Shared-Memory Model
Thread Programming

Objectives

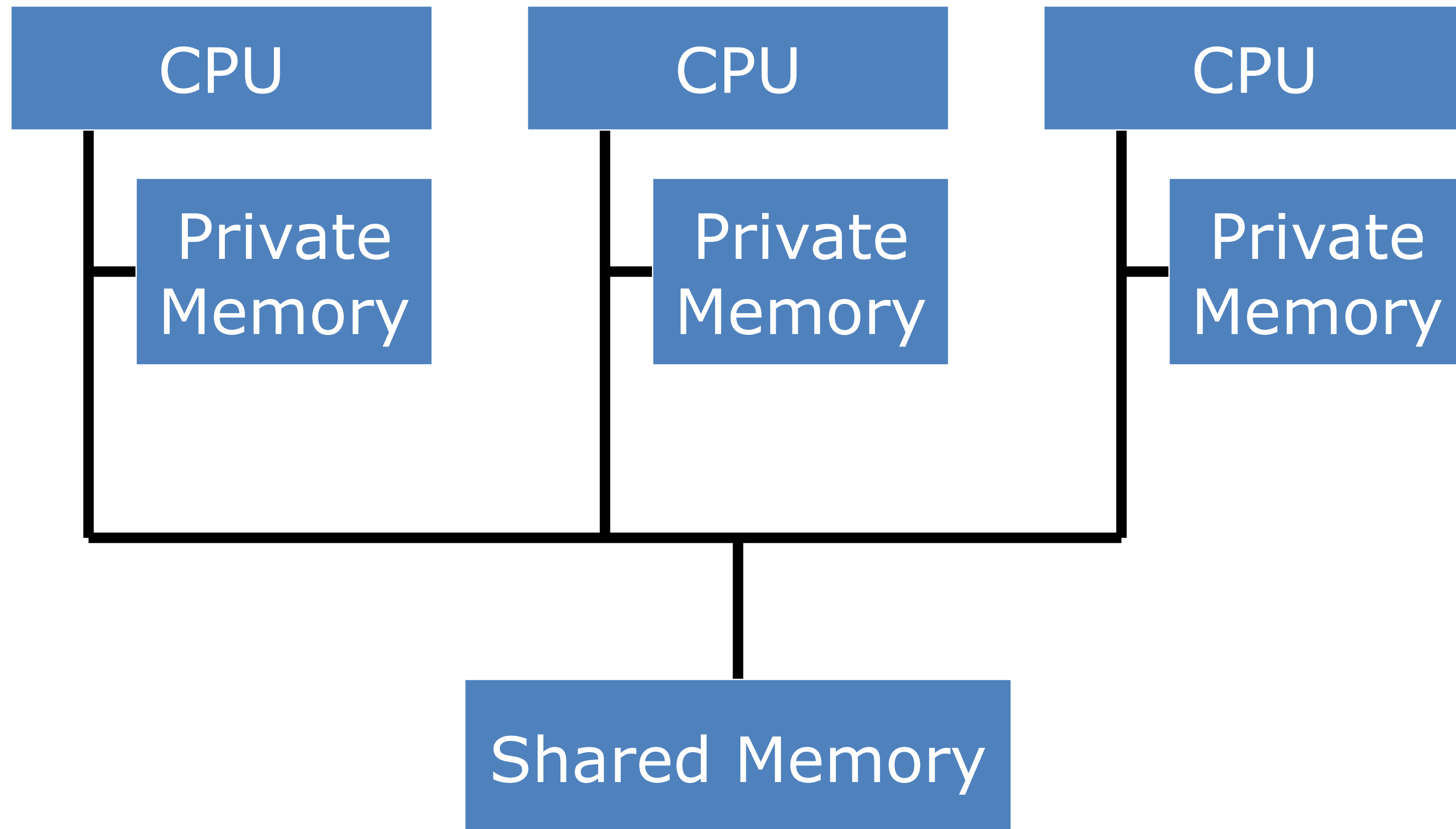
At the end of this module you should be able to:

- ✓ Describe the shared-memory model of parallel programming
- ✓ Describe the differences between the fork/join model and the general threads model
- ✓ Demonstrate how to implement domain and functional decompositions using threads
- ✓ Decide whether a variable in a multithreaded program should be shared or private

Cooperating Parallel Processes

- Parallel computing $\Rightarrow$ multiple processes working together to speed solution of a task
- Working together $\Rightarrow$ process cooperation
- Kinds of cooperation
  - ✓ Sharing information (communication)
  - ✓ Keeping out of each other's way (synchronization)

The Shared-Memory Model



Evaluating Parallel Models

- How do processes share information?
- How do processes synchronize?
  - ✓ In shared-memory model, both accomplished through shared variables
  - ✓ Communication: buffer
  - ✓ Synchronization: semaphore

Methodology

- Study problem, sequential program, or code segment
- Look for opportunities for parallelism
- Use threads to express parallelism

What Is a Process?

