CS 333
Introduction to Operating Systems

Class 3 – Threads & Concurrency

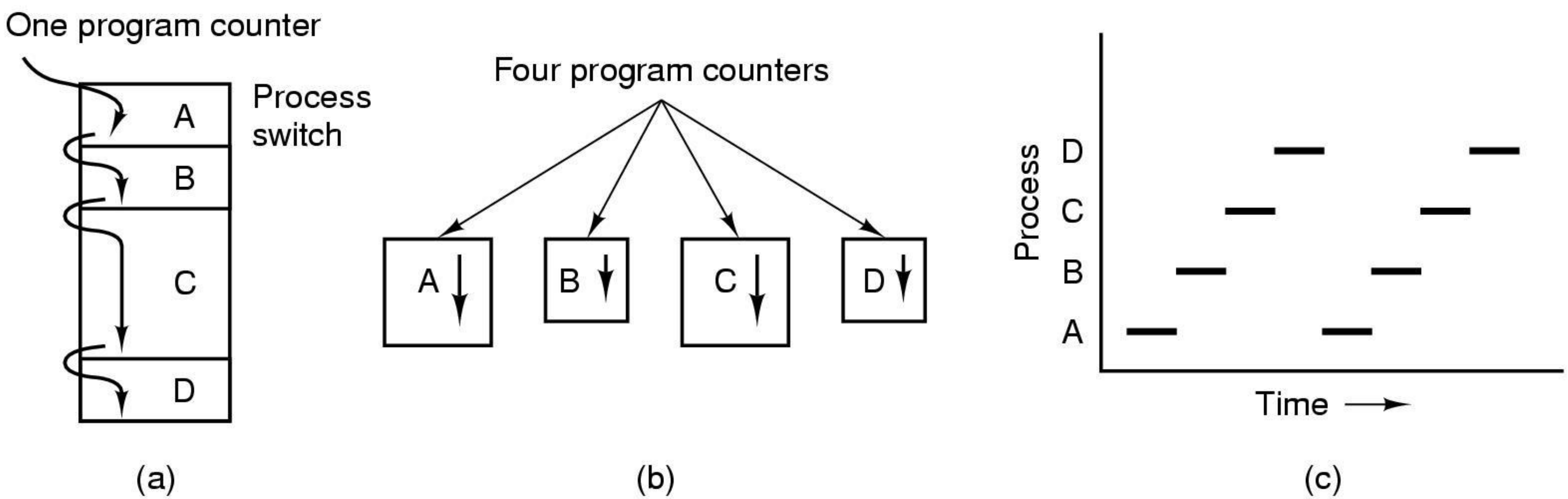
Jonathan Walpole
Computer Science
Portland State University

The Process Concept

The Process Concept

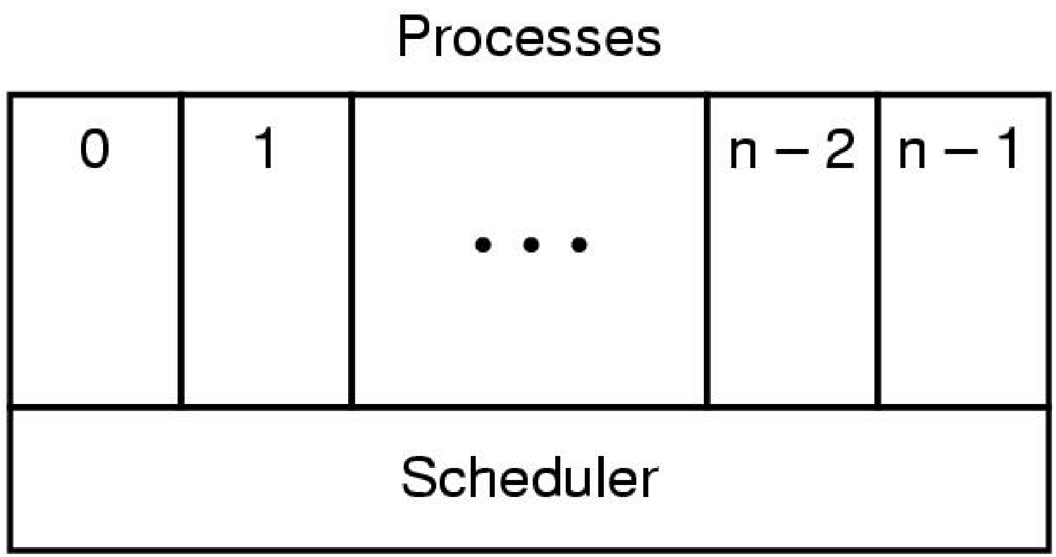
- **Process – a program in execution**
  - Program
    - **description of how to perform an activity**
    - **instructions and static data values**
  - Process
    - **a snapshot of a program in execution**
    - **memory (program instructions, static and dynamic data values)**
    - **CPU state (registers, PC, SP, etc)**
    - **operating system state (open files, accounting statistics etc)**

Why use the process abstraction?



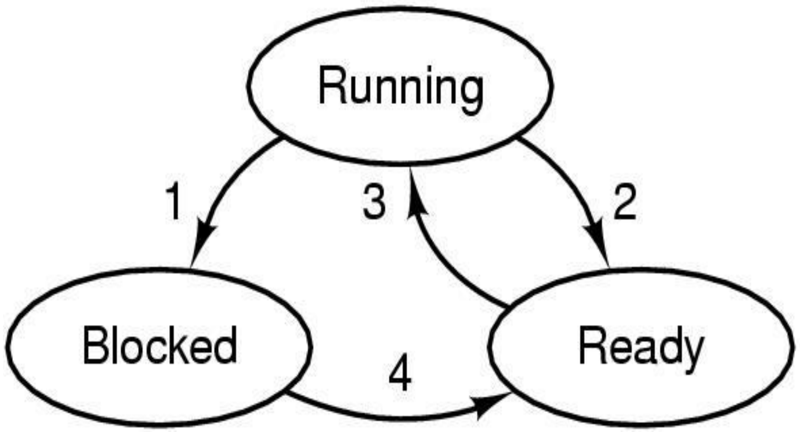
- **Multiprogramming of four programs in the same address space**
- **Conceptual model of 4 independent, sequential processes**
- **Only one program active at any instant**

The role of the scheduler



- **Lowest layer of process-structured OS**
  - handles interrupts & scheduling of processes
- **Sequential processes only exist above that layer**

Process states



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

- **Possible process states**
  - running
  - blocked
  - ready

How do processes get created?

Principal events that cause process creation

- System initialization
- Initiation of a batch job
- User request to create a new process
- Execution of a process creation system call from another process

Process hierarchies

- **Parent creates a child process,**
  - special system calls for communicating with and waiting for child processes
  - each process is assigned a unique identifying number or process ID (PID)
- **Child processes can create their own child processes**
  - Forms a hierarchy
  - UNIX calls this a "process group"

Process creation in UNIX

- **All processes have a unique process id**
  - *getpid(), getppid()* system calls allow processes to get their information
- **Process creation**
  - *fork()* system call creates a copy of a process and returns in both processes (parent and child), but with a different return value
  - *exec()* replaces an address space with a new program
- **Process termination, signaling**
  - *signal(), kill()* system calls allow a process to be terminated or have specific signals sent to it

Example: process creation in UNIX

csh (pid = 22)

```
…

pid = fork()
if (pid == 0) {
  // child…
  …
  exec();
  }
else {
  // parent
  wait();
  }
…
```

Process creation in UNIX example

csh (pid = 22)

```
…

pid = fork()
if (pid == 0) {
  // child…
  …
  exec();
  }
else {
  // parent
  wait();
  }
…
```

csh (pid = 24)

```
…

pid = fork()
if (pid == 0) {
  // child…
  …
  exec();
  }
else {
  // parent
  wait();
  }
…
```

Process creation in UNIX example

csh (pid = 22)

```
…

pid = fork()
if (pid == 0) {
   // child…
   …
   exec();
   }
else {
   // parent
   wait();
   }
…
```

csh (pid = 24)

```
…

pid = fork()
if (pid == 0) {
   // child…
   …
   exec();
   }
else {
   // parent
   wait();
   }
…
```

Process creation in UNIX example

csh (pid = 22)

```
…

pid = fork()
if (pid == 0) {
   // child…
   …
   exec();
   }
else {
   // parent
   wait();
   }
…
```

csh (pid = 24)

```
…

pid = fork()
if (pid == 0) {
   // child…
   …
   exec();
   }
else {
   // parent
   wait();
   }
…
```

Process creation in UNIX example

csh (pid = 22)

```
…

pid = fork()
if (pid == 0) {
  // child…
  …
  exec();
  }
else {
  // parent
  wait();
  }
…
```

ls (pid = 24)

```
//ls program
main(){
  //look up dir
  …
}
```

Process creation (fork)

- **Fork creates a new process by *copying* the calling process**
- **The new process has its own**
  - memory address space (copied from parent)
    - **Instructions**
    - **Data**
    - **Stack**
  - Register set (copied from parent)
  - Process table entry in the OS

Threads

Threads

- **Processes have the following components:**
  - an address space
  - a collection of operating system state
  - a CPU context … or *thread* of control
- **On multiprocessor systems, with several CPUs, it would make sense for a process to have several CPU contexts (threads of control)**
  - Thread fork creates new thread not memory space
  - Multiple threads of control could run in the same memory space on a single CPU system too!

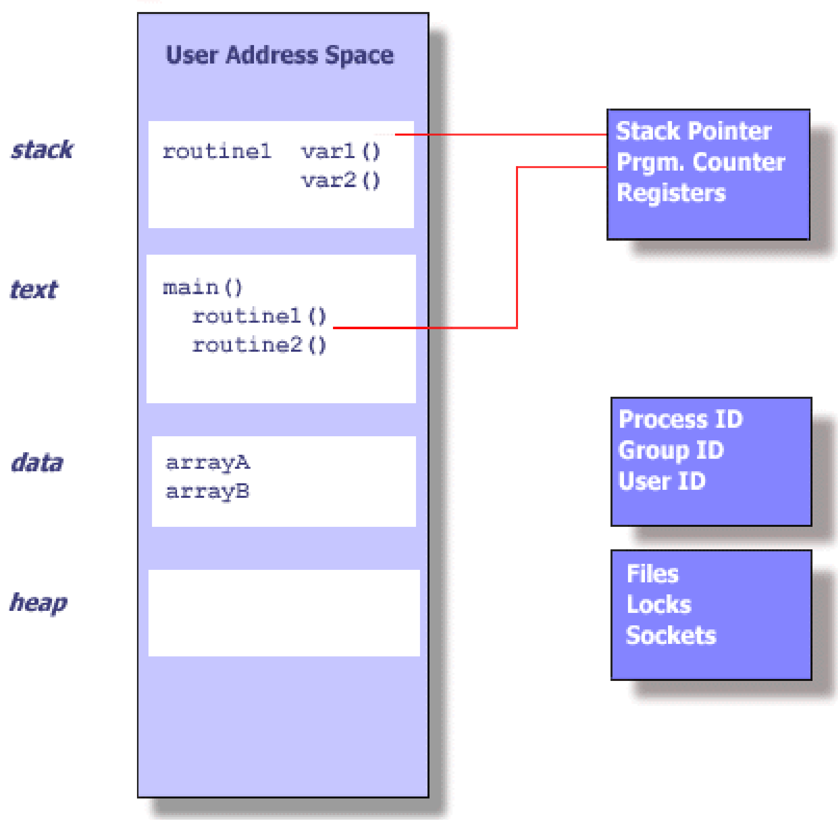
Threads

- **Threads share a process address space with zero or more other threads**

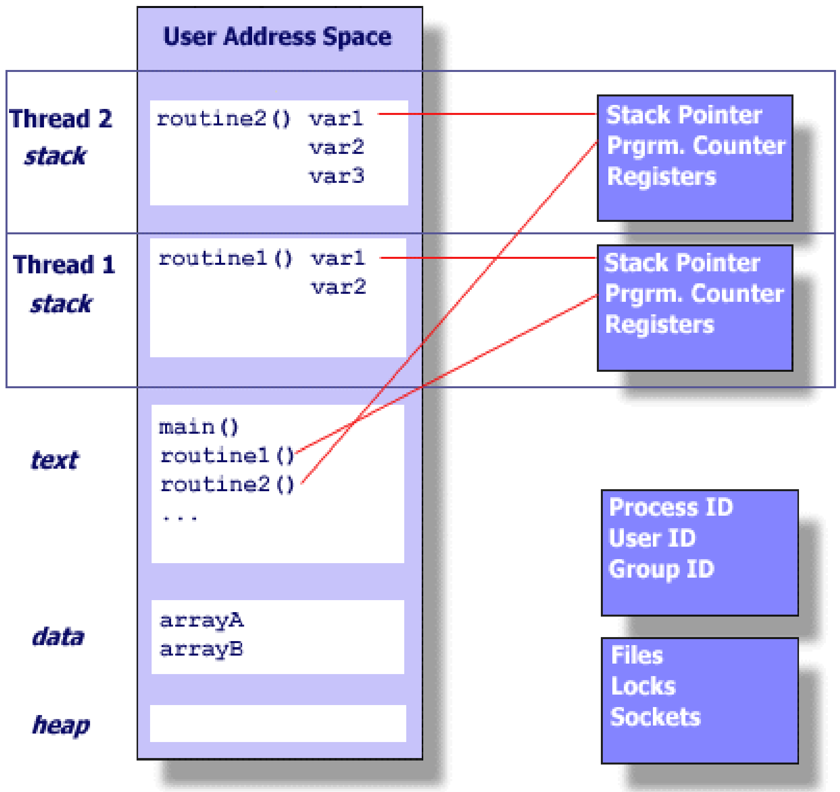
- **Threads have their own CPU context**
  - PC, SP, register state,
  - stack

- **A traditional process can be viewed as a memory address space with a single thread**

Single thread state within a process



Multiple threads in an address space



What is a thread?

- **A thread executes a stream of instructions**
  - it is an abstraction for control-flow
- **Practically, it is a processor context and stack**
  - Allocated a CPU by a scheduler
  - Executes in the context of a memory address space

Summary of private per-thread state

Things that define the state of a particular flow of control in an executing program:

- Stack (local variables)
- Stack pointer
- Registers
- Scheduling properties (i.e., priority)

Shared state among threads

Things that relate to an instance of an executing program (that may have multiple threads)

- User ID, group ID, process ID
- Address space
  - **Text**
  - **Data (off-stack global variables)**
  - **Heap (dynamic data)**
- Open files, sockets, locks

Important: Changes made to shared state by one thread will be visible to the others

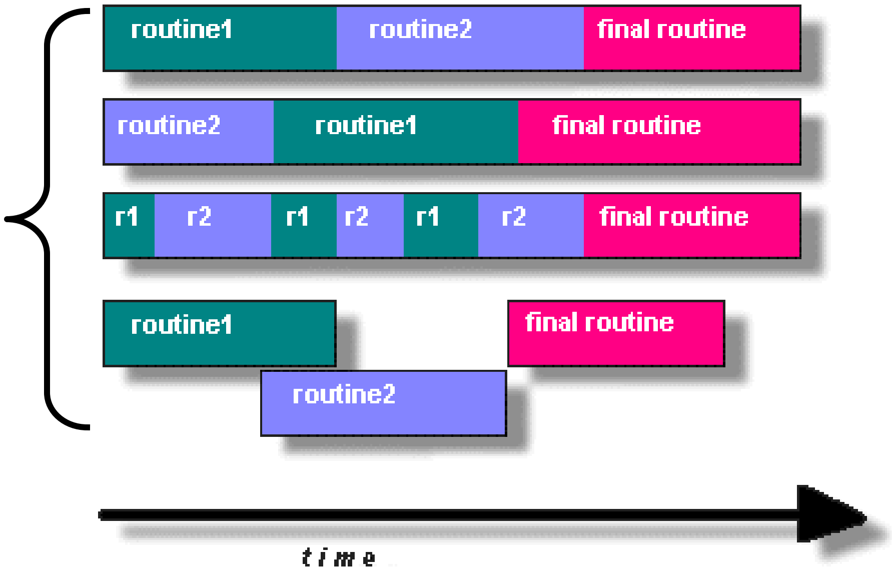
- Reading and writing memory locations requires synchronization! … a major topic for later …

How do you program using threads?

