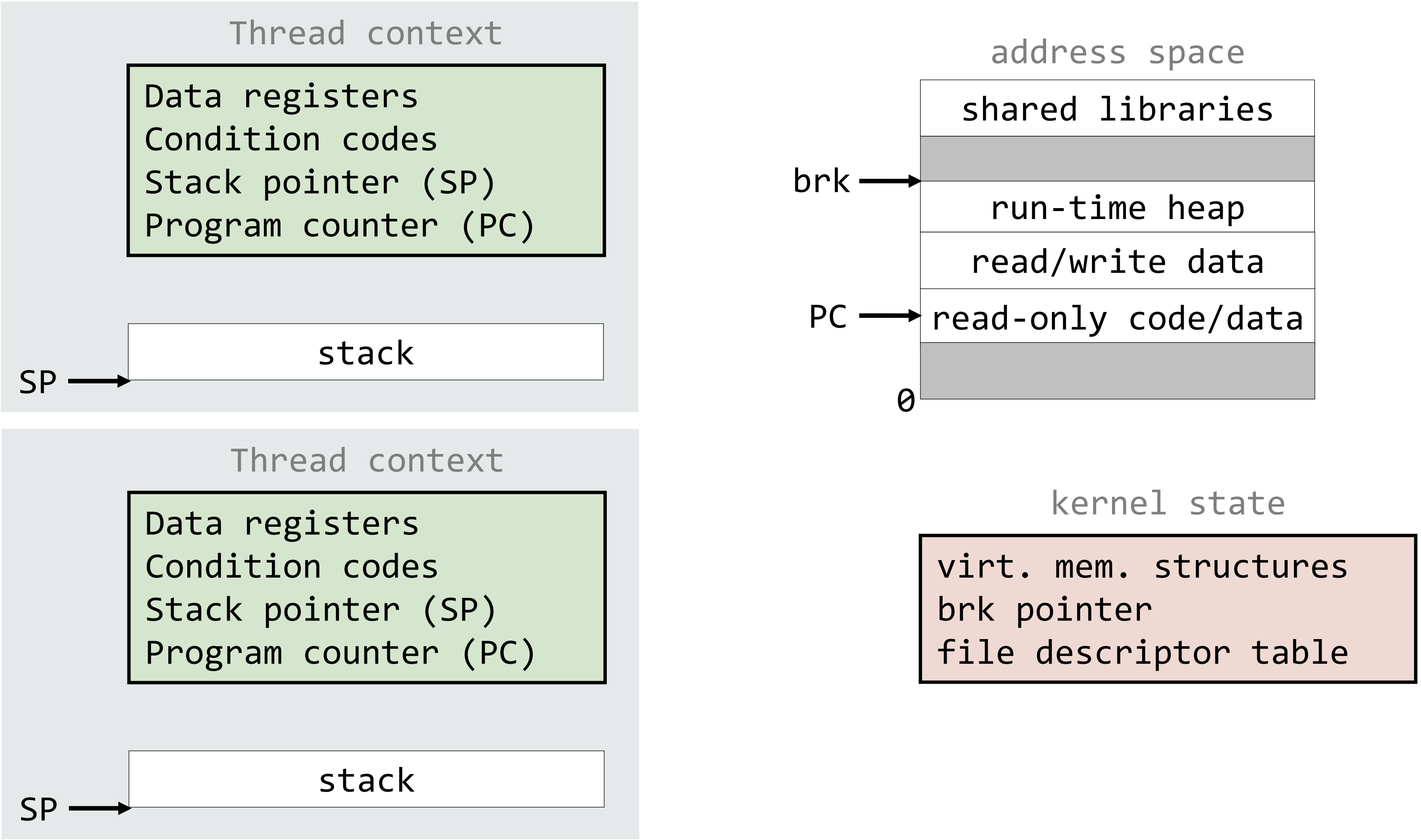
Threads: Lightweight Processes



environment (resource)   execution

(a) Three processes each with one thread
(b) One process with three threads

Process with Two Threads



Thread context

Data registers
Condition codes
Stack pointer (SP)
Program counter (PC)

stack

SP

Thread context

Data registers
Condition codes
Stack pointer (SP)
Program counter (PC)

stack

SP

address space

shared libraries

brk

run-time heap

read/write data

PC

read-only code/data

0

kernel state

virt. mem. structures
brk pointer
file descriptor table

Threads vs. Processes

- Similarities
  - Each has its own logical control flow
  - Each runs independently of (concurrently with) others
- Differences
  - Threads share code and data, processes (typically) do not
  - Threads are much less expensive than processes
    - Process control (creating and destroying processes) is more expensive than thread control
    - Process context switch is much more expensive than thread switch

Pros and Cons of Thread-Based Designs

- Pros
  - Easy to share data structures between threads
    - e.g., logging information, file cache
  - Threads are more efficient than processes
    - E.g., on Intel 2.6 GHz Xeon E5-2670:
      - Fork: 162 usecs
      - Thread creation: 18 usecs
- Cons
  - Unintentional sharing can cause subtle, hard-to-reproduce errors!
  - The ease with which data can be shared is both the greatest strength and the greatest weakness of threads