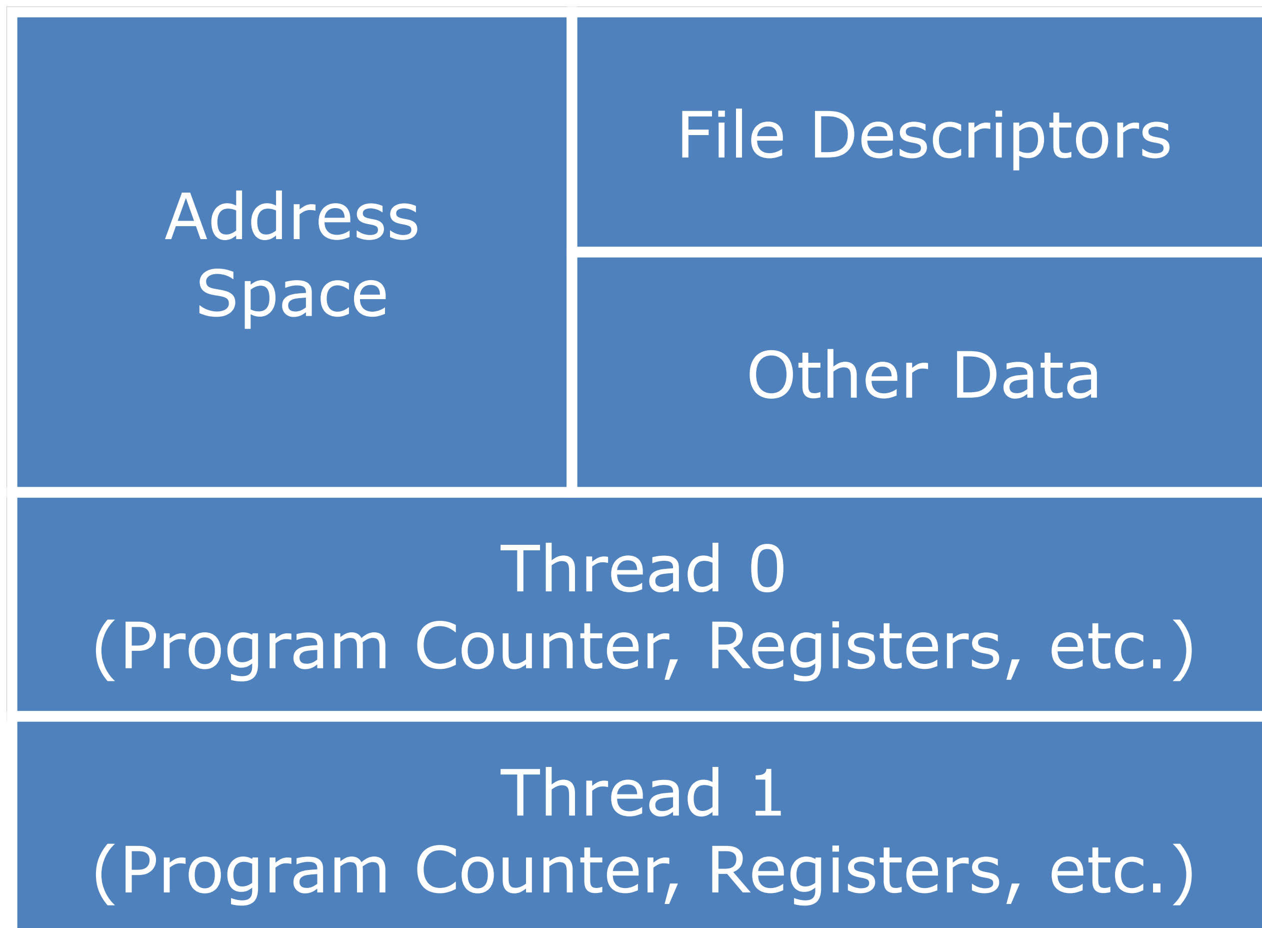
- A program in some state of execution
  - ✓ Code
  - ✓ Data
  - ✓ Logical address space
- Information about a process includes
  - ✓ Process state
  - ✓ Program counter
  - ✓ Values of CPU registers
  - ✓ Memory management information

What Is a Thread?

- "A unit of control within a process"
- ***Main thread*** executes program's "main" function
- Main thread may create other threads to execute other functions.
- Threads have own program counter, copy of CPU registers, and stack of activation records.
- Threads share process's data, code, address space, and other resources.
- Threads have lower overhead than processes.