Split program into routines to execute in parallel

- True or pseudo (interleaved) parallelism

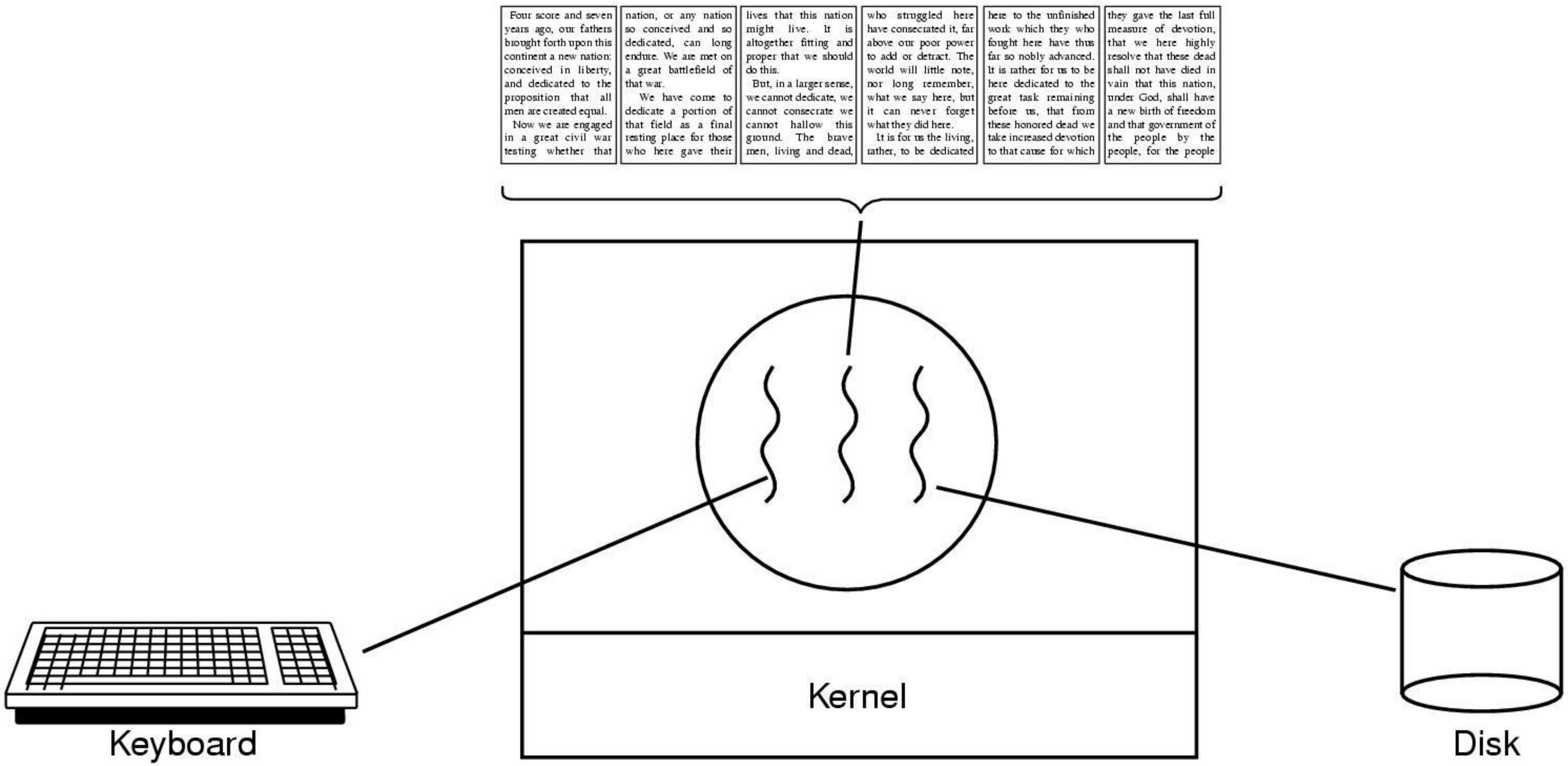
Alternative strategies for executing multiple rountines



Why program using threads?

- **Utilize multiple CPU's concurrently**
- **Low cost communication via shared memory**
- **Overlap computation and blocking on a single CPU**
    - Blocking due to I/O
    - Computation and communication
- **Handle asynchronous events**

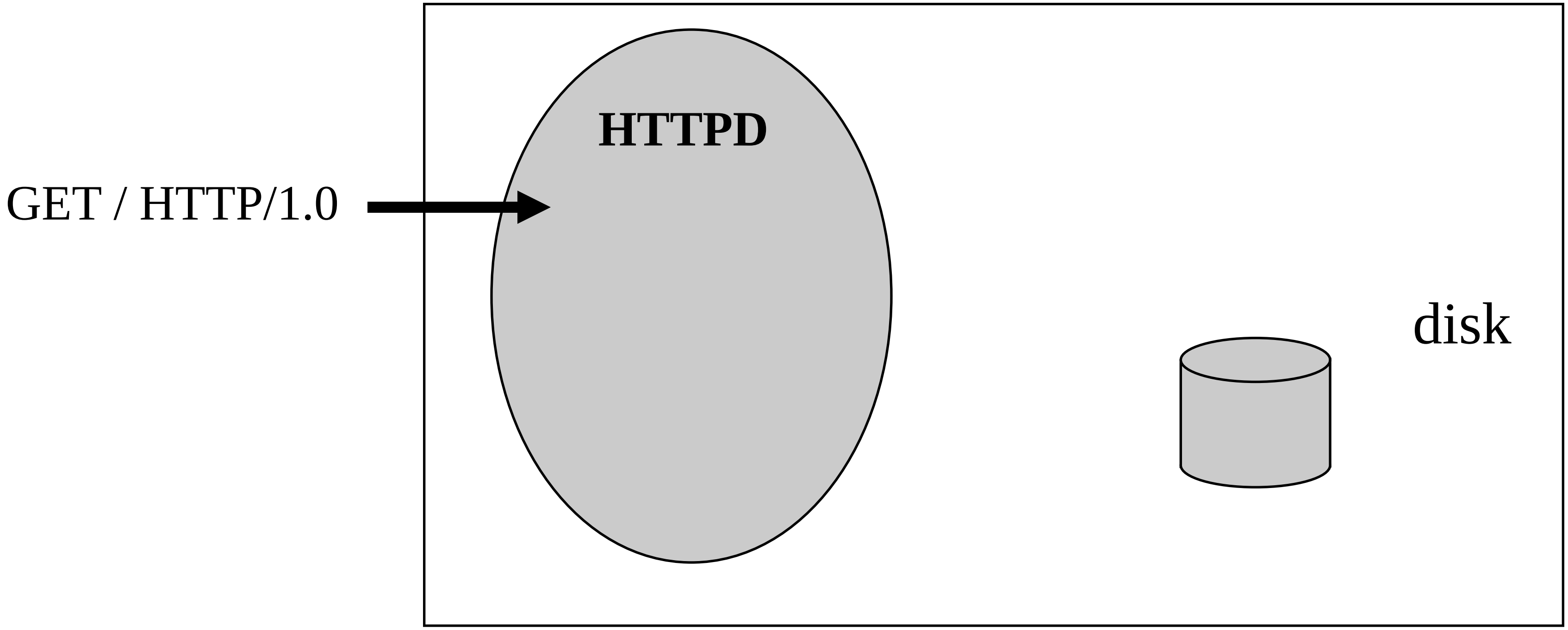
Thread usage



A word processor with three threads

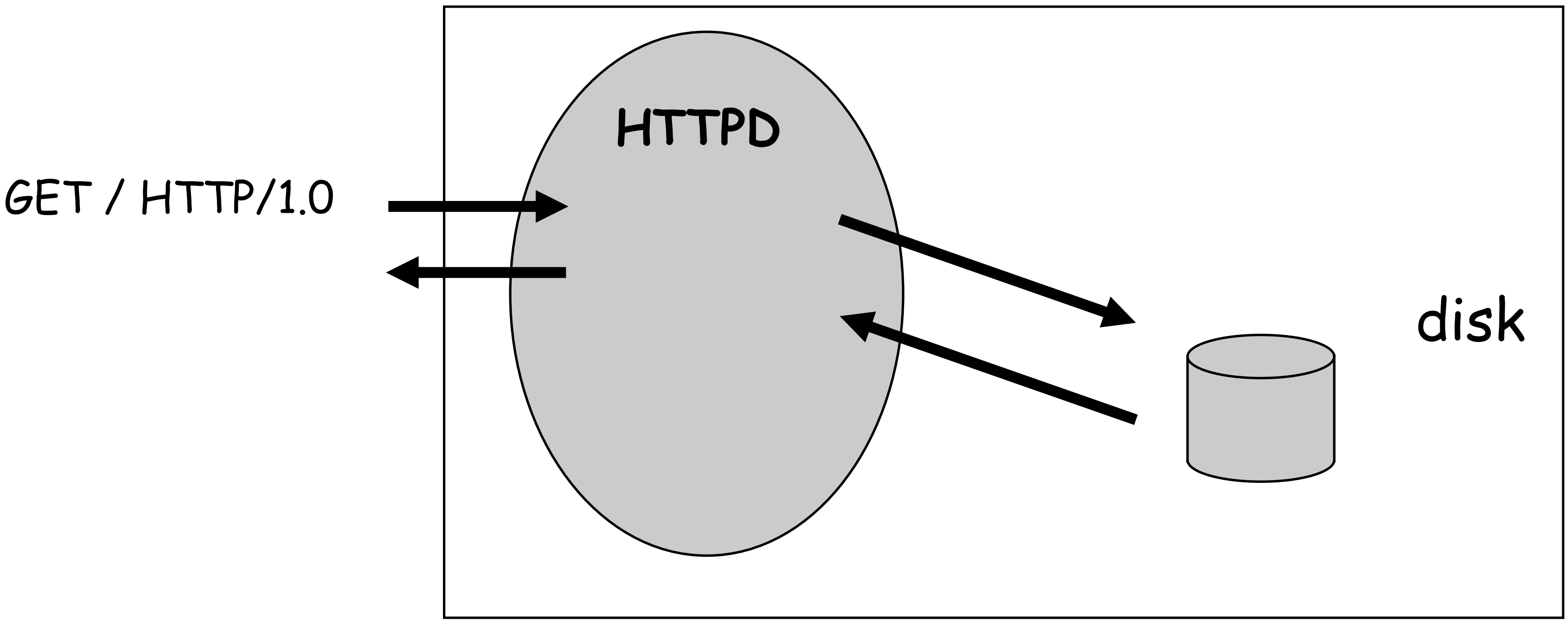
Processes versus threads - example

- A WWW process



Processes versus threads - example

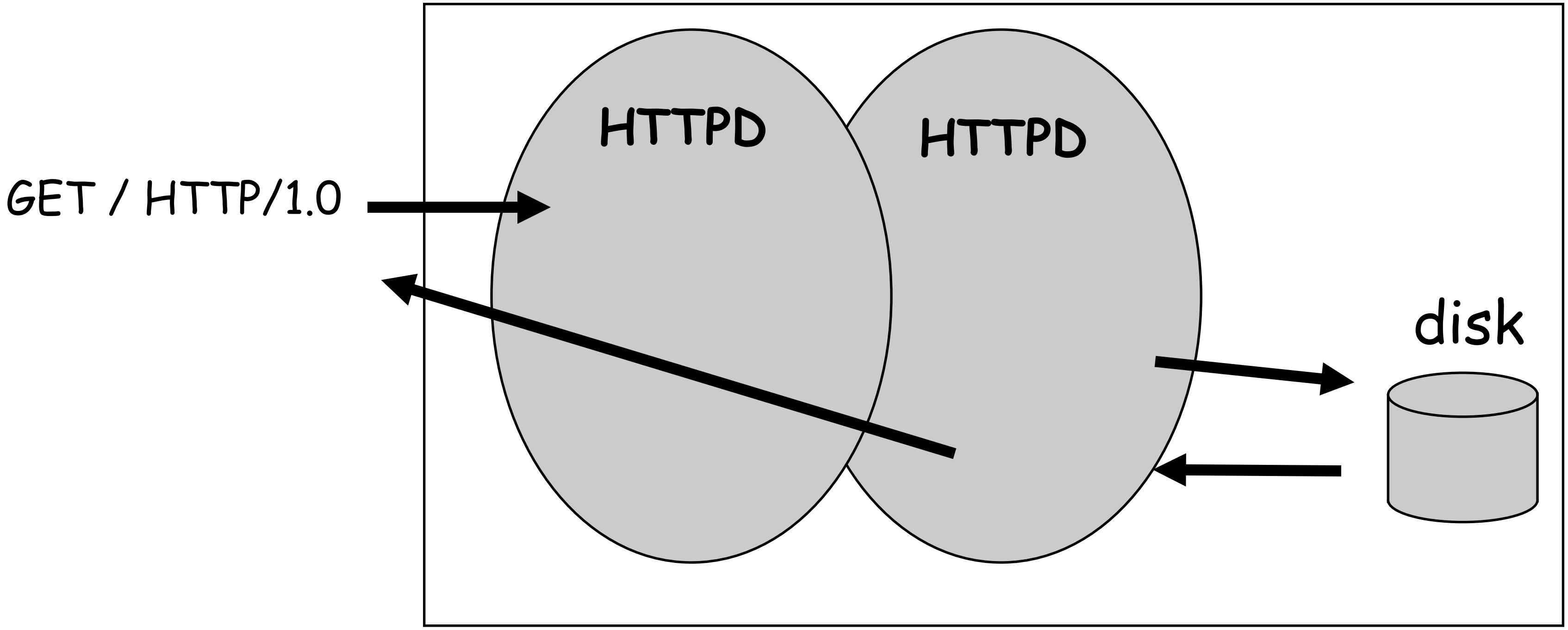
- **A WWW process**



Why is this not a good web server design?