Posix Threads

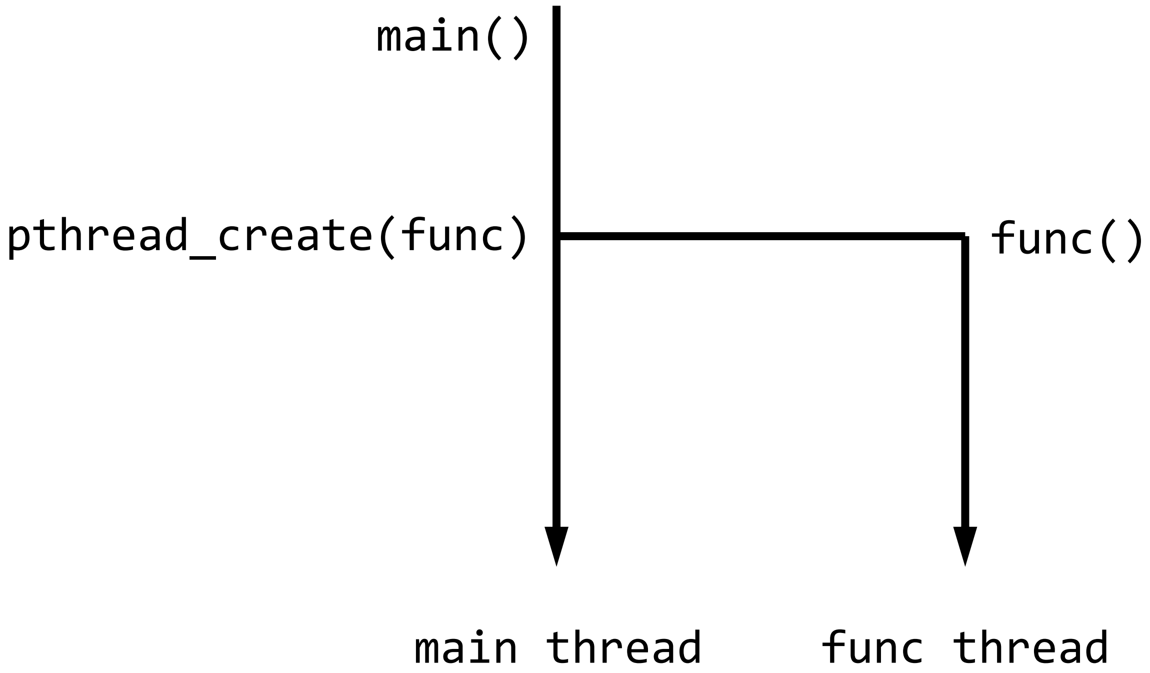
Posix Threads (Pthreads) Interface

- Pthreads: Standard interface of ~60 functions that manipulate threads from C programs
- Creating and reaping threads
  - `pthread_create(pthread_t *tid, …, func *f, void *arg)`
  - `pthread_join(pthread_t tid, void **thread_return)`
- Determining your thread ID
  - `pthread_self()`

Posix Threads (Pthreads) Interface

- Terminating threads
  - `pthread_cancel(pthread_t tid)`
  - `pthread_exit(void *thread_return)`
  - `return` (in primary thread routine terminates the thread)
  - `exit()` (terminates all threads)
- Synchronizing access to shared variables
  - Later

Example of Thread Creation



main()

pthread_create(func)

func()

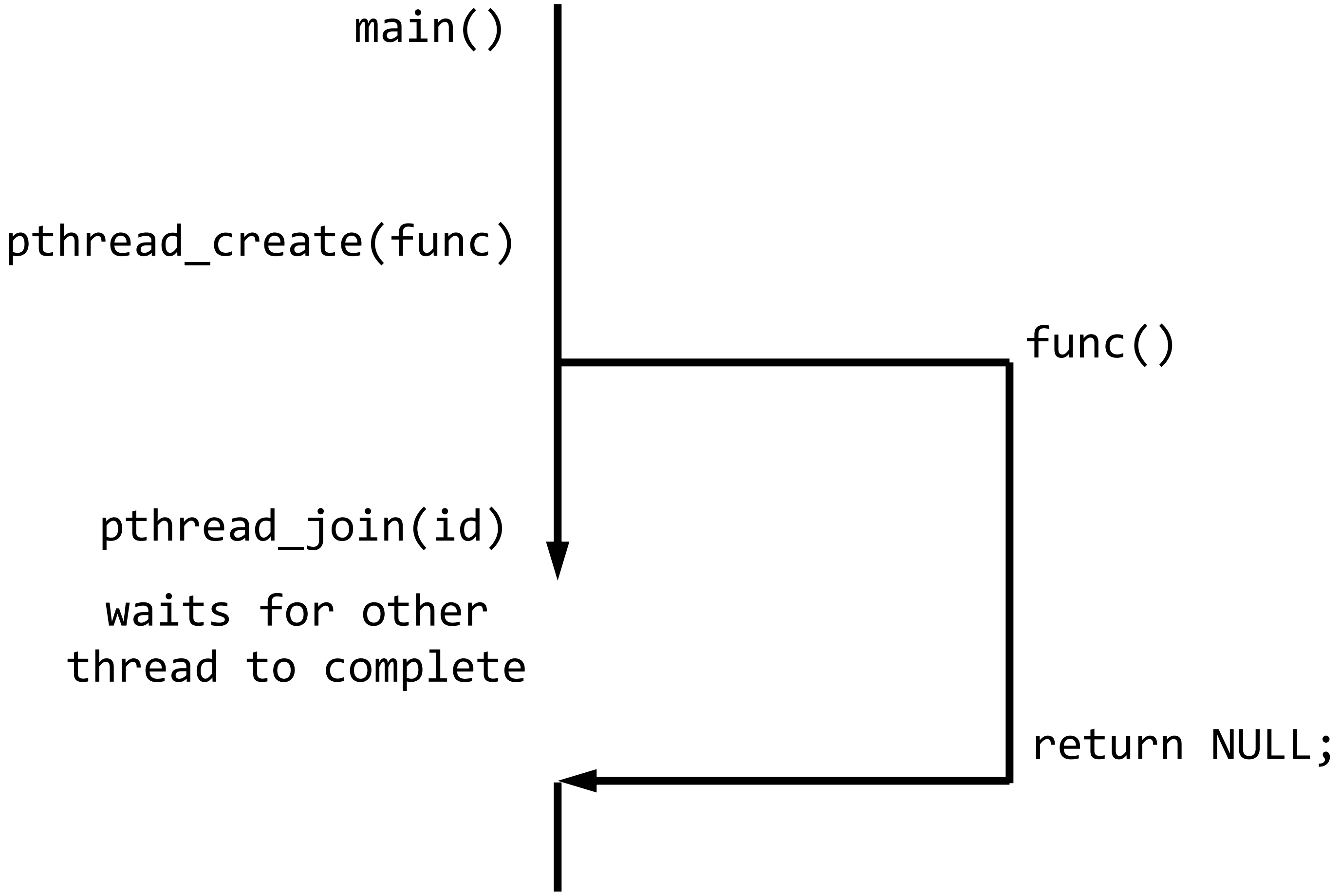
main thread

func thread

Thread Joining Example

```
void *func(void *) { ….. }
pthread_t id;
int X;
pthread_create(&id, NULL, func, (void *)&X);
…
pthread_join(id, NULL);   // awaits function to return
…
```

Example of Thread Creation (contd.)



main()

pthread_create(func)

func()

pthread_join(id)

waits for other
thread to complete

return NULL;

The Pthreads
“Hello, world" Program

```
/* hello.c - Pthreads "hello, world" program */
#include "csapp.h"

void *thread(void *vargp);

int main() {
  pthread_t tid;

  Pthread_create(&tid, NULL, &thread, NULL);
  Pthread_join(tid, NULL);
  exit(0);
}

/* thread routine */
void *thread(void *vargp) {
  printf("Hello, world!\n");
  return NULL;
}
```

Thread attributes
(usually NULL)

Thread arguments
(void *p)

assigns return value
(void **p)