Another Way of Looking at It

Process

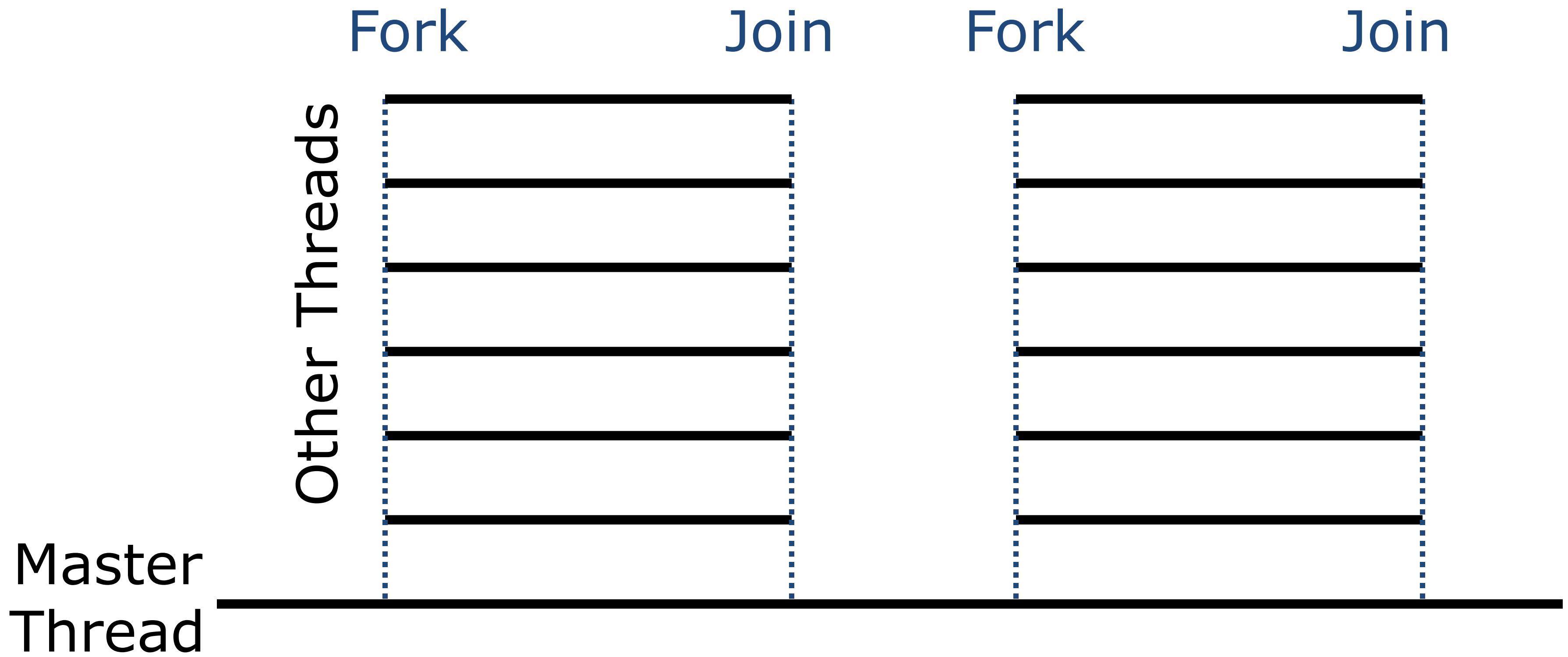


What Are Threads Good For?

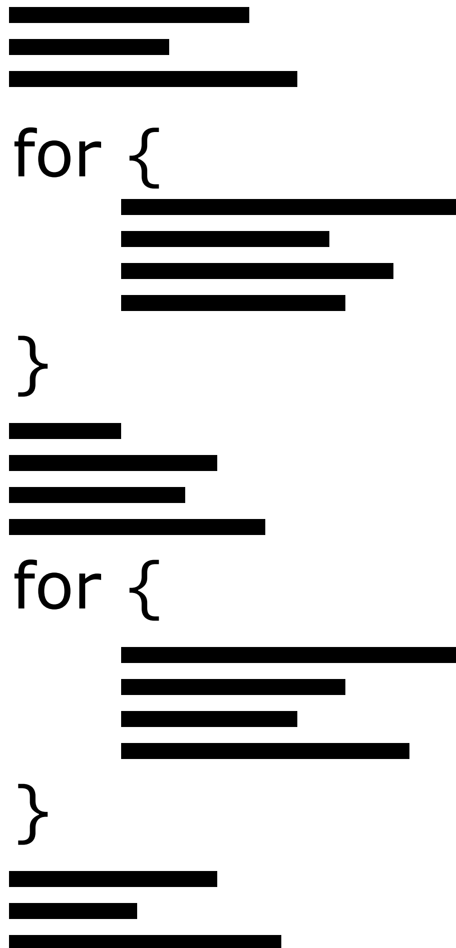
- Making programs easier to understand
- Overlapping computation and I/O
- Improving responsiveness of GUIs
- Improving application performance through parallel execution