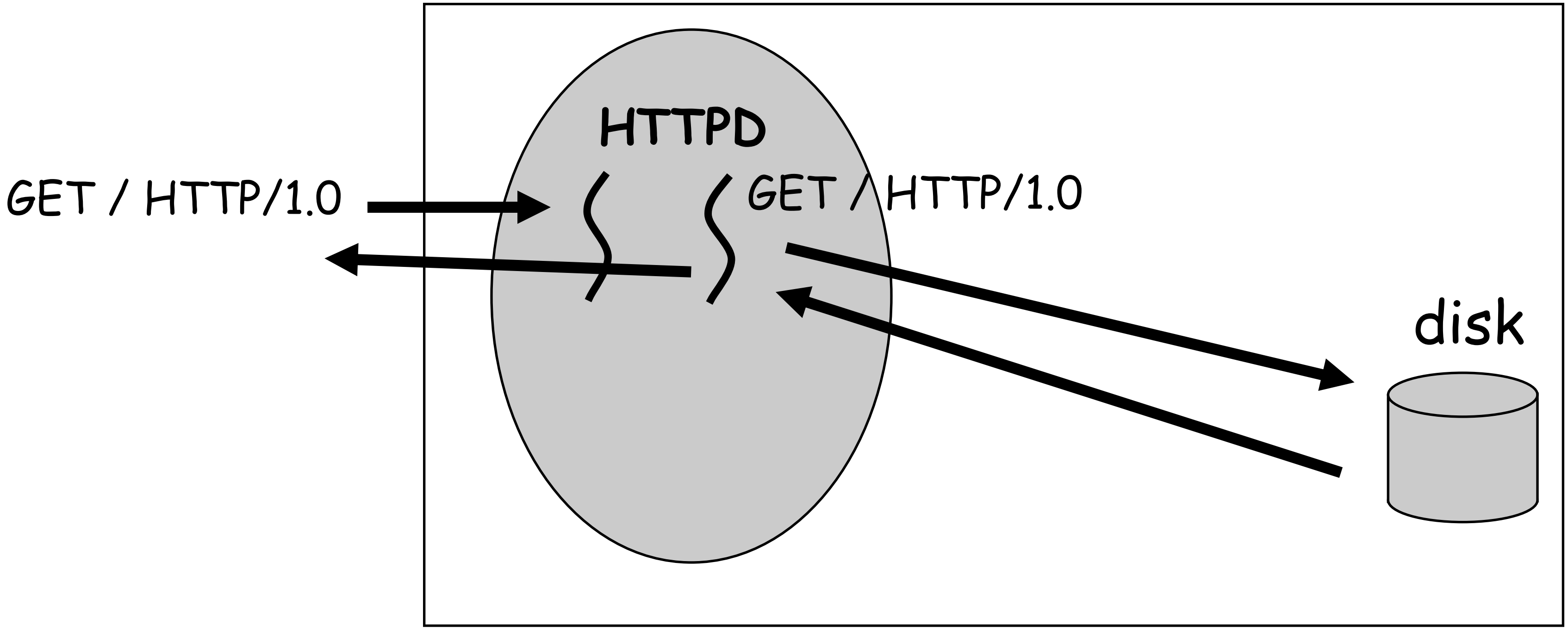
Processes versus threads - example

- A WWW process



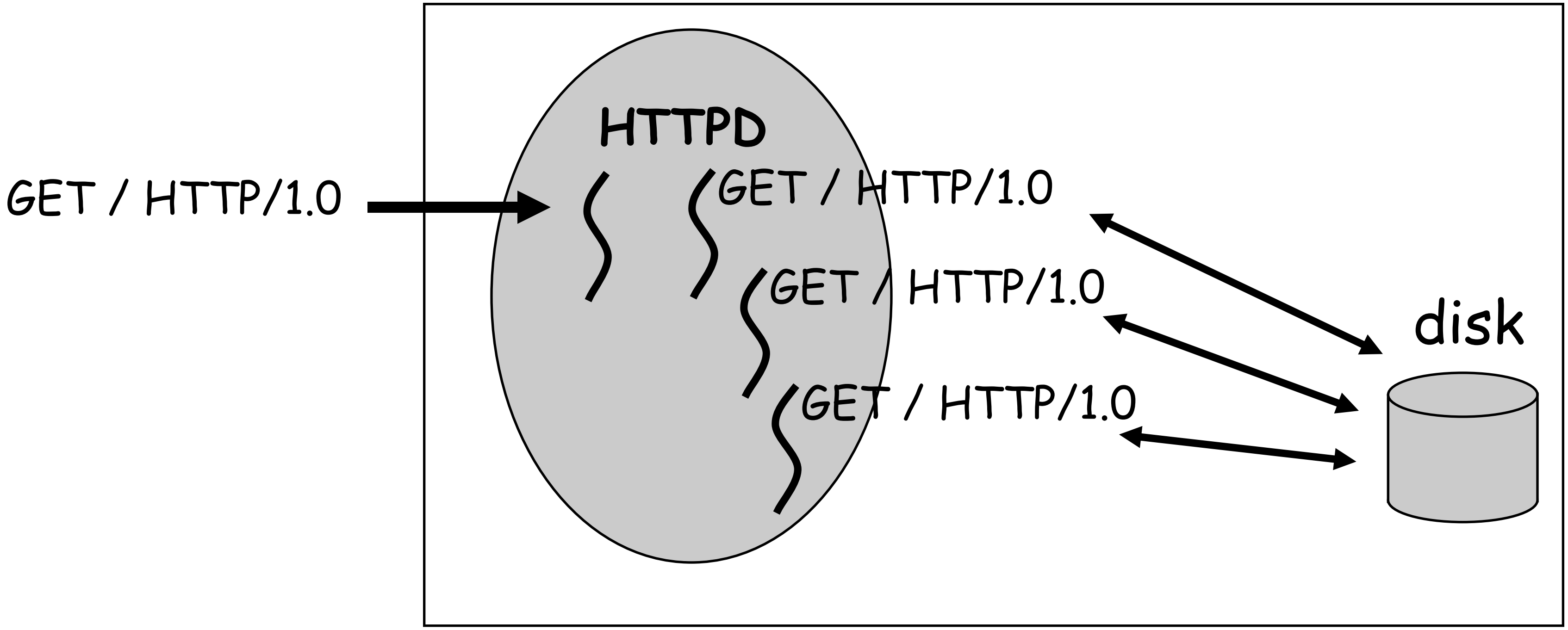
Processes versus threads - example

- **A WWW process**



Processes versus threads - example

- **A WWW process**



System structuring options

Model	Characteristics
Threads	Parallelism, blocking system calls
Single-threaded process	No parallelism, blocking system calls
Finite-state machine	Parallelism, nonblocking system calls, interrupts

Three ways to construct a server

Common thread programming models

- **Manager/worker**
  - Manager thread handles I/O and assigns work to worker threads
  - Worker threads may be created dynamically, or allocated from a thread-pool
- **Pipeline**
  - Each thread handles a different stage of an assembly line
  - Threads hand work off to each other in a producer-consumer relationship

What does a typical thread API look like?

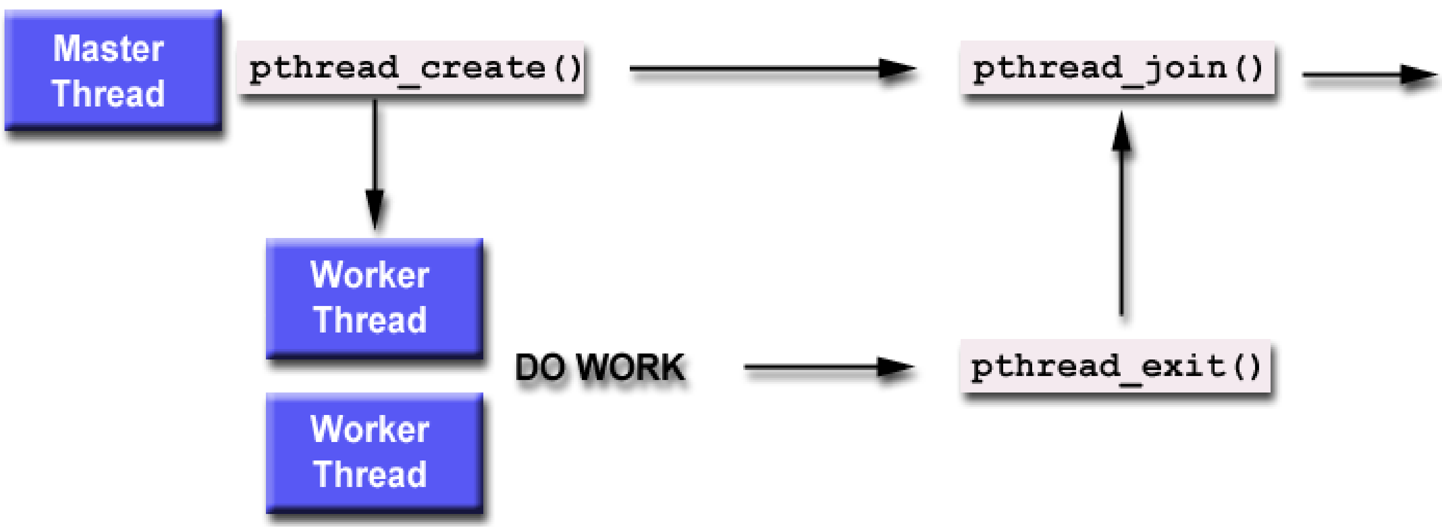
- **POSIX standard threads (Pthreads)**
- **First thread exists in main(), typically creates the others**

- **pthread_create (thread,attr,start_routine,arg)**
  - Returns new thread ID in "thread"
  - Executes routine specified by "start_routine" with argument specified by "arg"
  - Exits on return from routine or when told explicitly

Thread API (continued)

- **pthread_exit (status)**
  - Terminates the thread and returns “status” to any joining thread
- **pthread_join (threadid,status)**
  - Blocks the calling thread until thread specified by “threadid” terminates
  - Return status from pthread_exit is passed in “status”
  - One way of synchronizing between threads
- **pthread_yield ()**
  - Thread gives up the CPU and enters the run queue

Using create, join and exit primitives



An example Pthreads program

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

void *PrintHello(void *threadid)
{
  printf("\n%d: Hello World!\n", threadid);
  pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
  pthread_t threads[NUM_THREADS];
  int rc, t;
  for(t=0; t<NUM_THREADS; t++)
  {
    printf("Creating thread %d\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);
    }
  }
  pthread_exit(NULL);
}
```

Program Output

```
Creating thread 0
Creating thread 1
0: Hello World!
1: Hello World!
Creating thread 2
Creating thread 3
2: Hello World!
3: Hello World!
Creating thread 4
4: Hello World!
```

For more examples see: http://www.llnl.gov/computing/tutorials/pthreads

Pros & cons of threads

- **Pros**
  - Overlap I/O with computation!
  - Cheaper context switches
  - Better mapping to shared memory multiprocessors

- **Cons**
  - Potential thread interactions
  - Complexity of debugging
  - Complexity of multi-threaded programming
  - Backwards compatibility with existing code

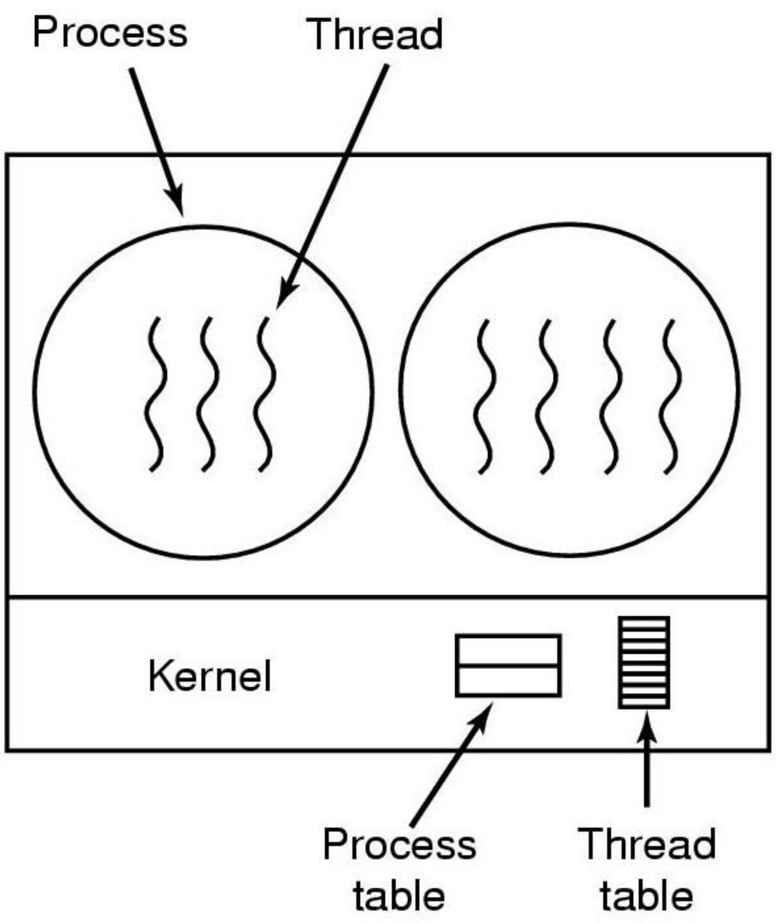
User-level threads

- **The idea of managing multiple abstract program counters above a single real one can be implemented using privileged or non-privileged code.**
  - Threads can be implemented in the OS or at user level

- **User level thread implementations**
  - thread scheduler runs as user code (thread library)
  - manages thread contexts in user space
  - The underlying OS sees only a traditional process above

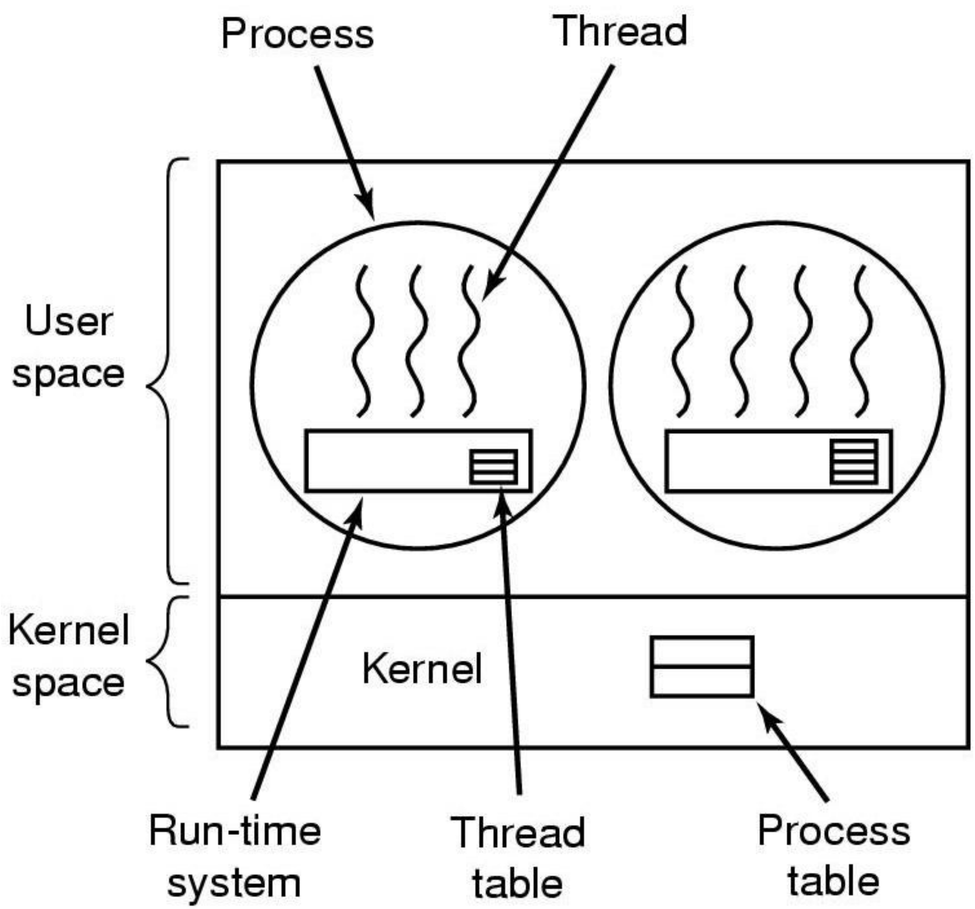
Kernel-level threads

The thread-switching code is in the kernel



User-level threads package

The thread-switching code is in user space



User-level threads

- **Advantages**
  - cheap context switch costs among threads in the same process!
    - **A procedure call not a system call!**
  - User-programmable scheduling policy
- **Disadvantages**
  - How to deal with blocking system calls!
  - How to overlap I/O and computation!

Concurrent Programming

Concurrent programming

Assumptions:

- Two or more threads
- Each executes in (pseudo) parallel
- We can't predict exact running speeds
- The threads can interact via access to shared variables

Example:

- One thread writes a variable
- The other thread reads from the same variable
- Problem – non-determinism:
  - *The relative order of one thread's reads and the other thread's writes determines the end result!*

Race conditions

- **What is a race condition?**
  - two or more threads have an inconsistent view of a shared memory region (I.e., a variable)
- **Why do race conditions occur?**

Race conditions

- A simple multithreaded program with a race:

```
i++;
```

Race conditions

- A simple multithreaded program with a race:

```
...
load i to register;
increment register;
store register to i;
...
```

Race conditions

- **What is a race condition?**
  - two or more threads have an inconsistent view of a shared memory region (I.e., a variable)
- **Why do race conditions occur?**
  - values of memory locations replicated in registers during execution
  - context switches at arbitrary times during execution
  - threads can see "stale" memory values in registers

Race Conditions

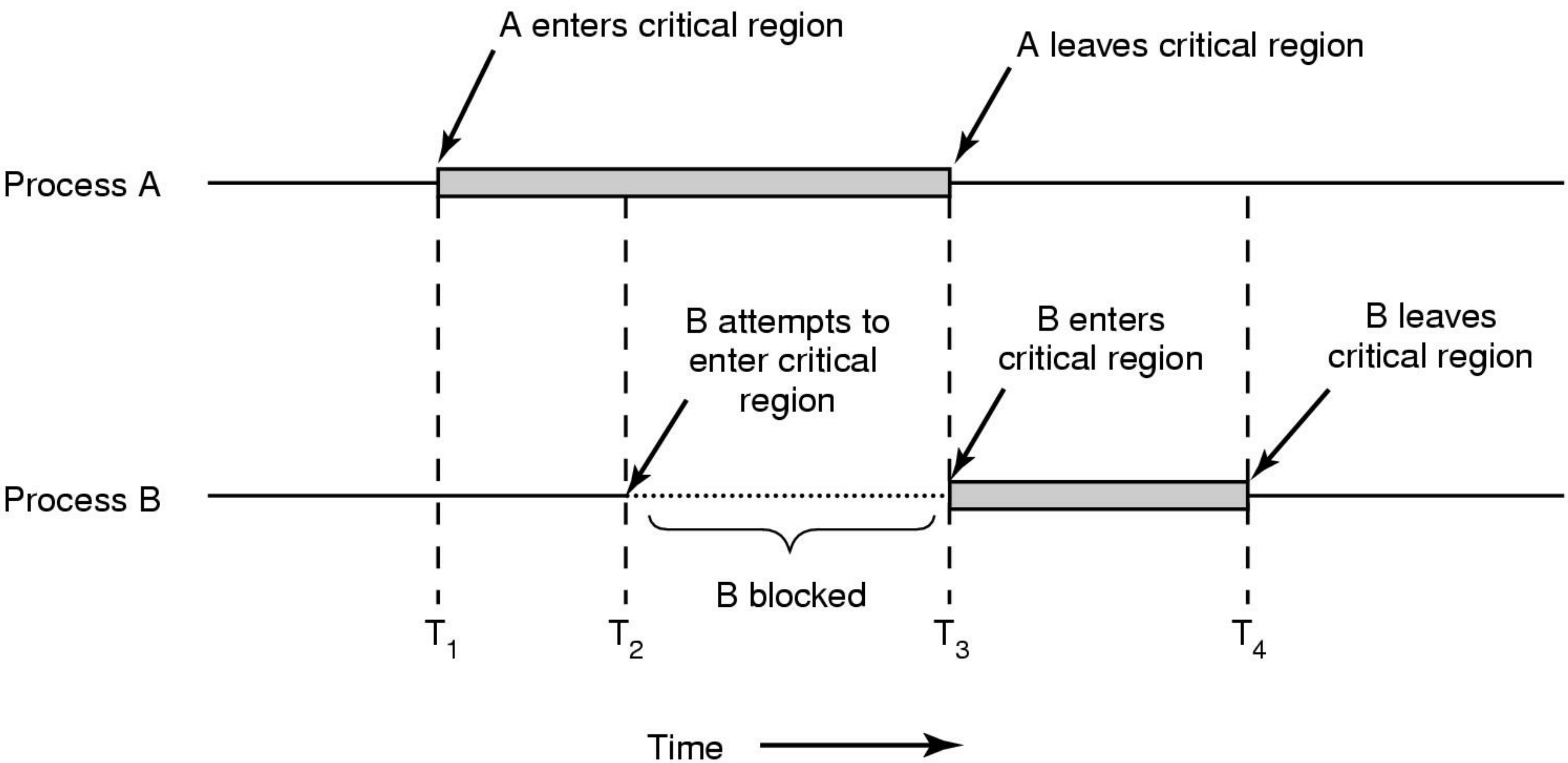
- **Race condition: whenever the output depends on the precise execution order of the processes!**
- **What solutions can we apply?**
  - prevent context switches by preventing interrupts
  - make threads coordinate with each other to ensure mutual exclusion in accessing critical sections of code