How to Program with Pthreads

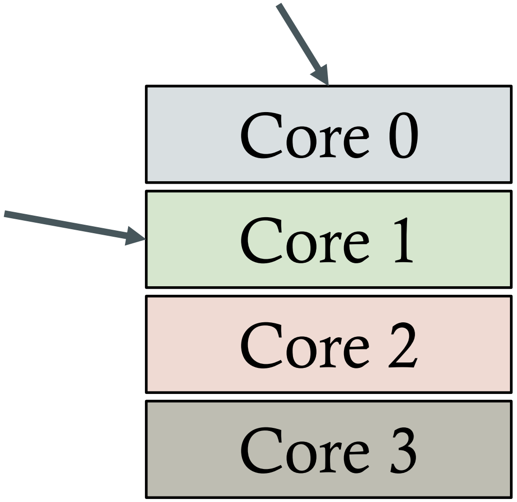
- Decide how to decompose the computation into parallel parts
- Create (and destroy) threads to support that decomposition
- Add synchronization to make sure dependences are satisfied
- Easier said than done!

Example: Matrix Multiply

```
for(i=0; i<n; i++) {
    for(j=0; j<n; j++) {
         for(k=0; k<n; k++) {
            c[i][j] += a[i][k]*b[k][j];
         }
    }
}
```

Core 0 calculates
c[0 … (n/4)-1][*]

Core 1 calculates
c[(n/4) … (2n/4)-1][*]



- All i- or j-iterations can be run in parallel
  - One option:
    - If we have p cores, assign n/p rows of C matrix to each core
    - Corresponds to partitioning the i-loop

Parallel Matrix Multiply

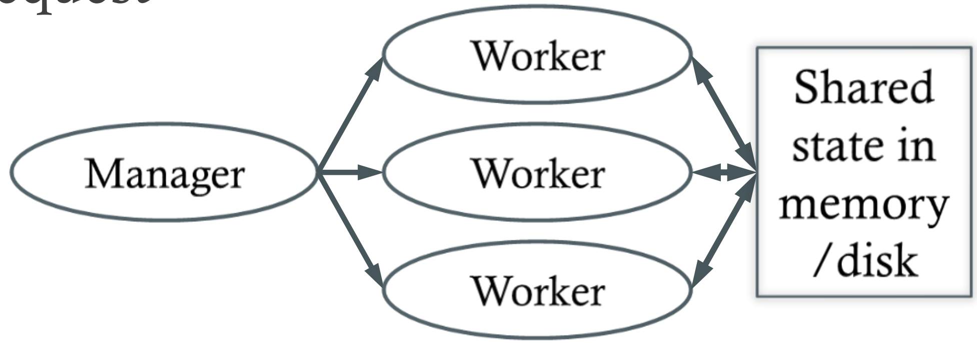
```
int n = 1000000000;
int p = 16; // # of cores

int main() {
  pthread_t thread[p];

  for(i=0; i<p; i++) {
    pthread_create(&thread[i], NULL, mmult, (void*) &i);
  }

  for(i=0; i<p; i++)
    pthread_join(thread[i], NULL);
  }
}
```

Matrix Multiply Per Thread

```
void mmult (void* s) {
  int slice = *((int *)s);

  int from = (slice*n)/p;
  int to = ((slice+1)*n)/p;

  for(i=from; i<to; i++) {
    for(j=0; j<n; j++) {
      for(k=0; k<n; k++)
        c[i][j] += a[i][k]*b[k][j];
    }
  }
}
```

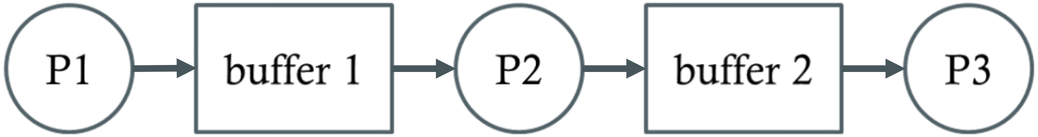
Some Models for Threaded Programs

- Manager / worker
  - Manager performs setup
  - Partitions work, e.g., by request
  - Synchronizes completion
  - Pros
    - Simple initial design
  - Cons
    - Shared state complicates design, synchronization
    - Easier to make workers stateless by separating state from workers, but then workers need to re-read up-to-date state when they operate on it
    - Hard to enforce job ordering



Some Models for Threaded Programs

- Pipelined execution
  - Execution of a request may be pipelined across multiple processes

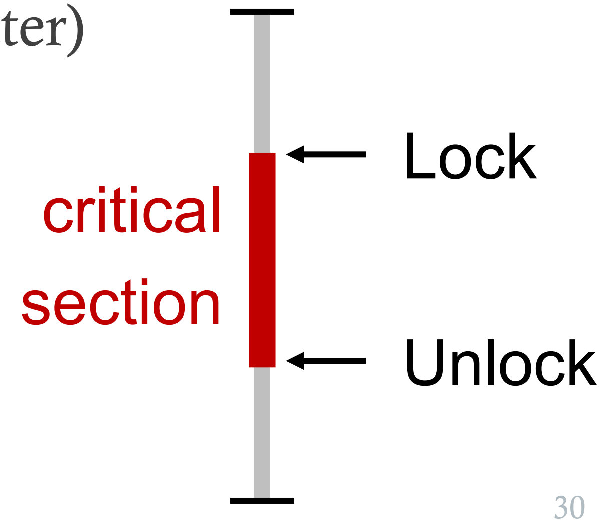


- Pros
  - Processes share no state, buffers perform synchronization
    - Each process is simple, written like a single threaded application
    - Each process can be stateful
  - Jobs can be ordered, logged
    - Simplifies tasks like backup, replication
- Cons
  - Code is written in callback style, making it hard to debug

Mutual Exclusion

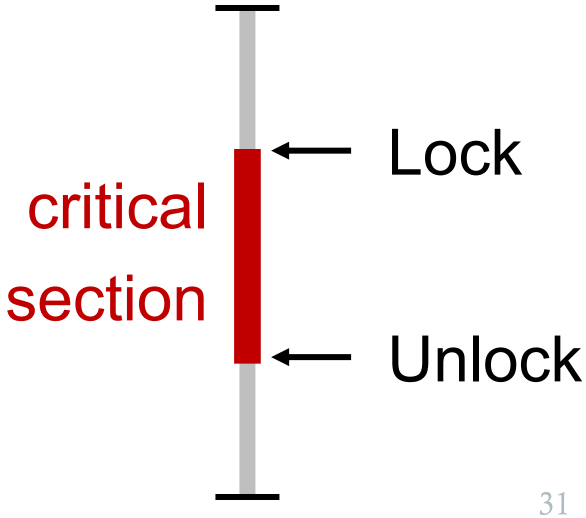
What is Mutual Exclusion?