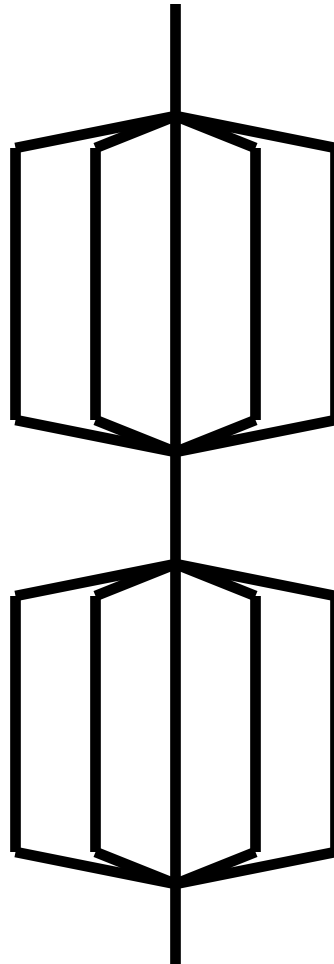
Fork/Join Programming Model

- ✓ When program begins execution, only master thread active
- ✓ Master thread executes sequential portions of program
- ✓ For parallel portions of program, master thread ***forks*** (creates or awakens) additional threads
- ✓ At ***join*** (end of parallel section of code), extra threads are suspended or die


Fork
Join
Fork
Join
Other Threads
Master Thread

Relating Fork/Join to Code





Sequential code

Parallel code

Sequential code

Parallel code

Sequential code

More General Threads Model

- ✓ When program begins execution, only one user thread, called the *main thread*, is active
- ✓ The main thread can create other threads, which execute other functions
- ✓ Created threads can also create additional threads
- ✓ How this is done varies according to programming language or API

Fork/Join versus General Model

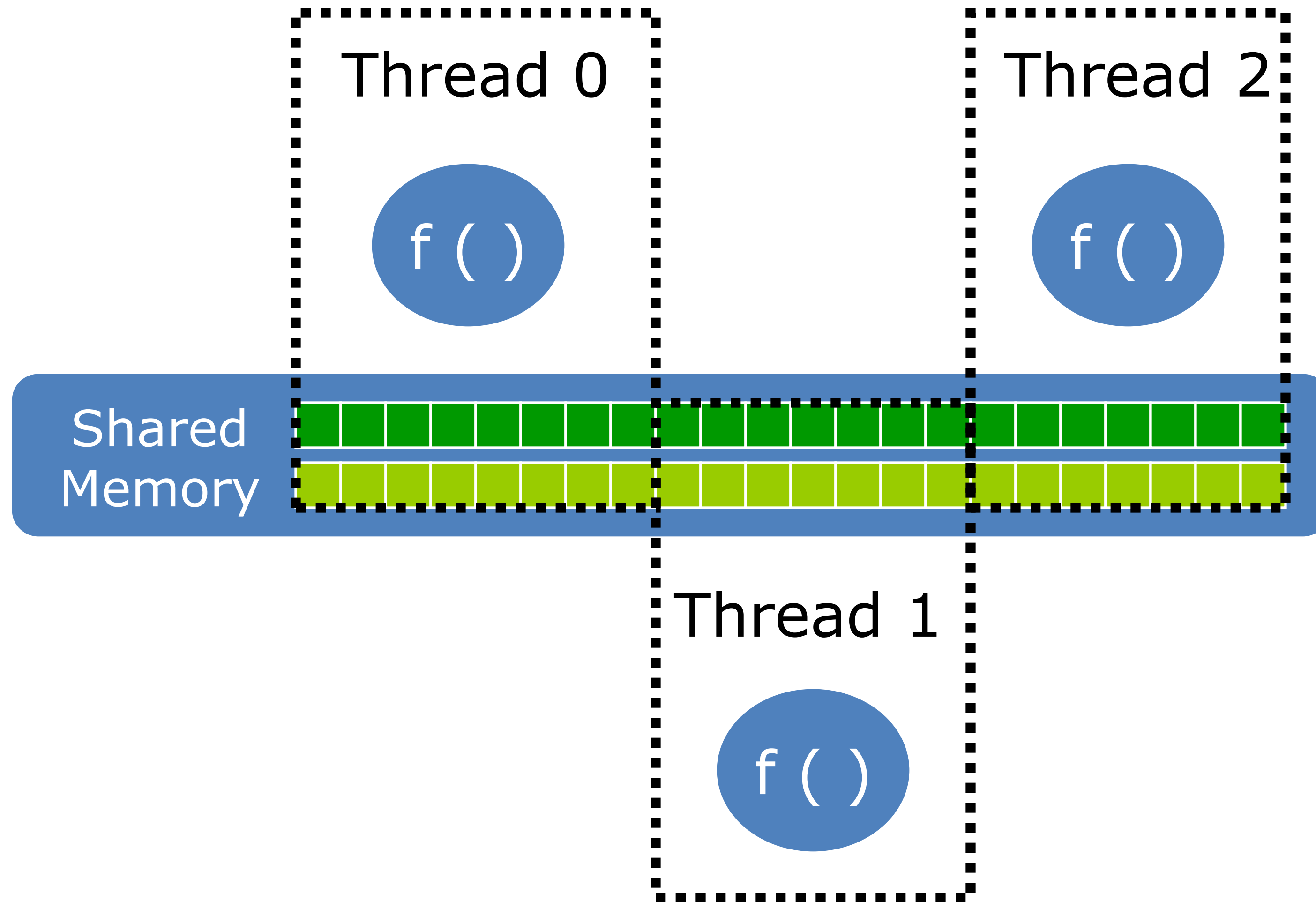
- ❖ You can implement fork/join in the general model
- ❖ Hence fork/join a special case of general model
  - ✓ More structured
  - ✓ More easily optimized
- ❖ General model
  - ✓ More flexible
  - ✓ Better support for task decompositions