Mutual exclusion conditions

- No two processes simultaneously in critical section
- No assumptions made about speeds or numbers of CPUs
- No process running outside its critical section may block another process
- No process must wait forever to enter its critical section

Spare Slides - intended for class 4

Critical sections with mutual exclusion

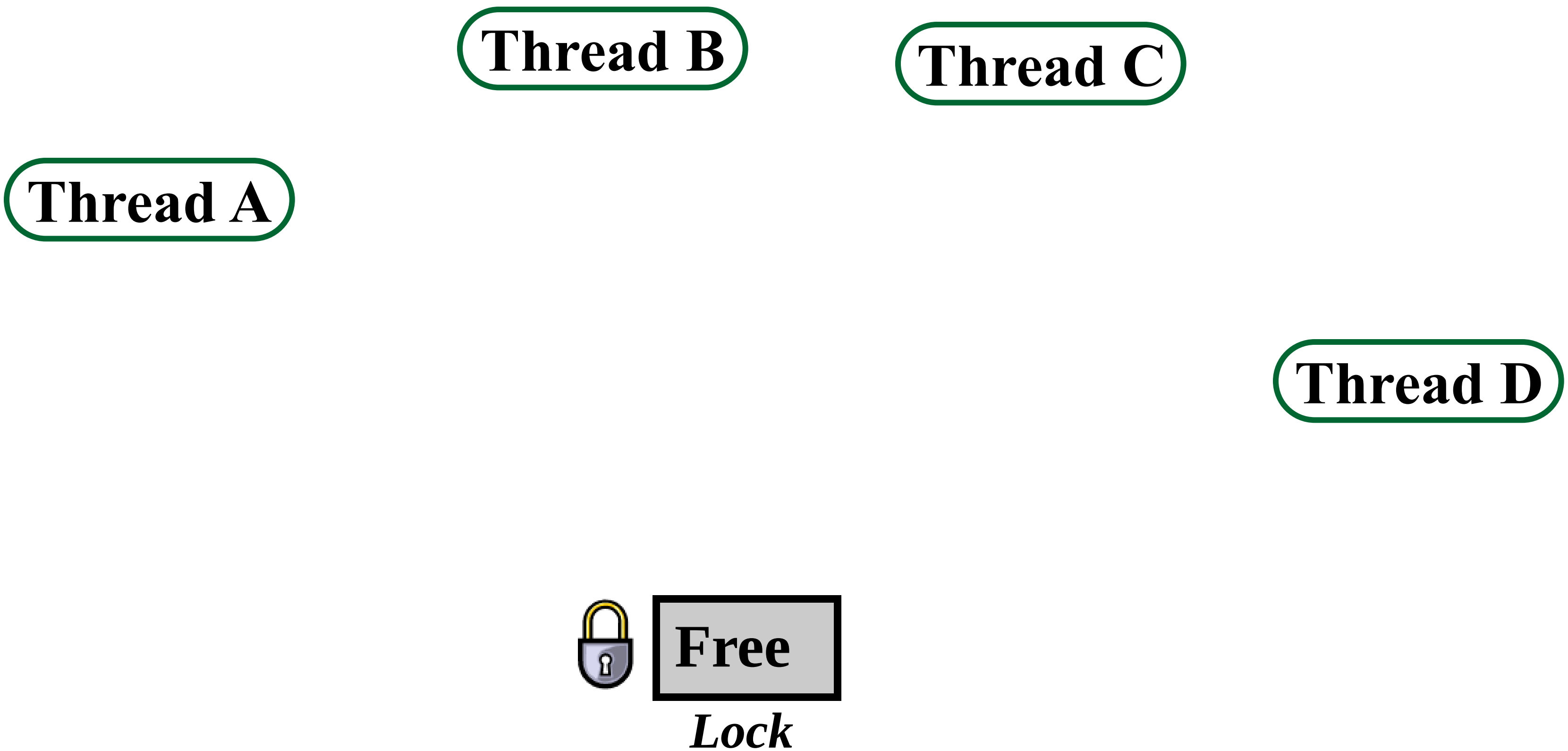


How can we enforce mutual exclusion?

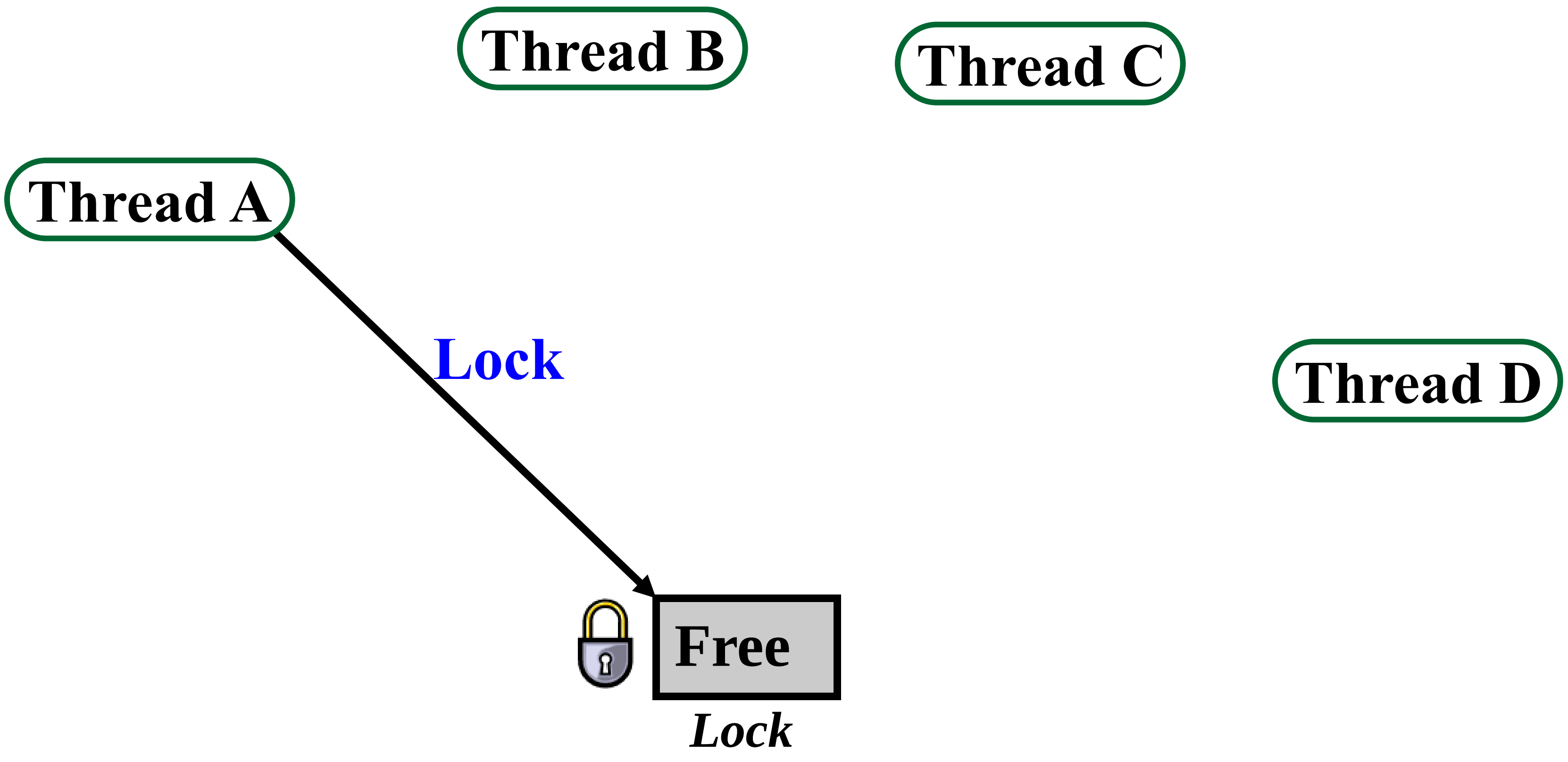
- **What about using *locks* ?**
- **Locks solve the problem of exclusive access to shared data.**
  - Acquiring a lock prevents concurrent access
  - Expresses intention to enter critical section

- ***Assumption:***
  - Each each shared data item has an associated lock
  - Every thread sets the right lock before accessing shared data!