- Ensures operations on shared data are performed by only one thread at a time, aka critical section
  - Ensured by using locks
  - Helps avoid races
- Does not provide any guarantees on ordering
  - Provided by synchronization (discussed later)

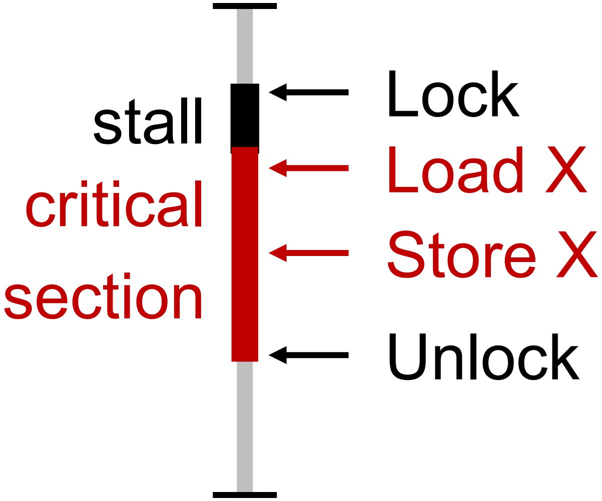


Mutual Exclusion Goals

- Works independent of speed of execution of threads
- Threads outside critical sections don't block other threads
- Threads trying to enter critical section succeed eventually
- Critical sections are small, allowing better scalability



Mutual Exclusion and Critical Sections



stall

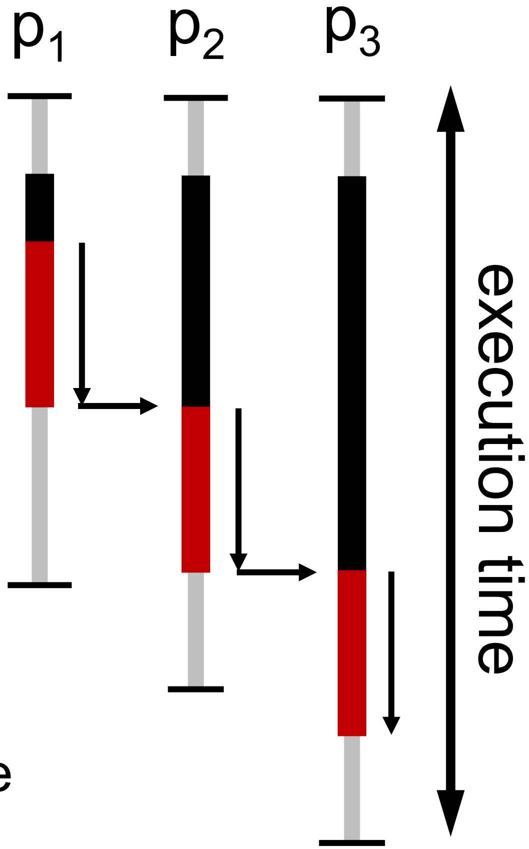
critical section

Lock

Load X

Store X

Unlock



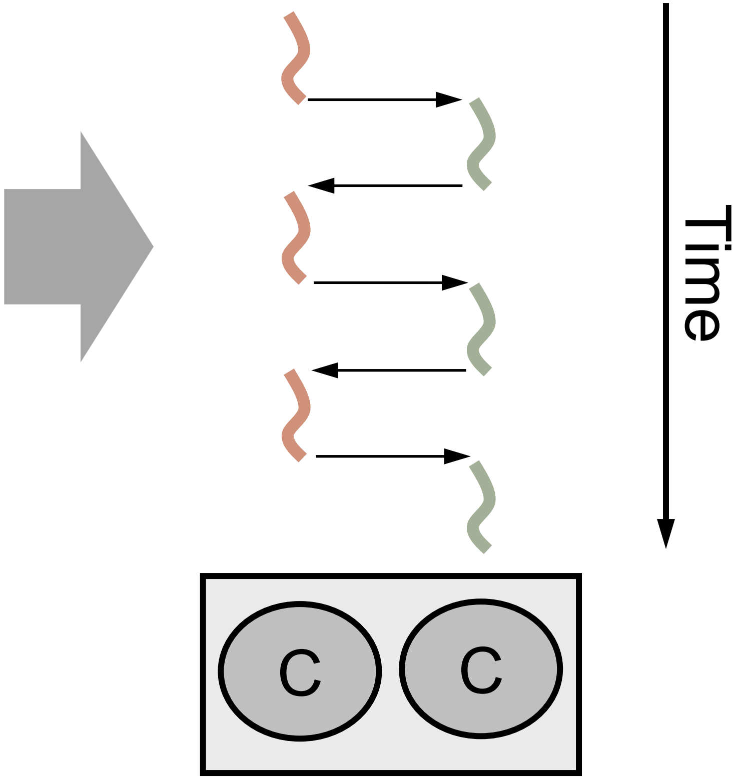
size of critical section impacts execution time

how can we reduce size of critical section?

Coarse-Grain Locking (a.k.a. Global Lock)

```
lock
… = A
…
A = …
unlock
```

```
lock
… = A
…
A = …
unlock
```

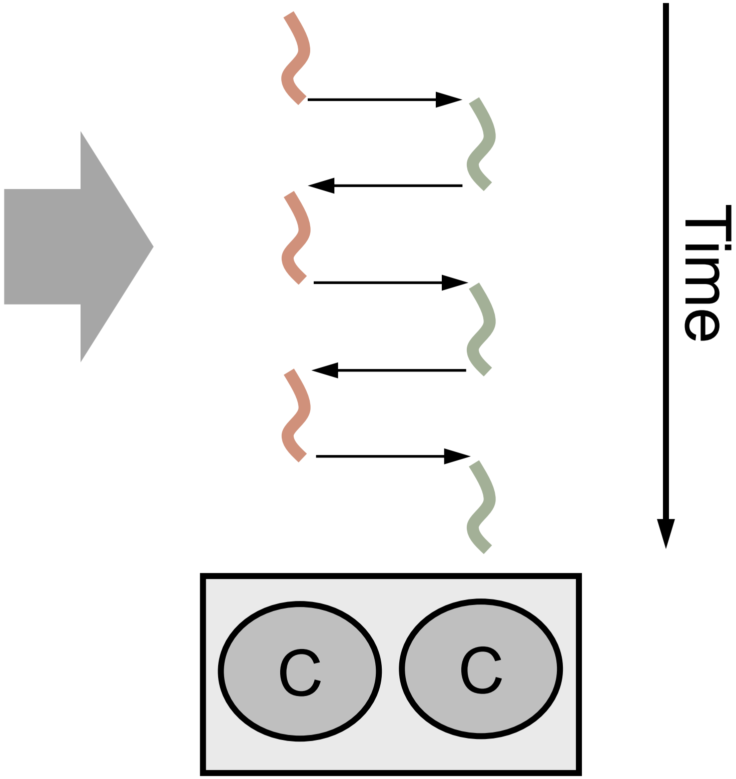


easy ☺

Coarse-Grain Locking (a.k.a. Global Lock)

```
lock
… = A
…
A = …
unlock
```

```
lock
… = B
…
B = …
unlock
```