Work Decomposition

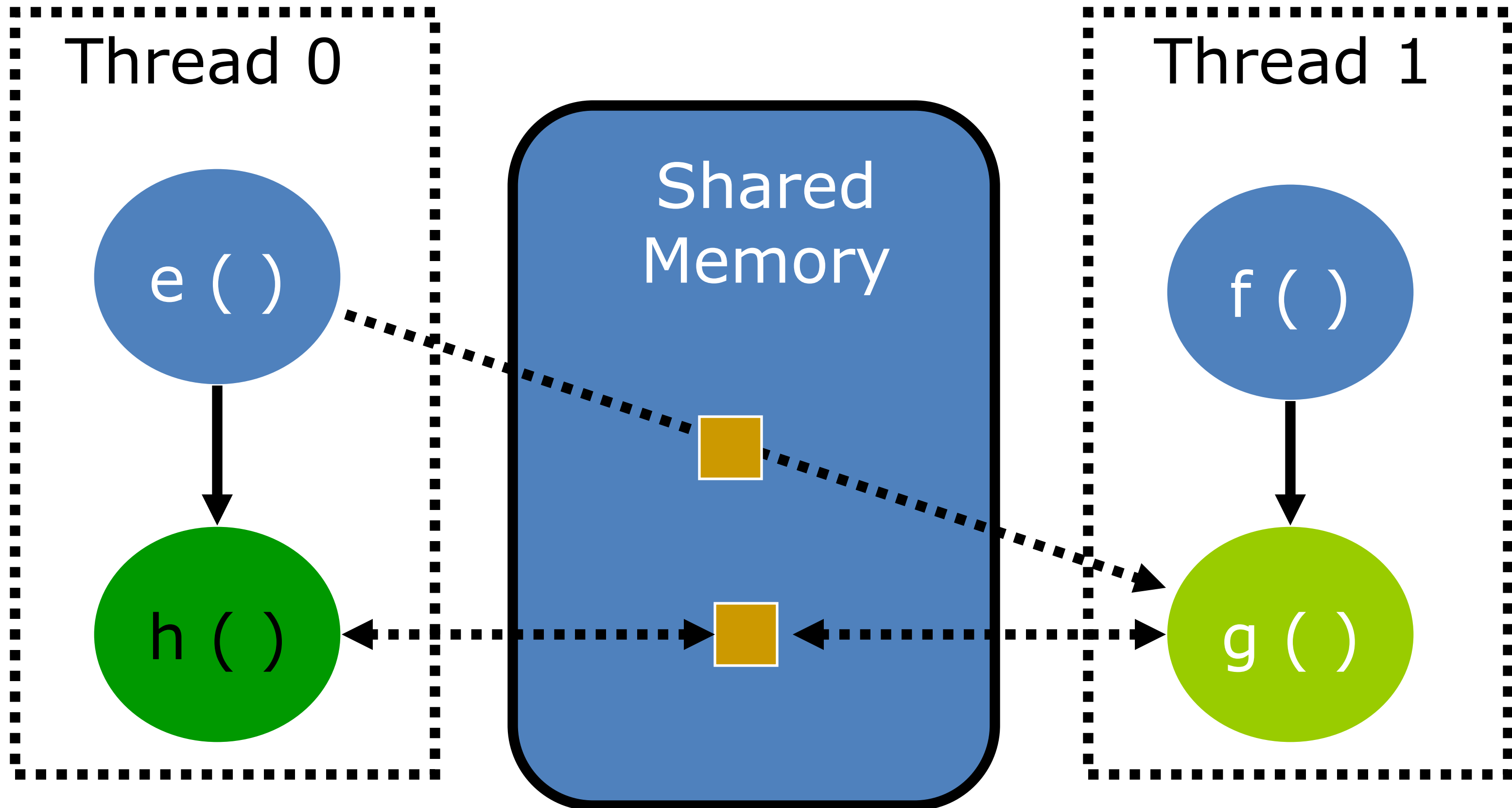
Threads are flexible enough to implement