  - Every thread releases the lock after it is done!

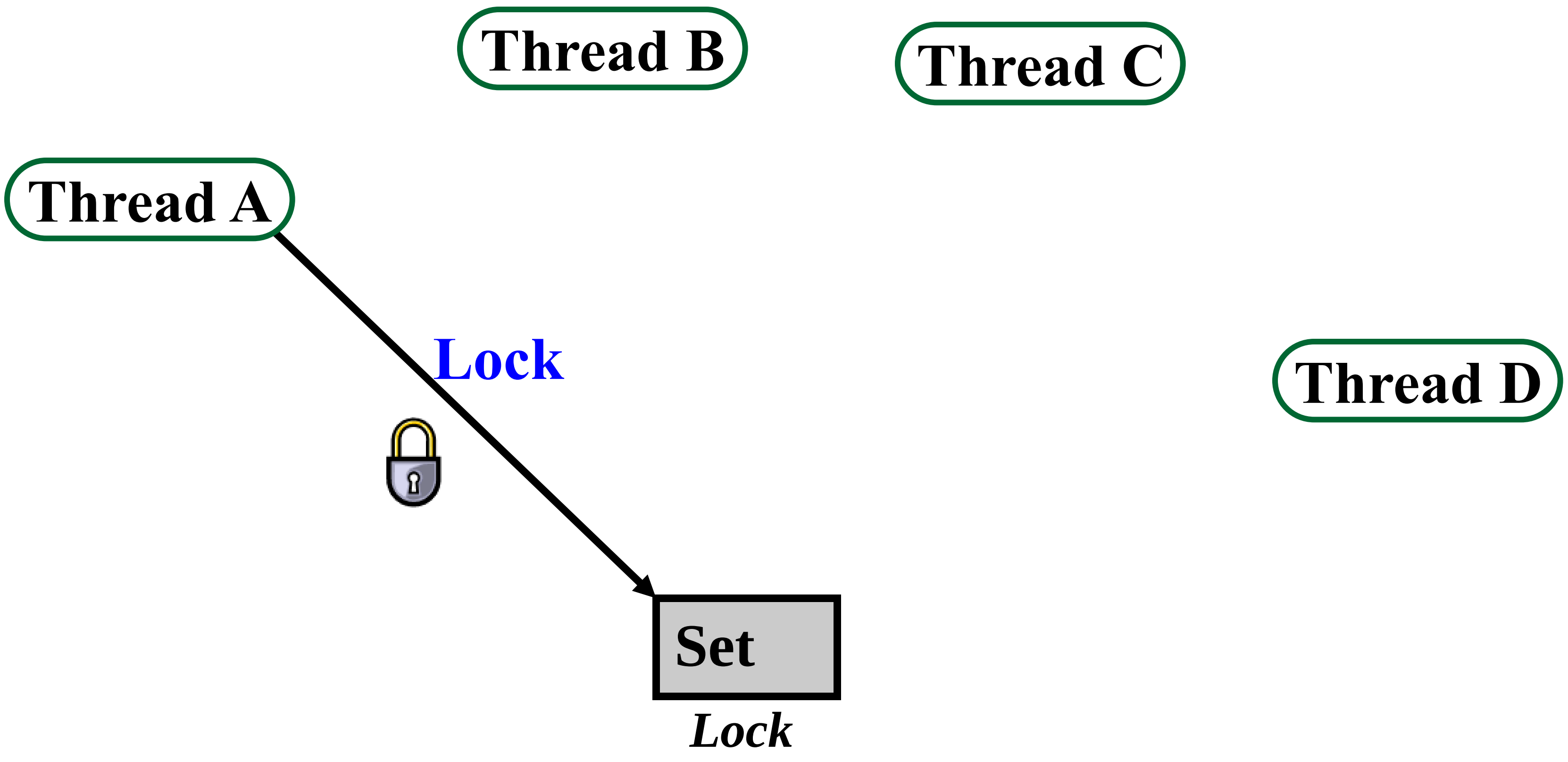
Acquiring and releasing locks



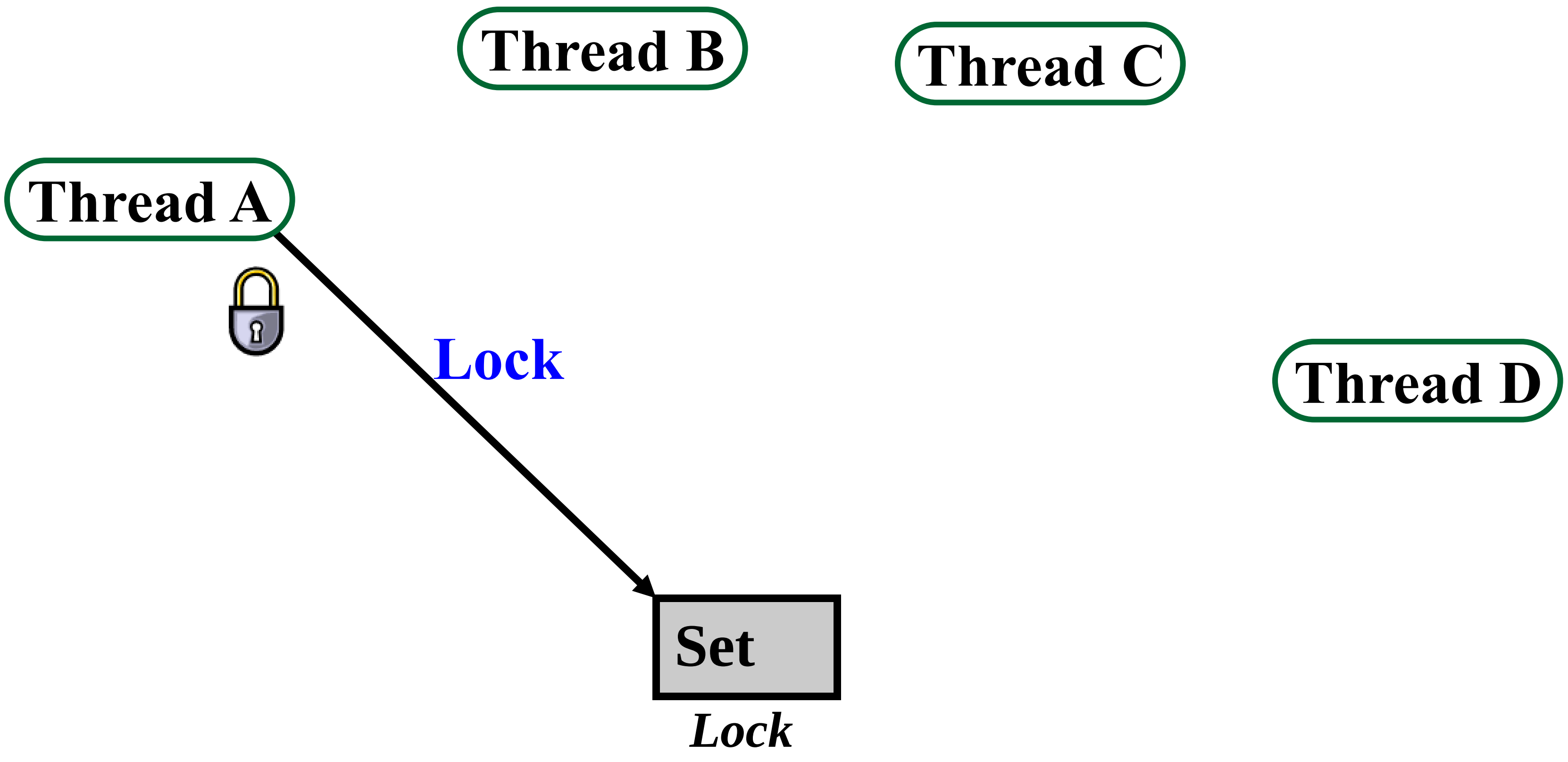
Acquiring and releasing locks



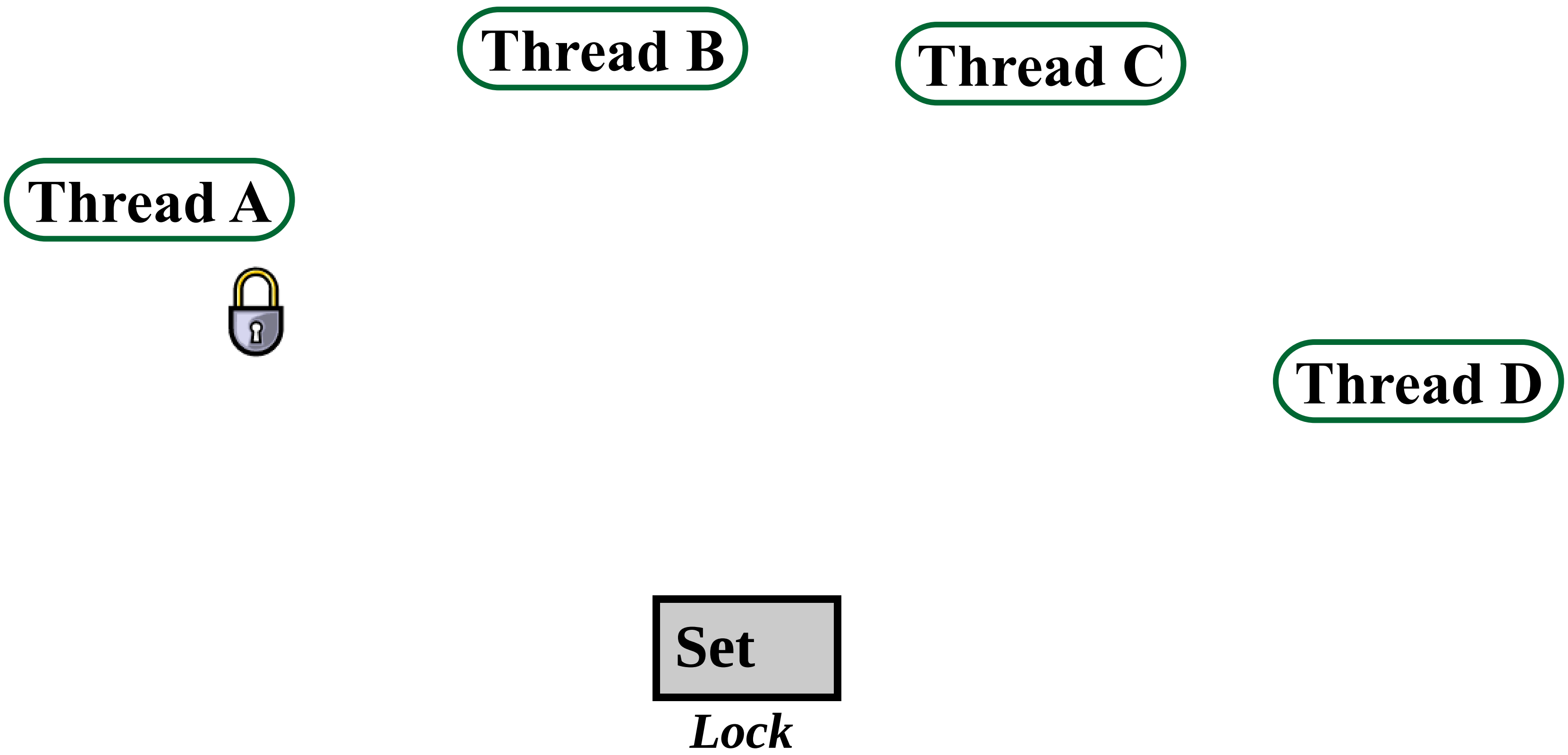
Acquiring and releasing locks



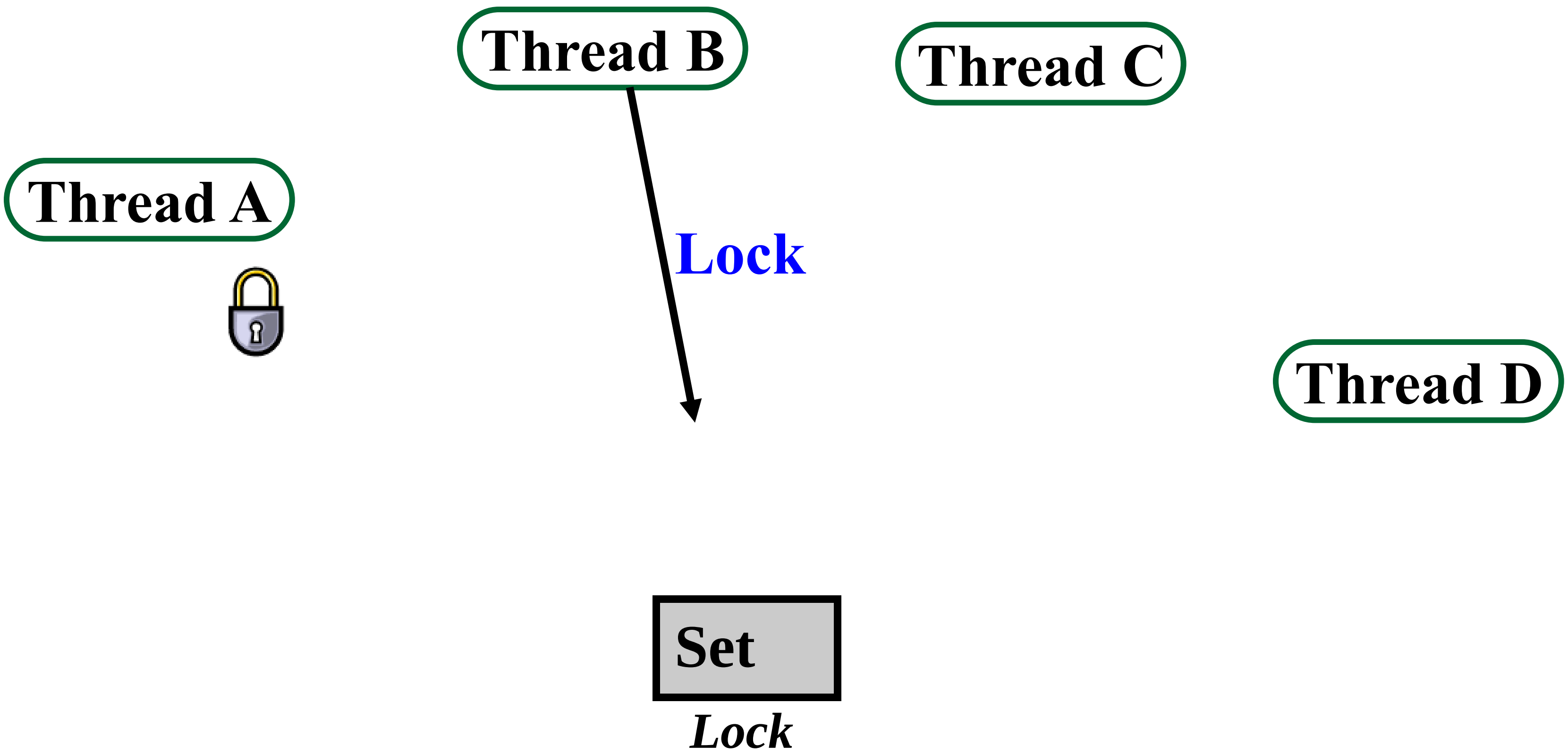
Acquiring and releasing locks



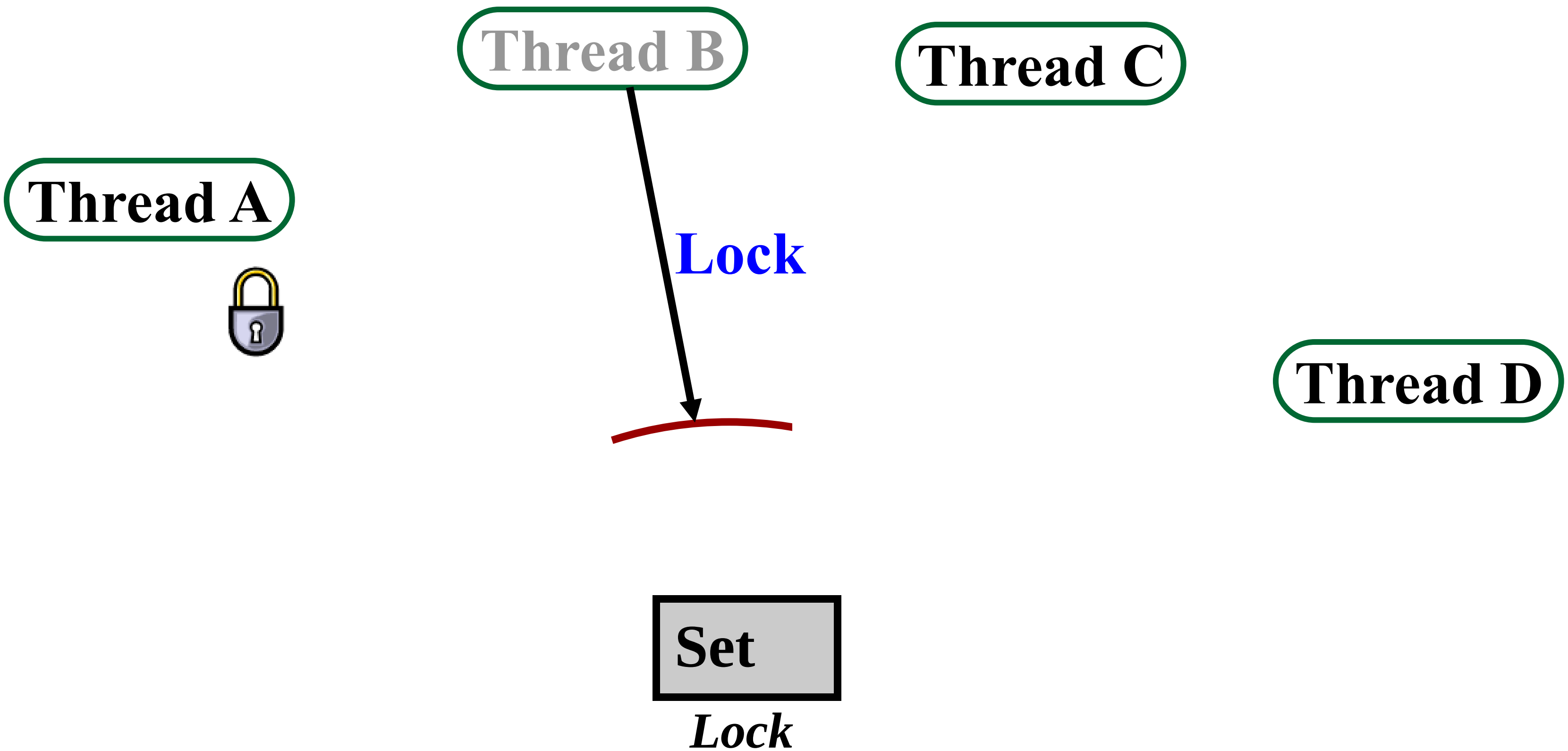
Acquiring and releasing locks



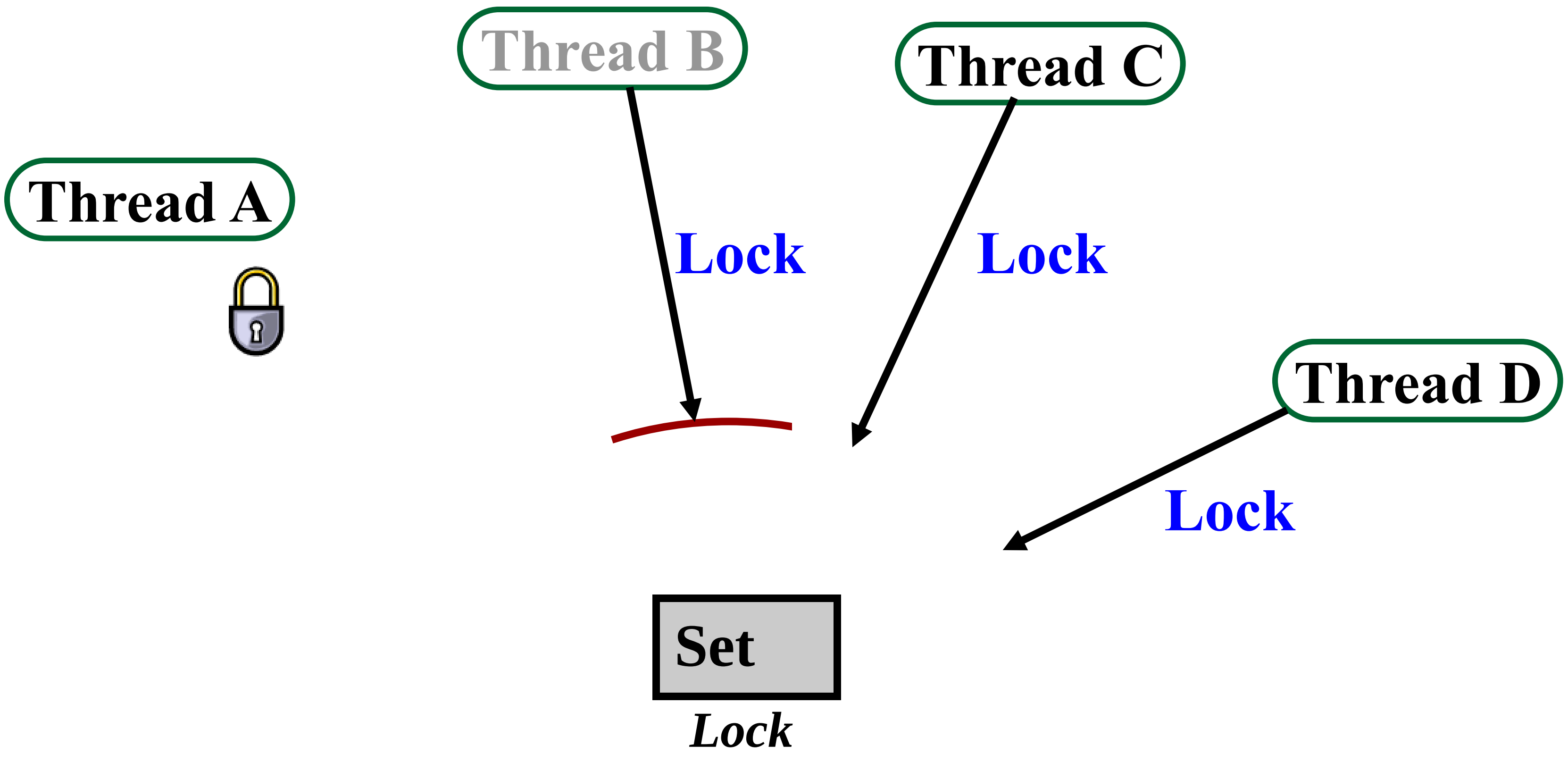
Acquiring and releasing locks



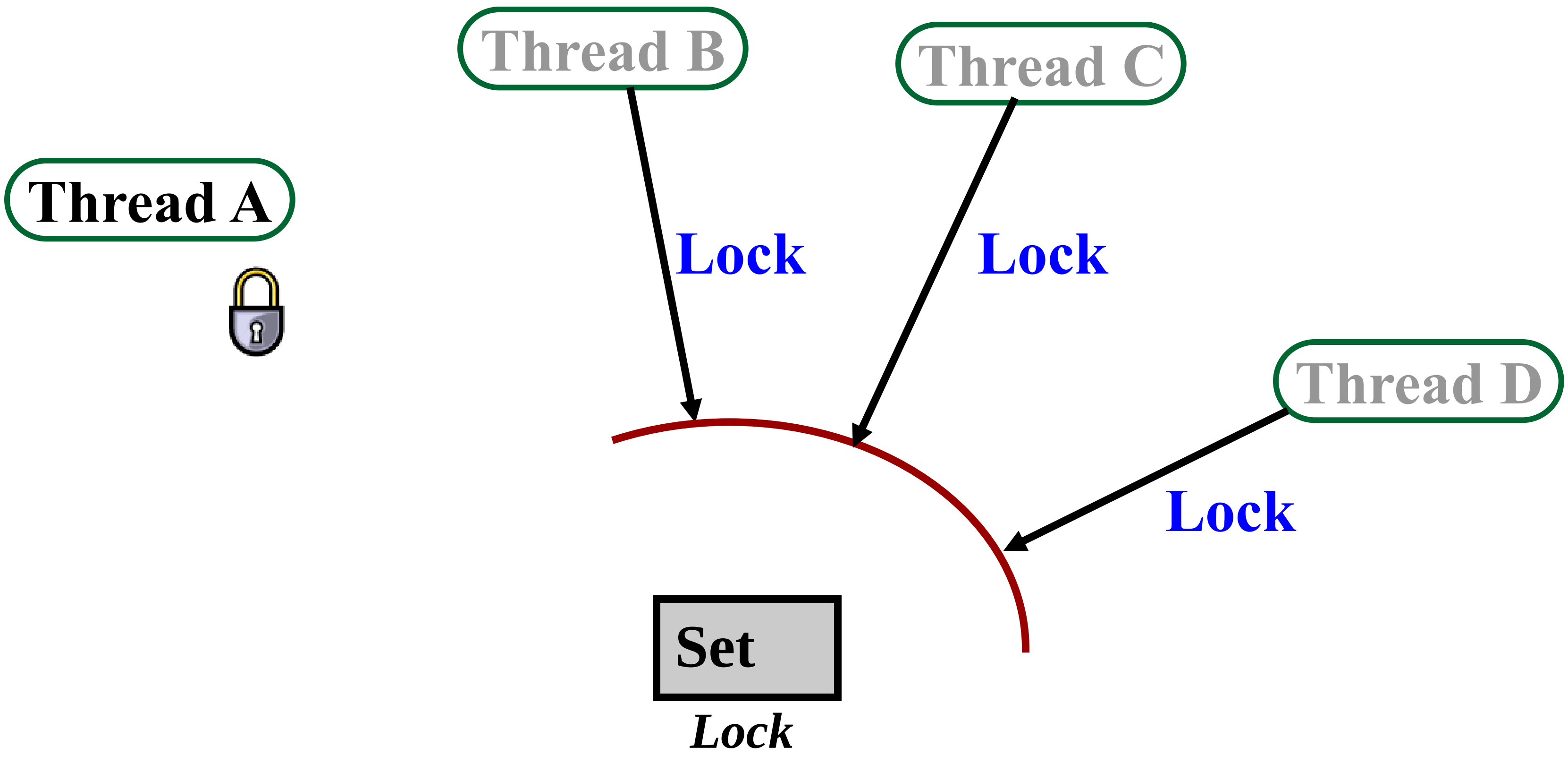
Acquiring and releasing locks



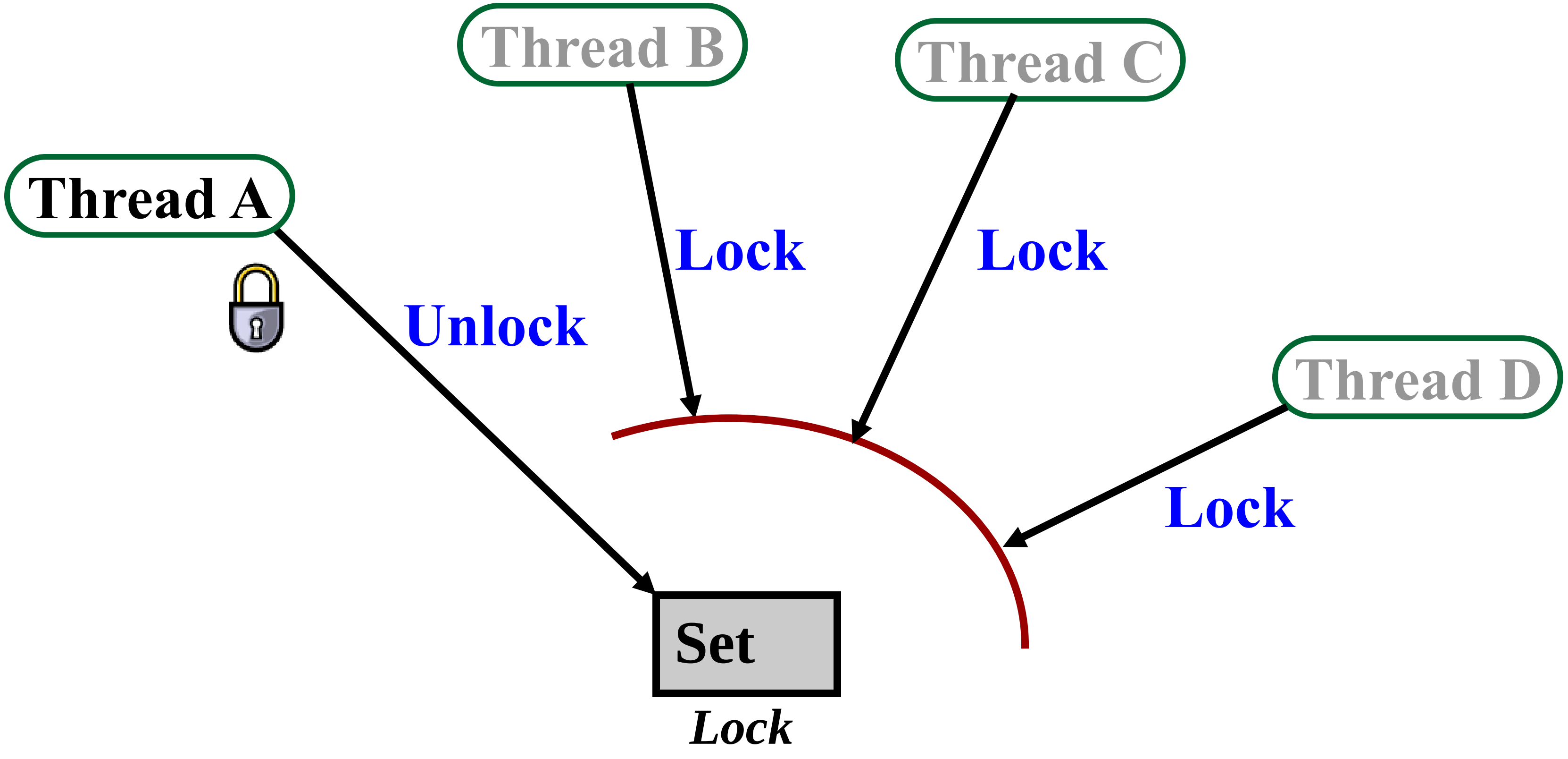
Acquiring and releasing locks



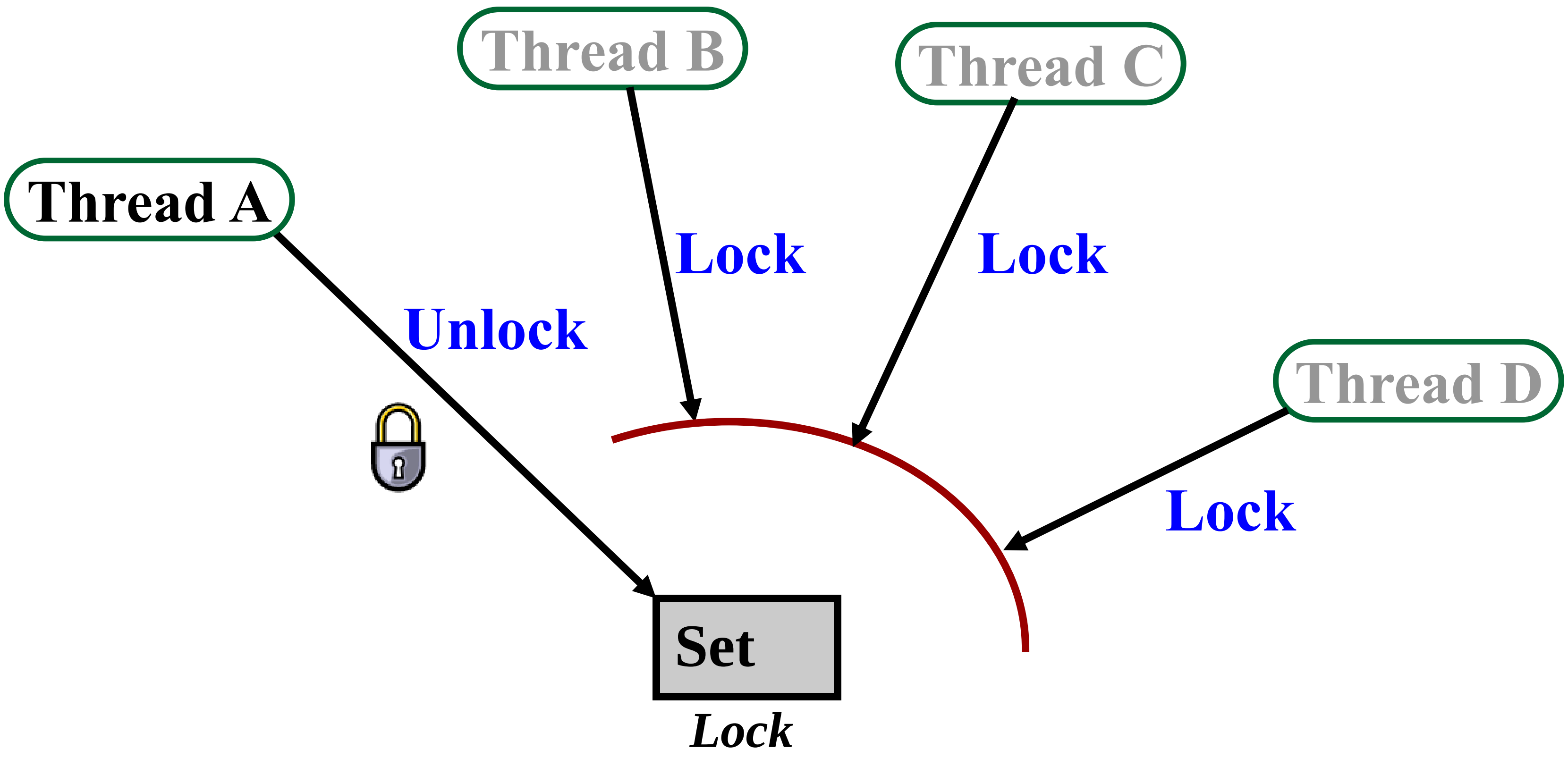
Acquiring and releasing locks



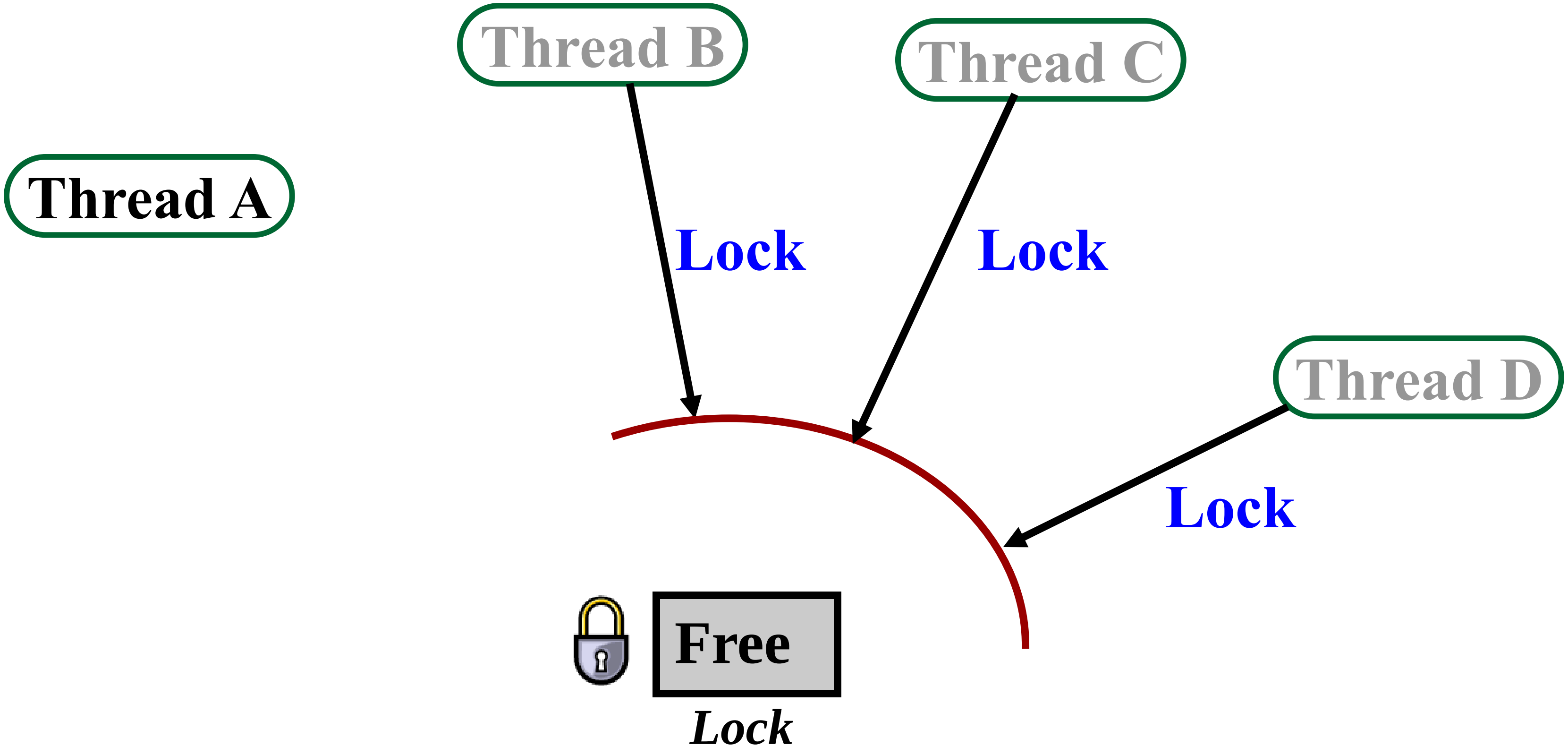
Acquiring and releasing locks