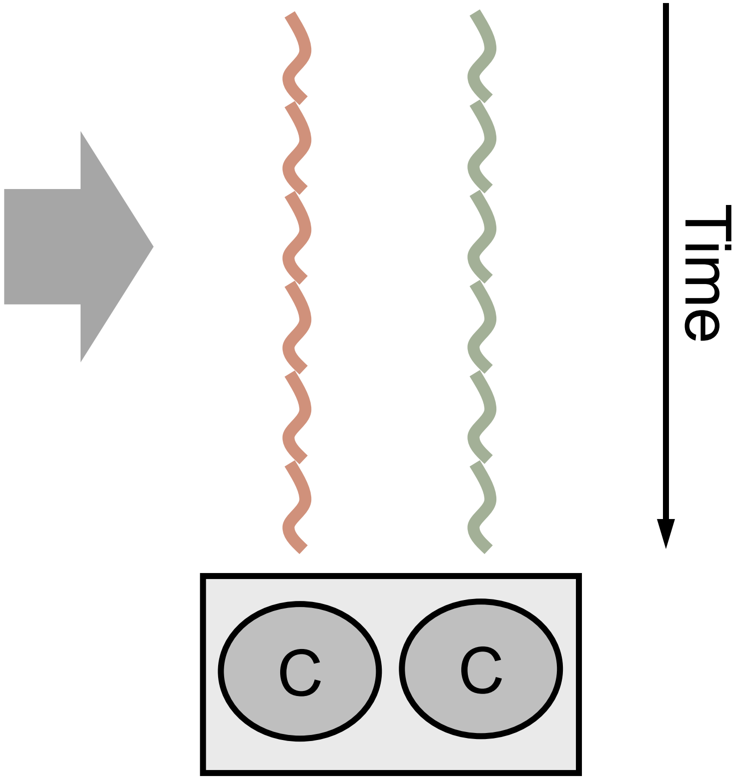


easy ☺ but slow ☹

Fine-Grained Locking

```
lock A
… = A

…

A = …
unlock A
```

```
lock B
… = B

…

B = …
unlock B
```



Fast ☺ but harder ☹

Contention and Scalability

- Contention refers to a lock that is held when another thread tries to acquire it
- Scalability refers to ability to handle increasing load with a larger system
- Locking serializes execution of critical sections
  - Limits ability to use multiple processors, recall Amdahl’s law
  - Locks that are frequently contended limit scalability
- Coarse-grained locking increases contention
  - Causes unnecessary cache misses from coherence protocol
  - May make performance even worse than single core

Example 1: Linux Kernel Scalability

- "An Analysis of Linux Scalability to Many Cores" [OSDI'10]
  - Linux 2.6.39 (2011) removed the last instance of big kernel lock

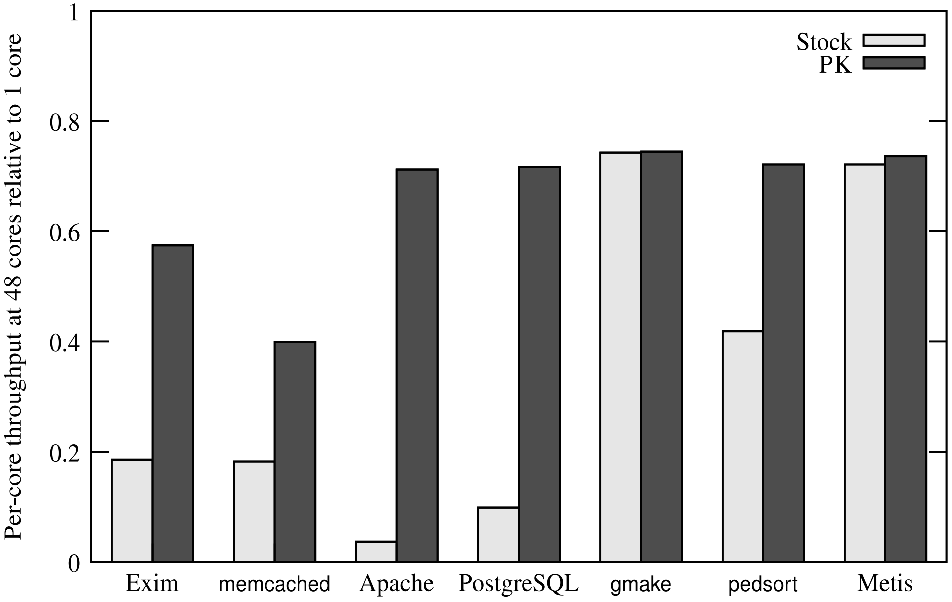
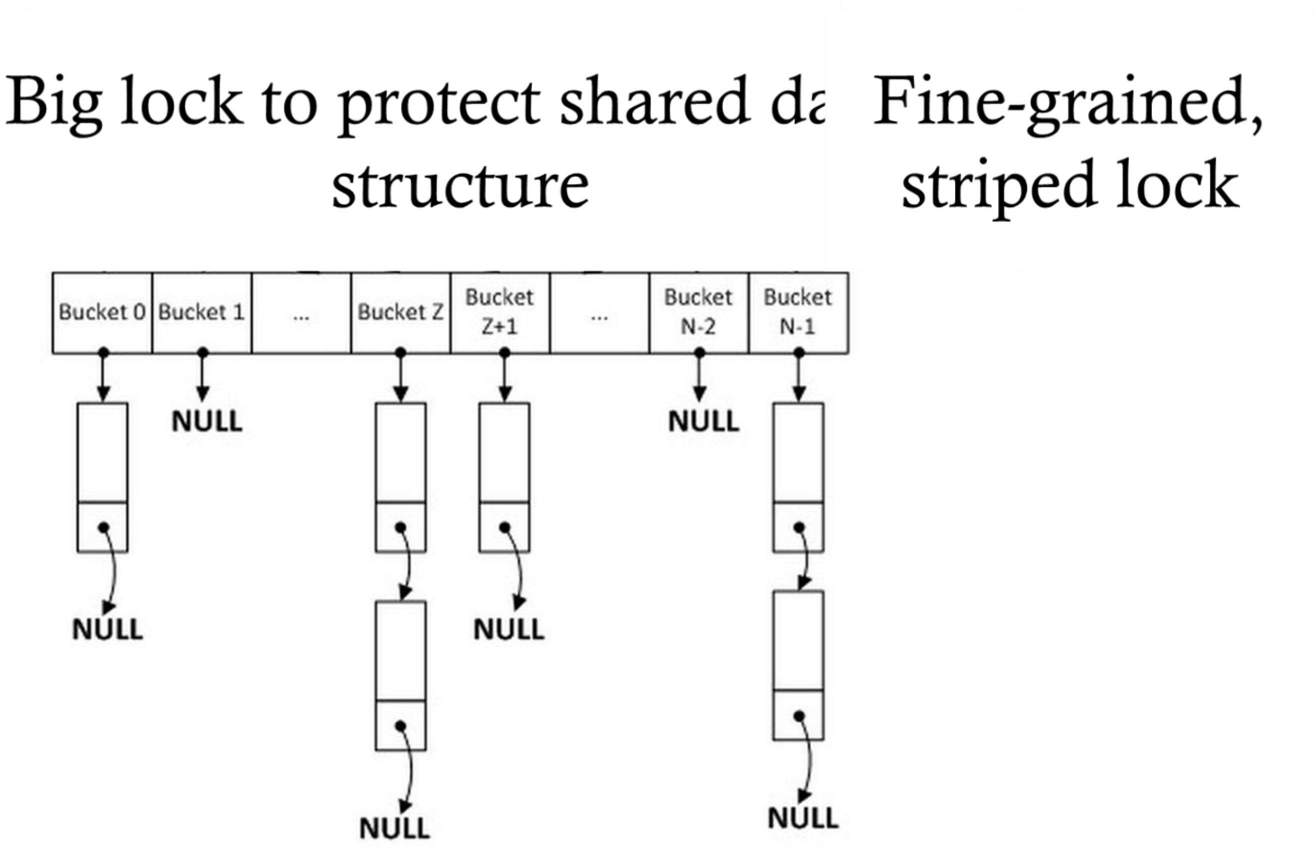