- ✓ Domain decomposition
- ✓ Functional decomposition
- ✓ Pipelining

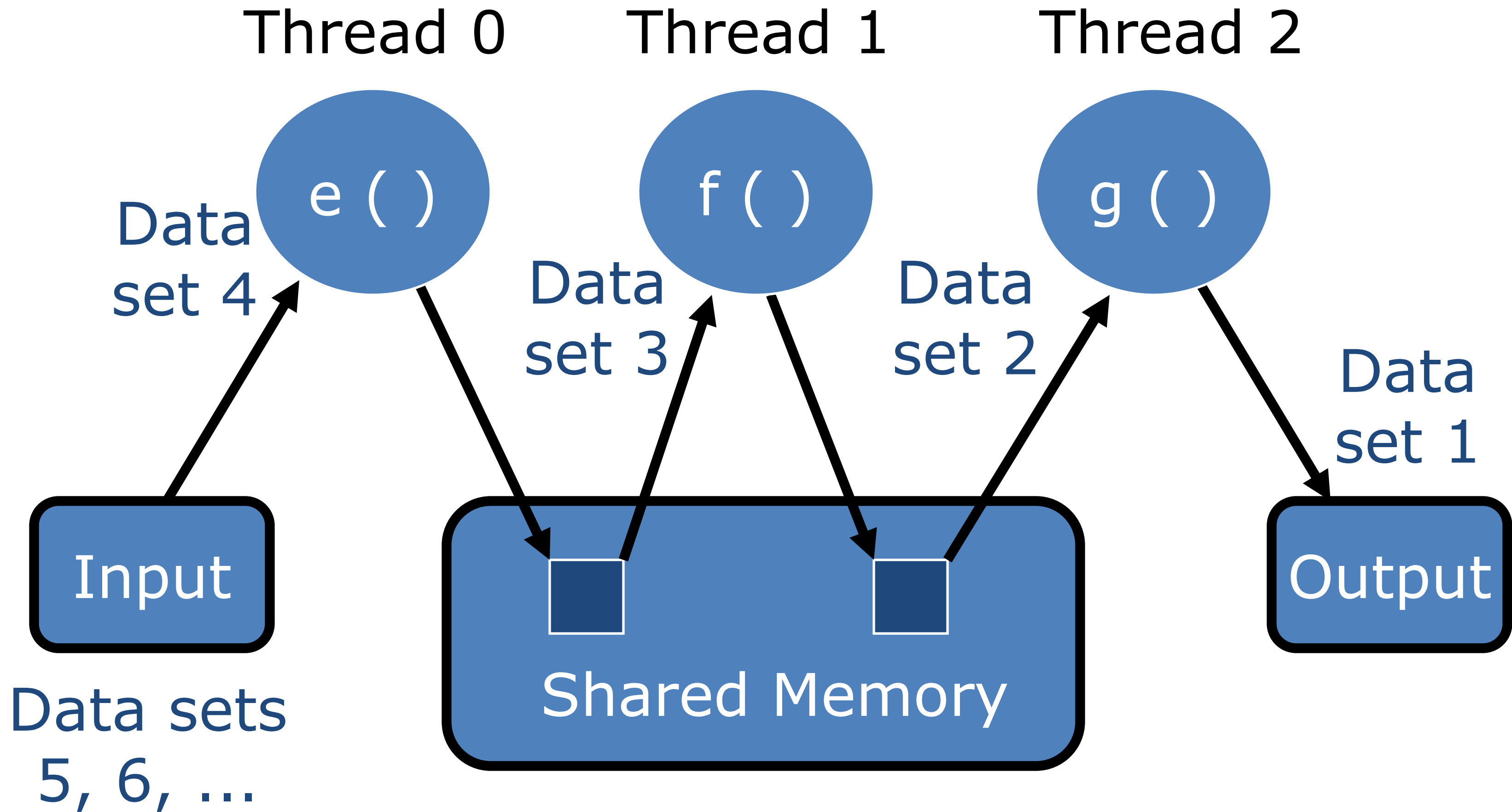
Domain Decomposition



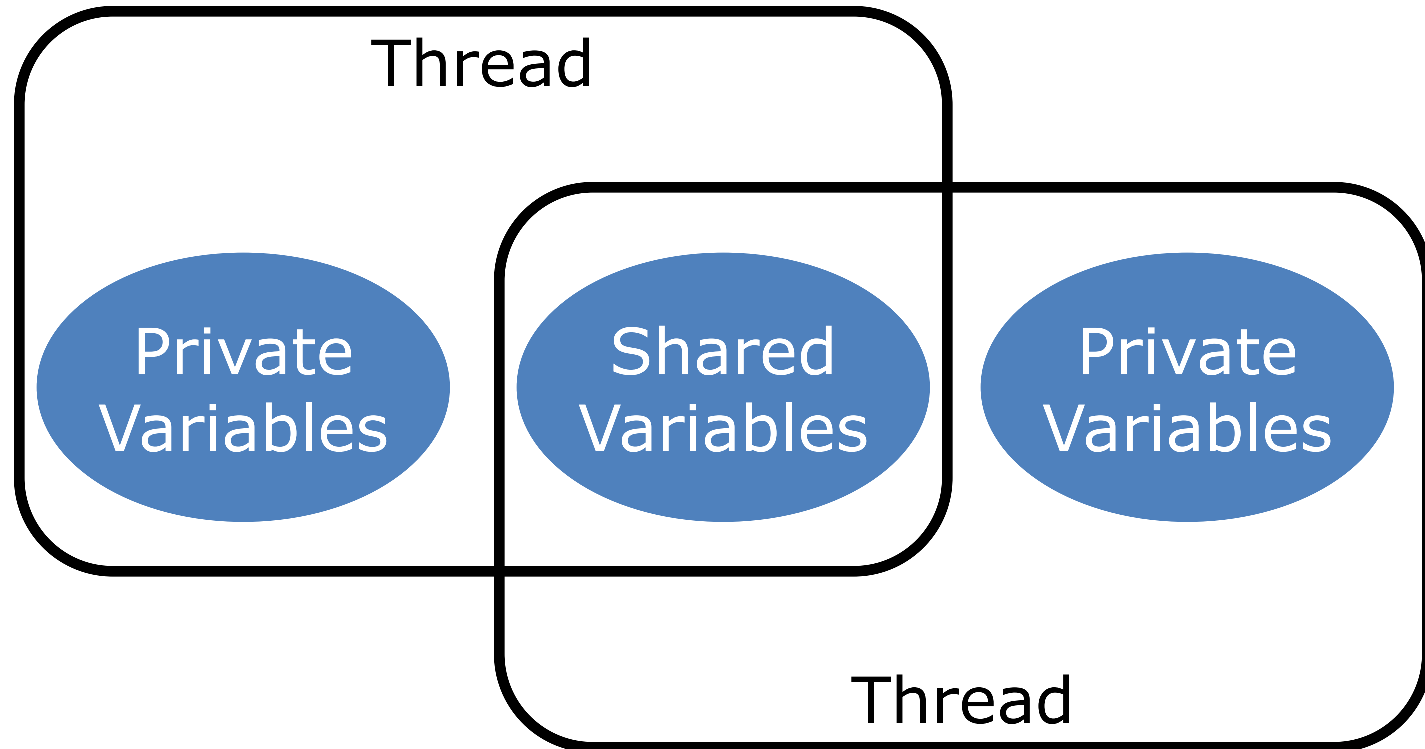
Functional Decomposition



Pipelining Using Threads



Shared versus Private Variables



Domain Decomposition

Sequential Code:

```
int a[1000], i;
for (i = 0; i < 1000; i++)
{
    a[i] = foo(i);
}
```

Domain Decomposition

- Sequential Code:

```
int a[1000], i;
for (i = 0; i < 1000; i++)
 a[i] = foo(i);
```

- Thread 0:

```
for (i = 0; i < 500; i++)
 a[i] = foo(i);
```

- Thread 1:

```
for (i = 500; i < 1000; i++)
 a[i] = foo(i);
```

Functional Decomposition

```
int e;

main () {
   int x[10], j, k, m;
   j = f(x, k);
   m = g(x, k); ...
}

int f(int *x, int k)
{
   int a;
   a = e * x[k] * x[k];
   return a;
}
```

```
int g(int *x, int k)
{
   int a;
   k = k-1;
   a = e / x[k];
   return a;
}
```

Functional Decomposition

```
int e;

main () {
    int x[10], j, k, m;
    j = f(x, k);
    m = g(x, k);
}
```

Thread 0

```
int f(int *x, int k)
{
    int a;   a = e * x[k] * x[k];   return a;
}
```

Thread 1

```
int g(int *x, int k)
{
    int a;   k = k-1;  a = e / x[k];   return a;
}
```

Functional Decomposition

```
int e;

main () {
    int x[10], j, k, m;
    j = f(x, k);
    m = g(x, k);
}
```

Static variable: Shared

Heap variable: Shared

Function's local variables: Private

Thread 0

```
int f(int *x, int k)
{
    int a;    a = e * x[k] * x[k];   return a;
}
```

Thread 1

```
int g(int *x, int k)
{
    int a;    k = k-1;   a = e / x[k];   return a;
}
```

Shared and Private Variables

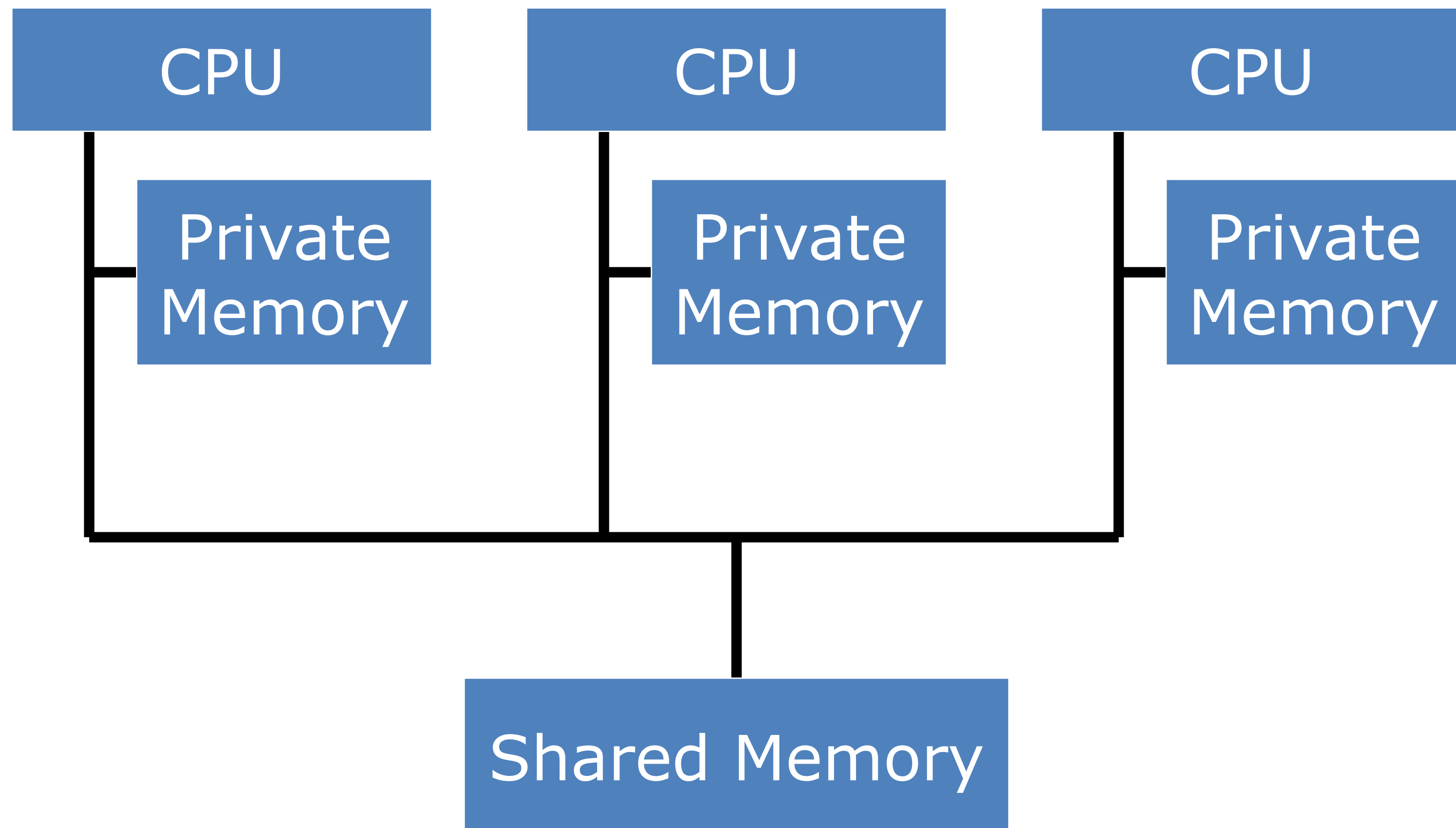
Shared variables

- ✓ Static variables
- ✓ Heap variables
- ✓ Contents of run-time stack at time of call

Private variables

- ✓ Loop index variables
- ✓ Run-time stack of functions invoked by thread

Shared-Memory Model (Reprise)



References

- Jim Beveridge and Robert Wiener, *Multithreading Applications in Win32®*, Addison-Wesley (1997).
- David R. Butenhof, *Programming with POSIX® Threads*, Addison-Wesley, (1997).
- Richard H. Carver and Kuo-Chung Tai, *Modern Multithreading: Implementing, Testing, and Debugging Java and C++/Pthreads/ Win32 Programs*, Wiley-Interscience (2006).
- Doug Lea, "A Java Fork/Join Framework," *Proceedings of the ACM 2000 Conference on Java Grande*, pp. 36-43.
- Michael J. Quinn, *Parallel Programming in C with MPI and OpenMP*, McGraw-Hill (2004).