Figure 3: MOSBENCH results summary. Each bar shows the ratio of per-core throughput with 48 cores to throughput on one core, with 1.0 indicating perfect scalability. Each pair of bars corresponds to one application before and after our kernel and application modifications.

Example 2: Facebook’s memcached



Big lock to protect shared da structure

Fine-grained, striped lock

“Scaling Memcache at Facebook” [NSDI’13]
“Enhancing the Scalability of Memcached” [Intel@ Developer Zone’12]

Races

- A race occurs when certain thread interleavings lead to incorrect program behavior
  - E.g., program depends on reading and writing a variable with no interleaving accesses to the variable, but the program doesn't enforce this behavior
  - E.g., program depends on one thread reaching point x before another thread reaches point y, but the program doesn't enforce this behavior
- A data race occurs when a variable is accessed (read or modified) by multiple threads without any synchronization, and at least one thread modifies the variable

Data Race Example

```
/* a threaded program with a data race */
int main() {
    pthread_t tid[N];
    int i;
    for (i = 0; i < N; i++)
        pthread_create(&tid[i], NULL, thread, &i);
    for (i = 0; i < N; i++)
        pthread_join(tid[i], NULL);
    exit(0);
}

/* thread routine */
void *thread(void *vargp) {
    int myid = *((int *)vargp);
    printf("Hello from thread %d\n", myid);
}
```

Output: Why?

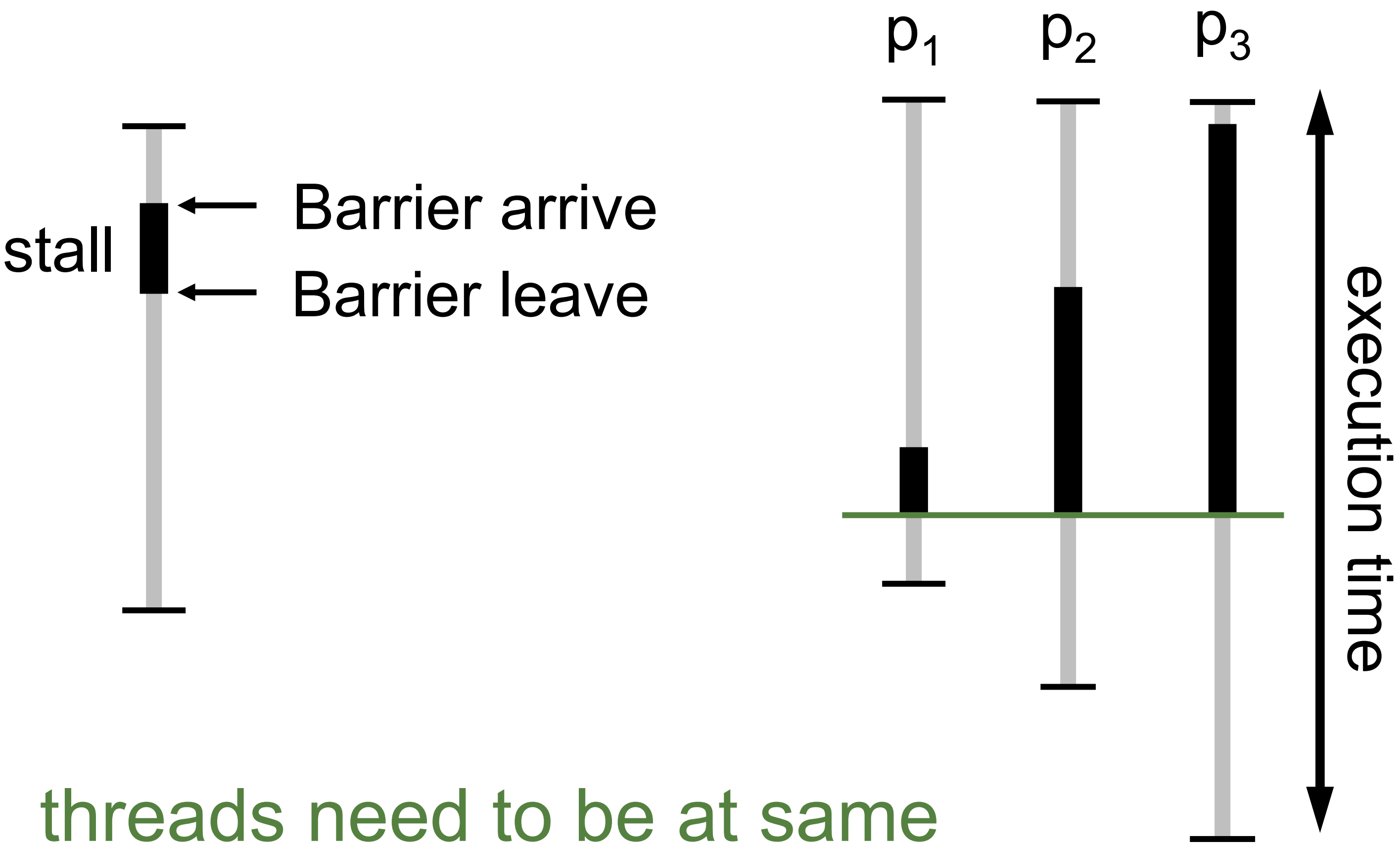
```
$ ./race
Hello from thread 1
Hello from thread 2
Hello from thread 6
Hello from thread 3
Hello from thread 7
Hello from thread 5
Hello from thread 9
Hello from thread 4
Hello from thread 9
Hello from thread 0
```

Synchronization

What is Synchronization?

- Ensures ordering of events to preserve dependencies
  - Barriers: All threads wait at same point in the program
    - E.g., doing pthread_join for all threads
  - Producer-consumer
    - E.g., Thread A writes a stream of samples, Thread B reads them
  - Condition variables used to wait for an event to occur

Barriers



threads need to be at same program point before continuing

Barriers with Pthreads

- Initialize a barrier with count threads

```
pthread_barrier_t *barrier;
pthread_barrier_init(pthread_barrier_t *barrier,
                     pthread_barrier_attr_t *attr,
                     unsigned int count)
```

- Wait for all threads to reach the barrier before continuing

```
pthread_barrier_wait(pthread_barrier_t *barrier)
```

Condition Variables with Pthreads

- Requires using locks while condition variables are used
- Create a new condition variable

```
pthread_cont_t *cond;
pthread_cond_init(pthread_cond_t *cond,
                  pthread_cond_attr *attr)
```

- Destroy a condition variable

```
pthread_cond_destroy(pthread_cond_t *cond)
```

Condition Variable Synchronization

- Block the calling thread on condition variable cond

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

  - Atomically unlocks the mutex, waits on cond
  - When cond is signaled, reacquires mutex

- Unblock one thread waiting on cond

```
pthread_cond_signal(pthread_cond_t *cond)
```

  - If no thread is waiting, signal does nothing

- Unblock all threads waiting on cond

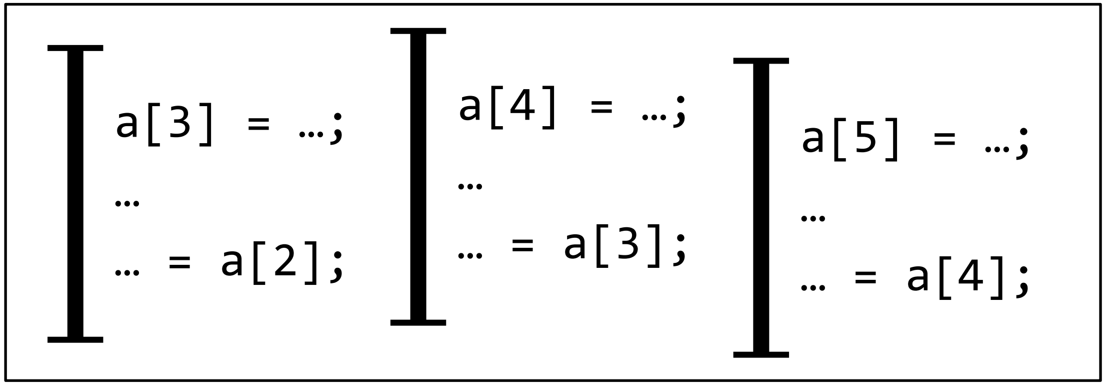
```
pthread_cond_broadcast(pthread_cond_t *cond)
```

  - If no thread waiting, then broadcast does nothing.

Example: Parallelize This Code

```
for(i=1; i<100; i++) {
  a[i] = …;
  …; // long code
  … = a[i-1];
}
```

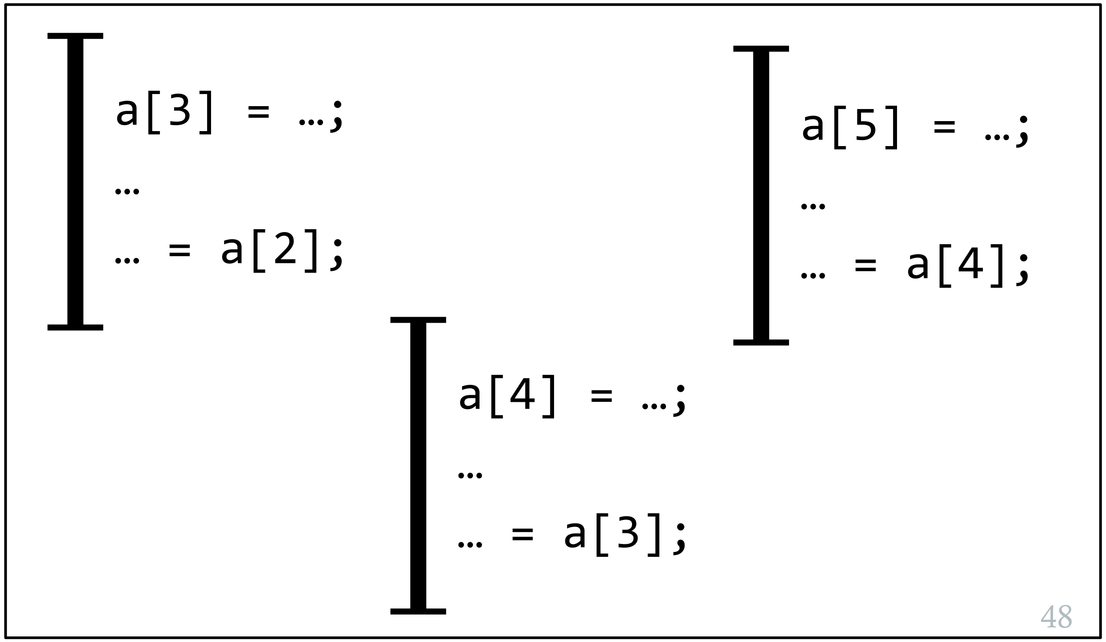
- Problem:
  - Each iteration depends on the previous iteration



```
a[3] = …;
…
… = a[2];
```

```
a[4] = …;
…
… = a[3];
```

```
a[5] = …;
…
… = a[4];
```



```
a[3] = …;
…
… = a[2];
```

```
a[5] = …;
…
… = a[4];
```

```
a[4] = …;
…
… = a[3];
```

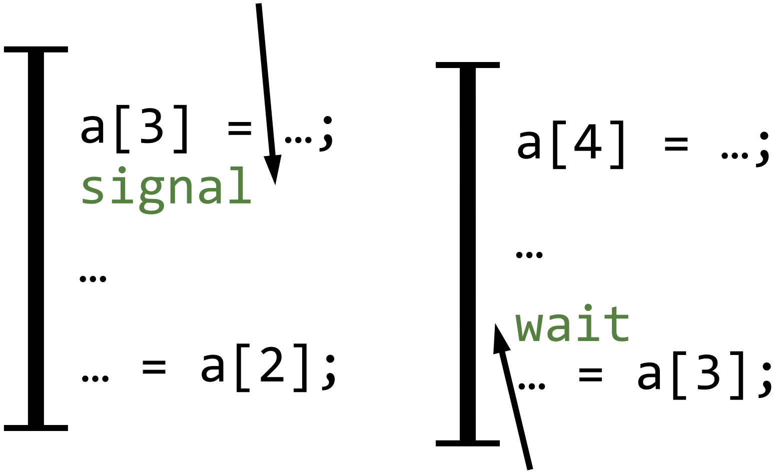
Synchronization with Condition Variable

```
for( i=...; i<...; i++ ) {
        a[i] = …;
        signal(e_a[i]);
        …; // long code
        wait(e_a[i-1]);
        … = a[i-1];
}
```

need a cond variable for each iteration

- PROBLEM
  - signal does nothing if corresponding wait hasn't already executed
    - i.e., signal gets "lost"

signal is lost if wait hasn't executed yet



```
a[3] = …;
signal
…
… = a[2];
```

```
a[4] = …;
…
wait
… = a[3];
```

wait waits forever -- deadlock!

How to Remember a Signal

```
signal(i) {

  arrived[i] = 1;   // track that signal(i) has happened
  pthread_cond_signal(&cond[i]); //signal

}

wait(i) {

  if (arrived[i] == 0) // wait only if signal hasn’t happen yet
    pthreads_cond_wait(&cond[i], …);
  arrived[i] = 0;  // reset for next time

}
```

How to Remember a Signal

```
signal(i) {
  pthread_mutex_lock(&mutex_rem[i]);
  arrived[i] = 1;   // track that signal(i) has happened
  pthread_cond_signal(&cond[i]); //signal
  pthread_mutex_unlock(&mutex_rem[i]);
}

wait(i) {
  pthreads_mutex_lock(&mutex_rem[i]);
  if (arrived[i] == 0) // wait only if signal hasn’t happen yet
    pthreads_cond_wait(&cond[i], mutex_rem[i]);
  arrived[i] = 0;  // reset for next time
  pthreads_mutex_unlock(&mutex_rem[i]);
}
```

Synchronization with Semaphores

- Semaphores store state and so wait() and signal() can happen in either order

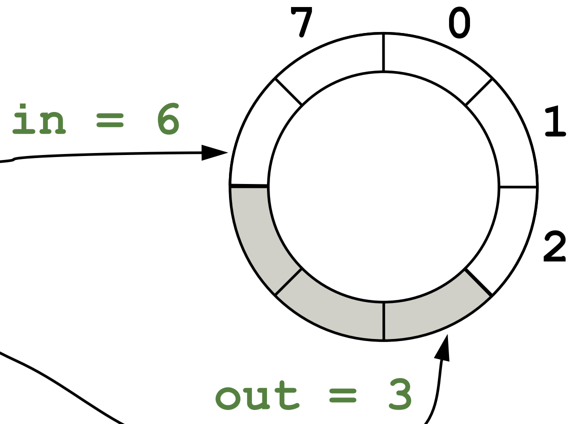
```
for( i=...; i<...; i++ ) {
  a[i] = …;
  up(e_a[i]);     // similar to signal, increments semaphore
  …; // long code
  down(e_a[i-1]); // similar to wait, decrements semaphore
  … = a[i-1];
}
```

Producer-Consumer Problem

- Threads communicate with each other using a shared buffer of a fixed size (i.e., bounded buffer)
  - One or more producers fill buffer
  - One or more consumers empty buffer
- Two synchronizations conditions
  - Producers wait if the buffer is full
  - Consumers wait if the buffer is empty

Bounded Buffer Implementation

```
shared variables:
char buf[8]; // 7 usable slots
int in;      // place to write
int out;     // place to read
```



- Implementation uses a circular buffer
  - Producers write at in, increment in, go clockwise
  - Consumers read from out, increment out, go clockwise
  - Number of elements in buffer: count = (in - out + n) % n
    - E.g., count = (6 – 3 + 8) % 8 = 3 // n is 8 since buffer has 8 slots
  - Buffer is full when it has n-1 elements, i.e., count == (n - 1)
  - Buffer is empty when it has no elements, i.e., count == 0

Producer-Consumer with Monitors

```
Global variables:
buf[n], in, out;
lock l = 0;
cv full; // no initialization
cv empty;
```

- Why two condition variables?
- Why use “while”, instead of “if”?

```
void send(char msg) {
  lock(l);
  while ((in-out+n)%n == n - 1) {
    wait(full, l);
  } // full
  buf[in] = msg;
  in = (in + 1) % n;
  signal(empty, l);
  unlock(l);
}
```

```
char receive() {
  lock(l);
  while (in == out) {
    wait(empty, l);
  } // empty
  msg = buf[out];
  out = (out + 1) % n;
  signal(full, l);
  unlock(l);
  return msg;
}
```

Producer-Consumer with Semaphores

```
Global variables:
buf[n], in, out;
lock l;
sem full = 0;  // no full slots
sem empty = n; // all slots are empty
```

- Why two condition variables?
- Why is locking needed?
- Can we switch down(), lock()?

```
void send(char msg) {
  down(empty);
  lock(l);
  buf[in] = msg;
  in = (in + 1) % n;
  unlock(l);
  up(full);
}
```

```
char receive() {
  down(full);
  lock(l);
  msg = buf[out];
  out = (out + 1) % n;
  unlock(l);
  up(empty);
  return msg;
}
```