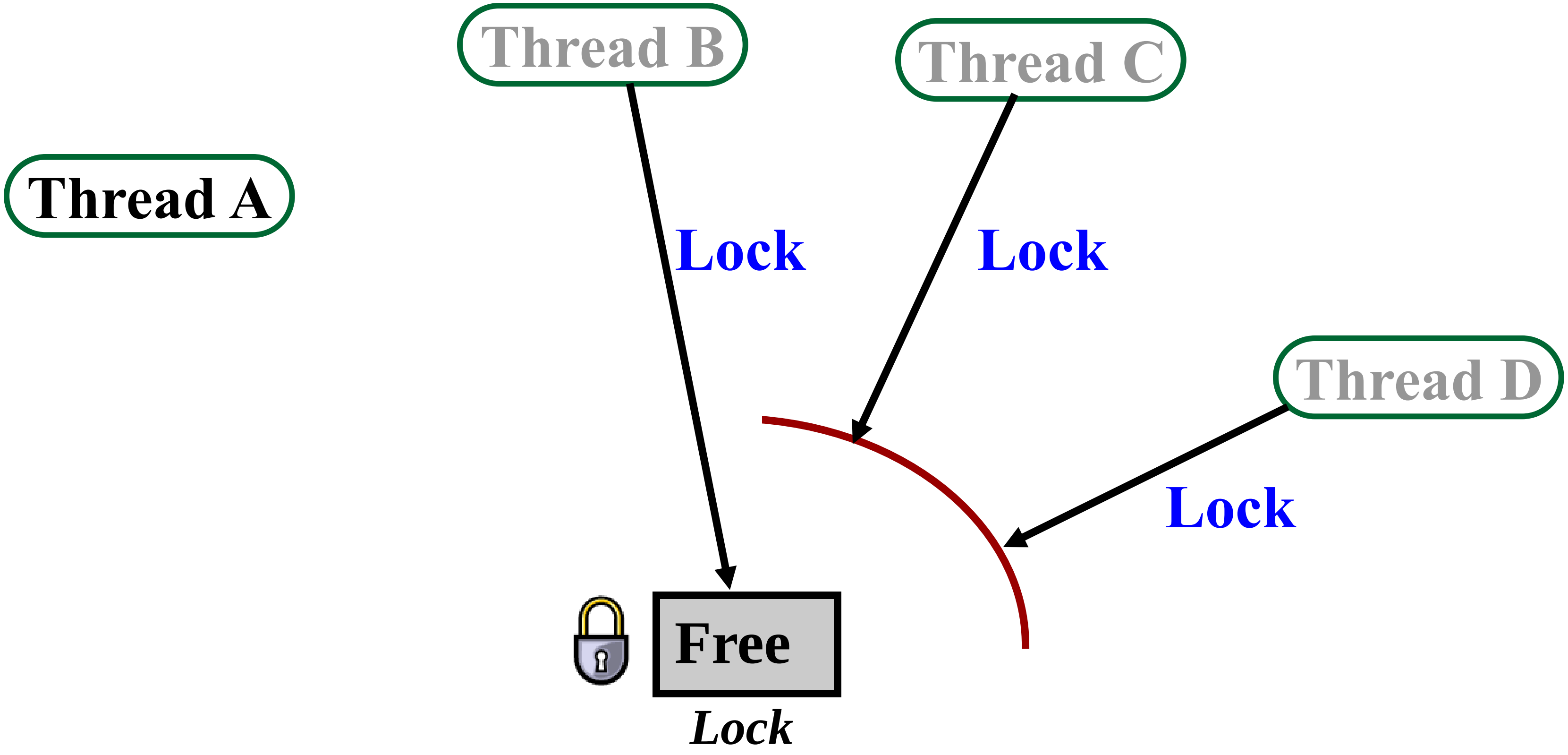
Acquiring and releasing locks



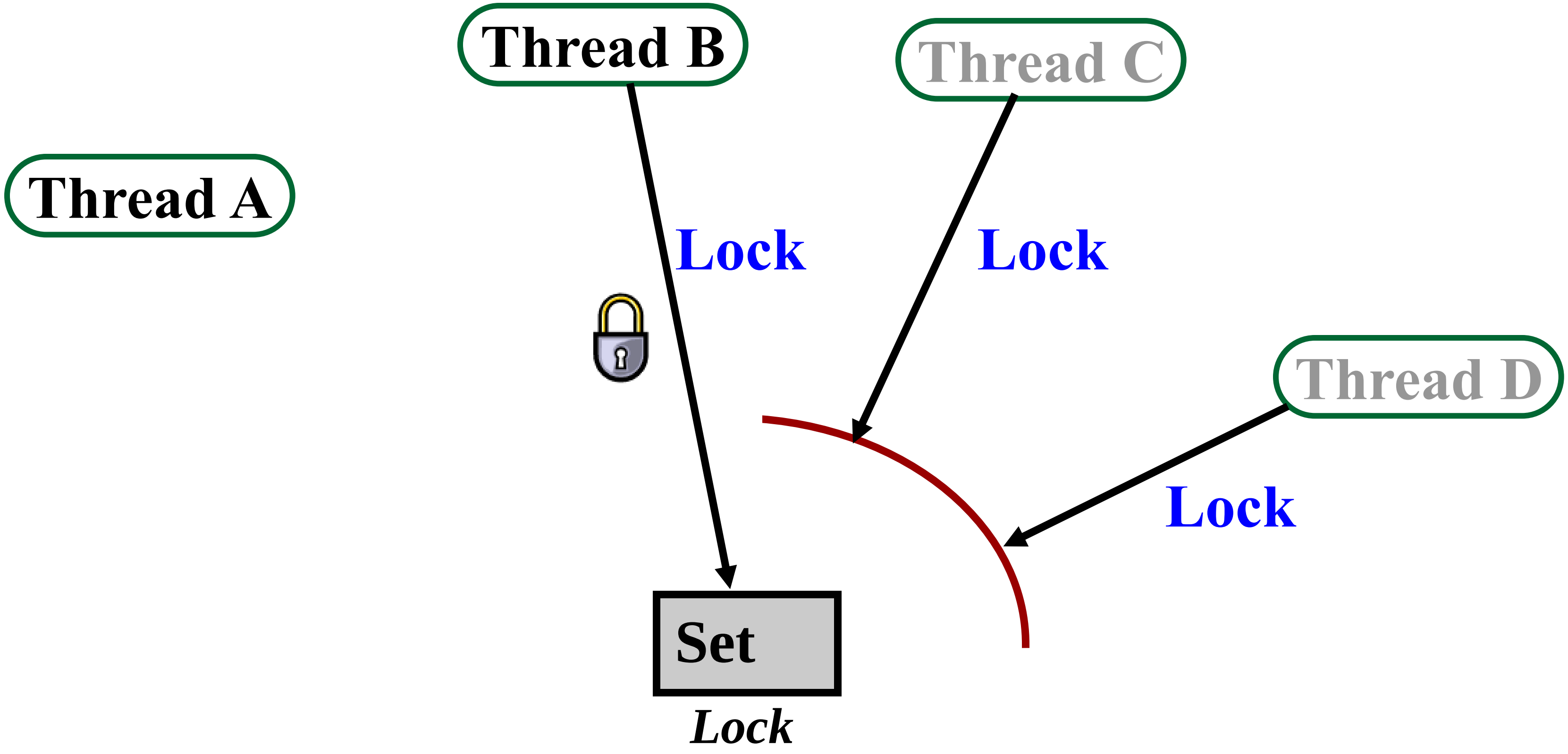
Acquiring and releasing locks



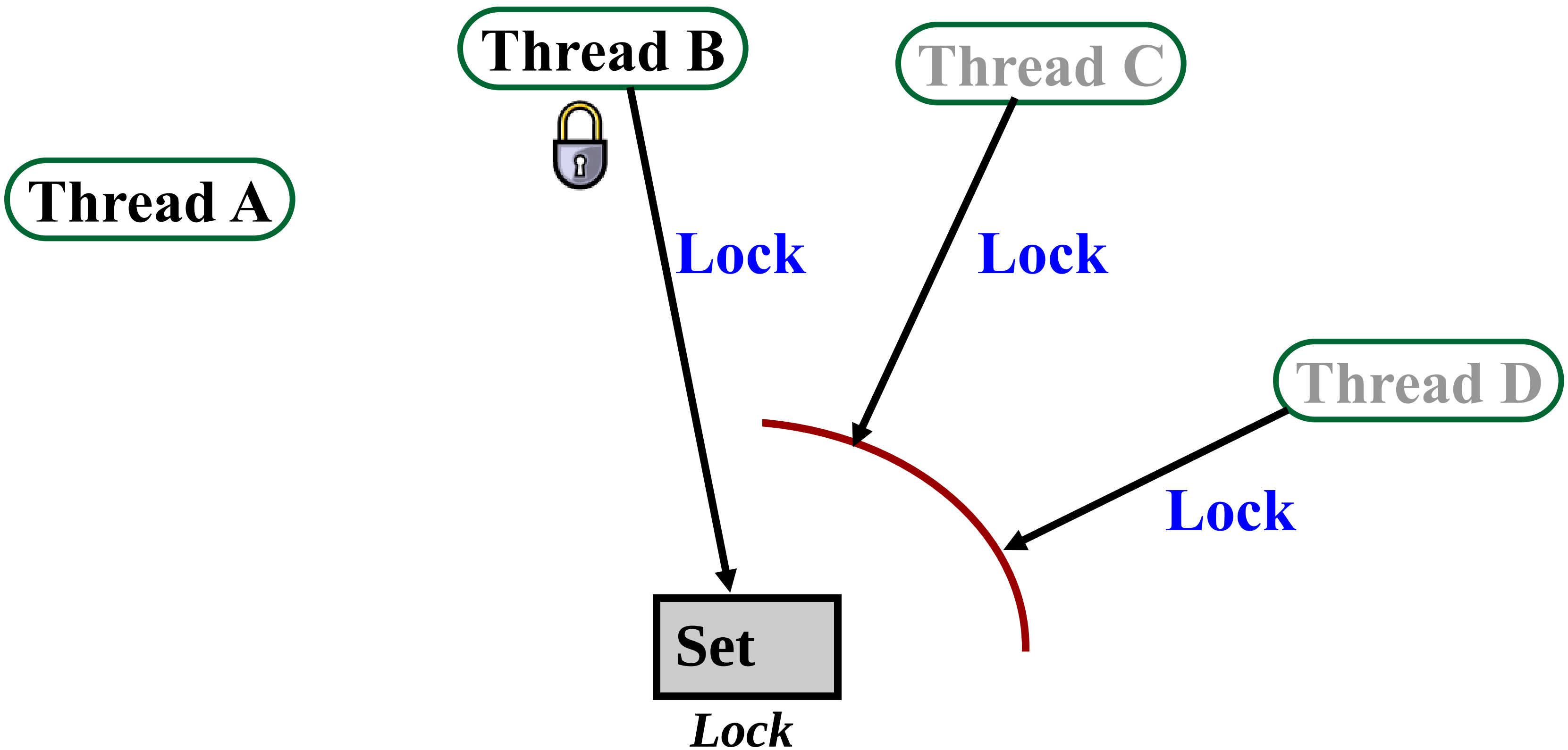
Acquiring and releasing locks



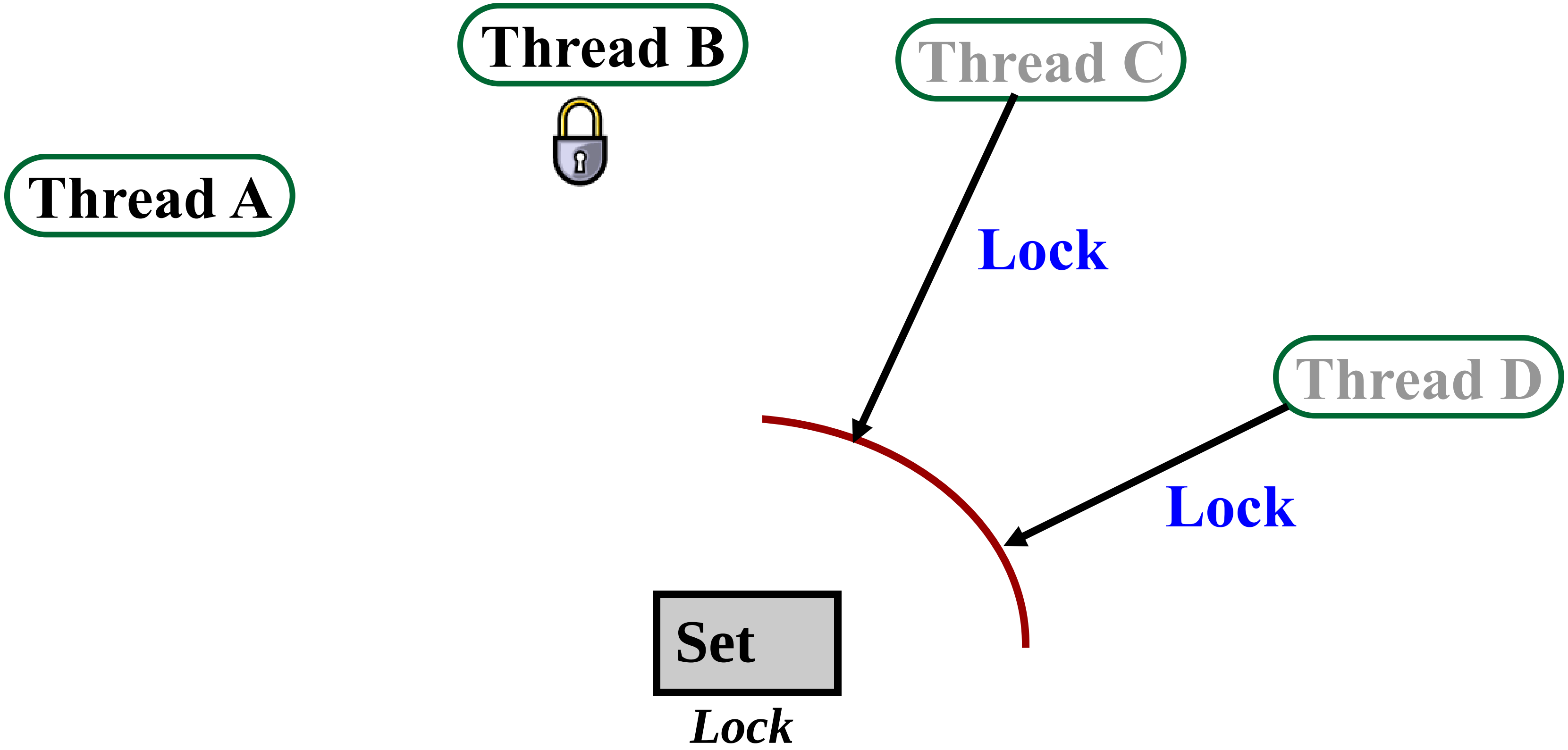
Acquiring and releasing locks



Acquiring and releasing locks



Acquiring and releasing locks



Mutex locks

- **An abstract data type**
- **Used for synchronization and mutual exclusion**
- **The mutex is either:**
  - Locked ("the lock is held")
  - Unlocked ("the lock is free")

Mutex lock operations

- **Lock (*mutex*)**
  - Acquire the lock if it is free
  - Otherwise wait until it can be acquired
- **Unlock (*mutex*)**
  - Release the lock
  - If there are waiting threads wake up one of them

How to use a mutex?

Shared data:

```
Mutex myLock;
```

```
repeat
   Lock(myLock);
   critical section
   Unlock(myLock);
   remainder section
until FALSE
```

```
repeat
   Lock(myLock);
   critical section
   Unlock(myLock);
   remainder section
until FALSE
```

How to implement a mutex?

- **Both Lock and Unlock must be *atomic* !**
  - Does a binary “lock” variable in memory work?
- **Many computers have *some limited* hardware support for setting locks**
  - Atomic Test and Set Lock instruction
  - Atomic compare and swap operation
- **Can be used to implement mutex locks**

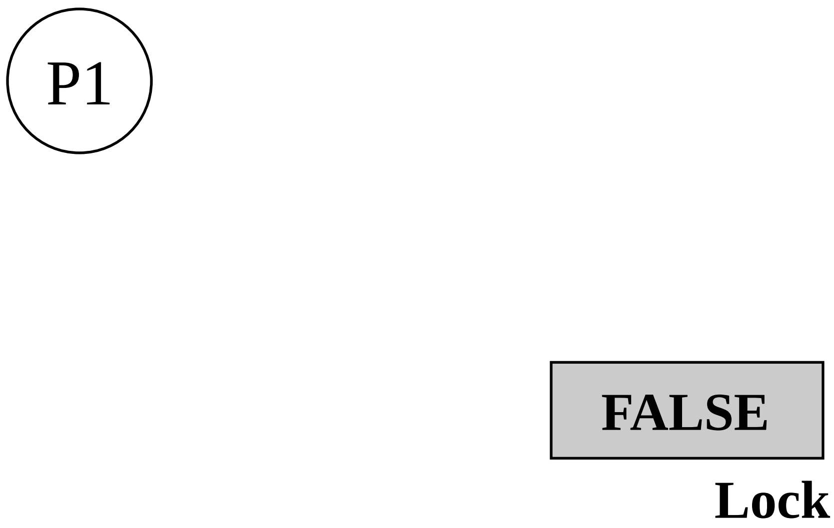
Test-and-set-lock instruction (TSL, tset)

- **A lock is a single word variable with two values**
  - 0 = FALSE = not locked
  - 1 = TRUE = locked

- **Test-and-set does the following *atomically*:**
  - Get the (old) value
  - Set the lock to TRUE
  - Return the old value

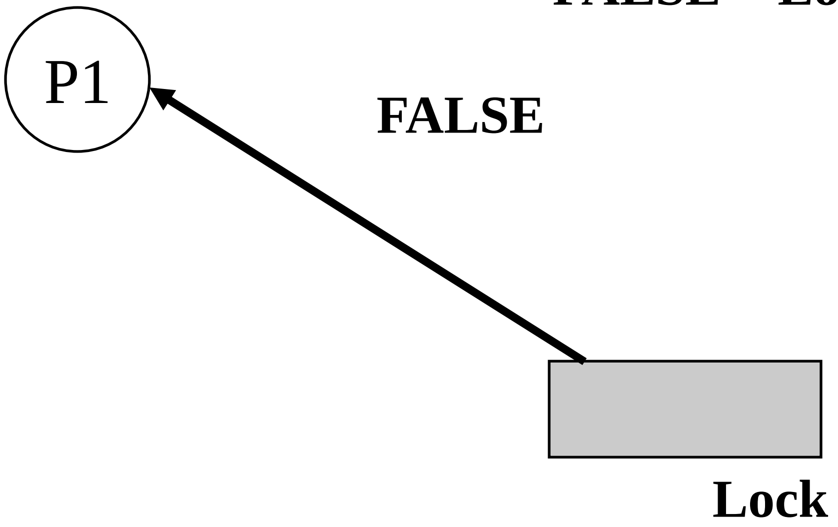
If the returned value was FALSE...
Then you got the lock!!!
If the returned value was TRUE...
Then someone else has the lock
(so try again later)

Test and set lock

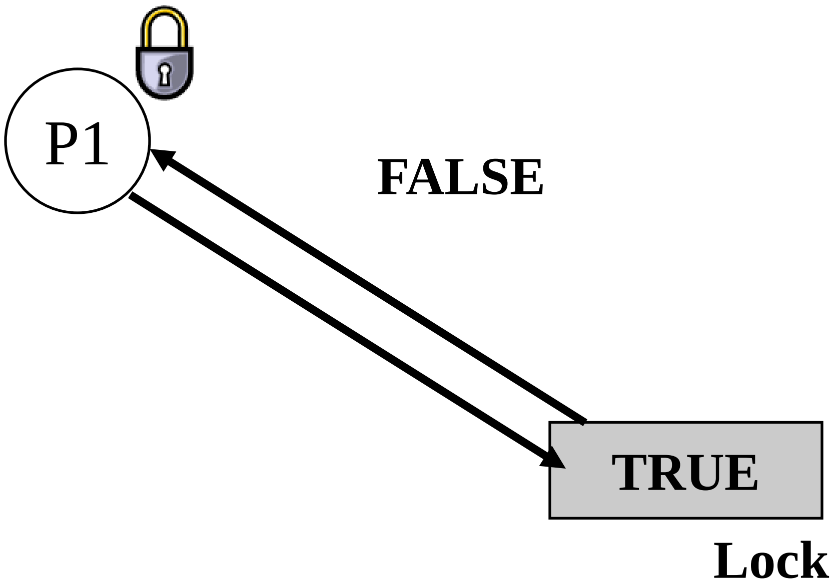


Test and set lock

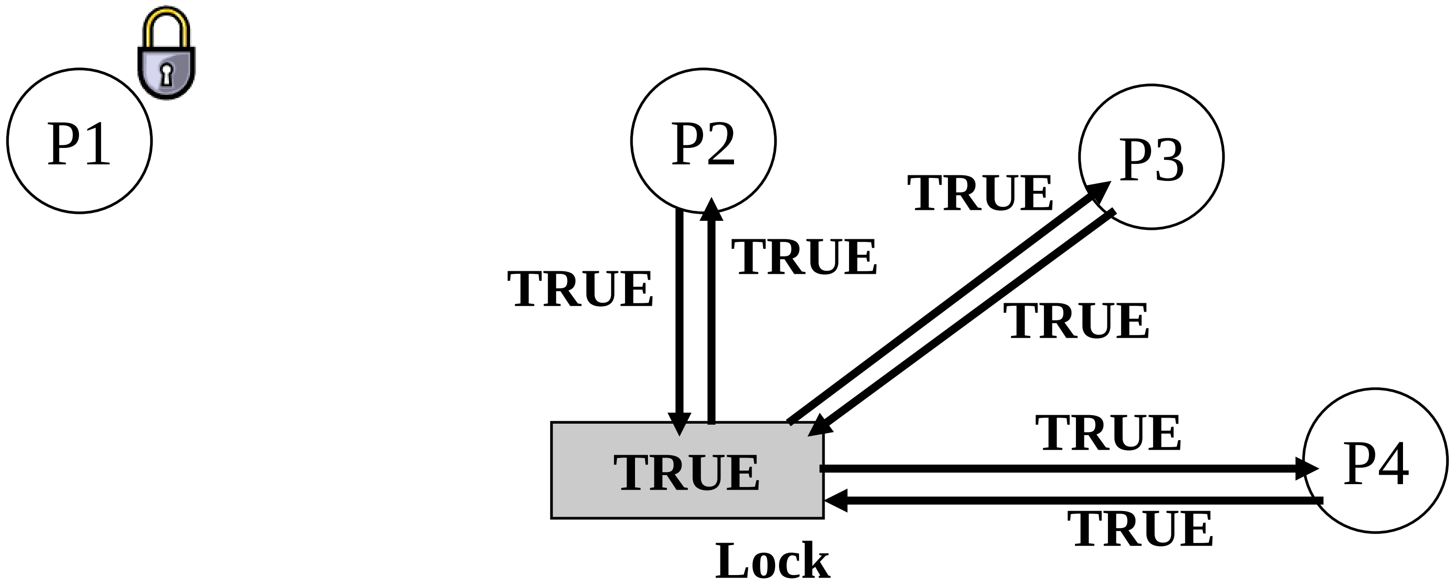
FALSE = Lock Available!!



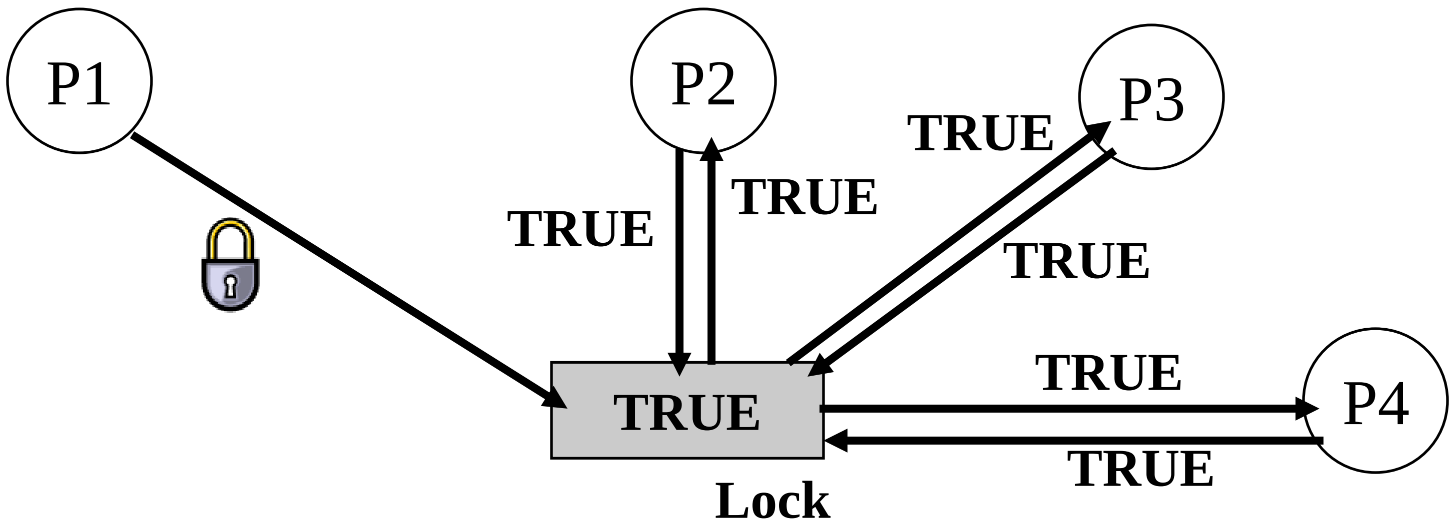
Test and set lock



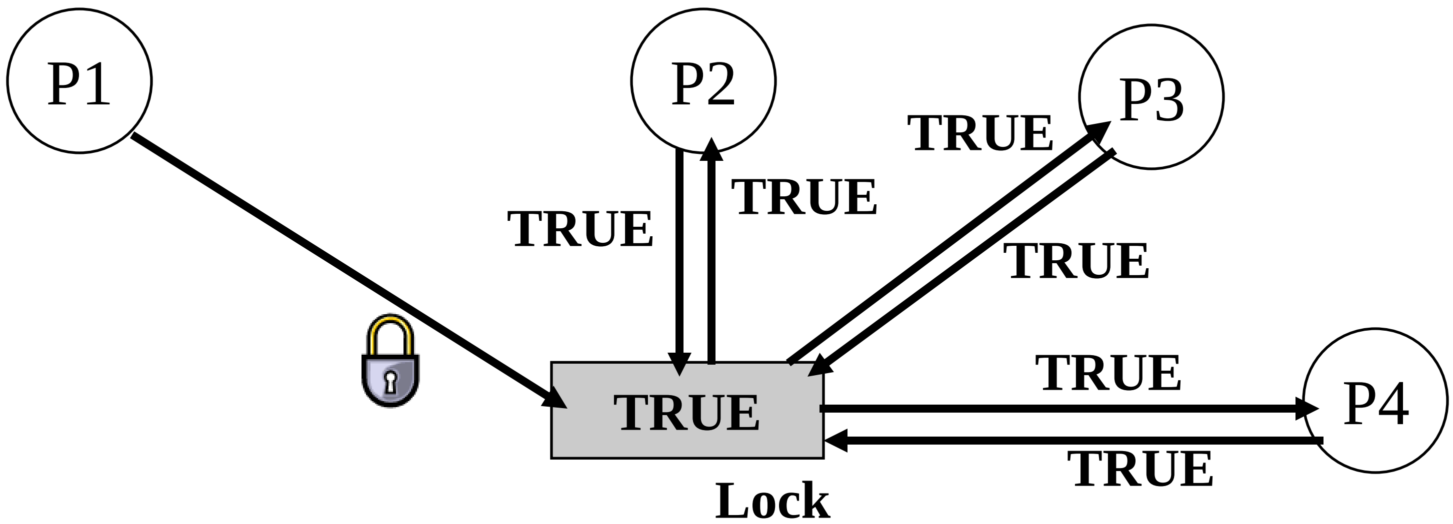
Test and set lock



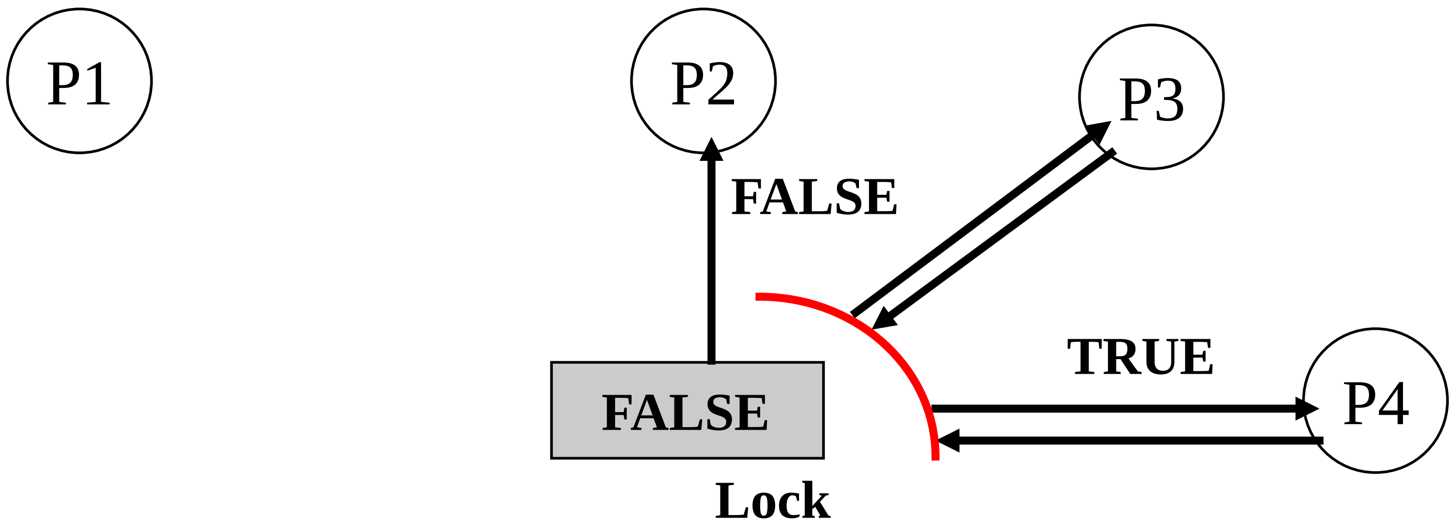
Test and set lock



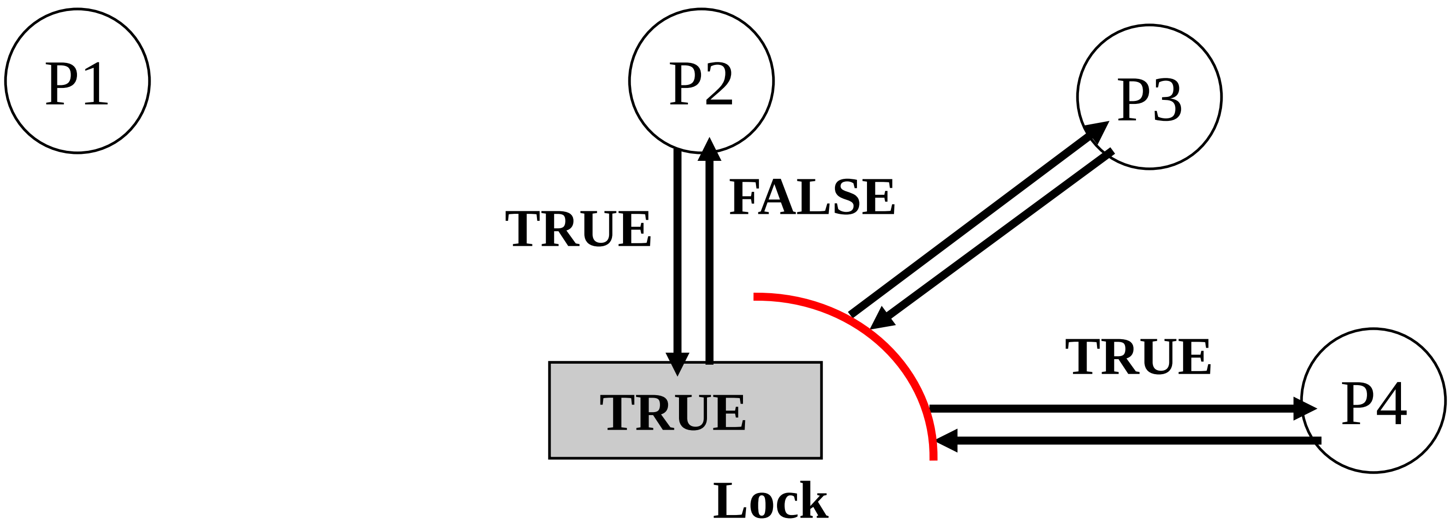
Test and set lock



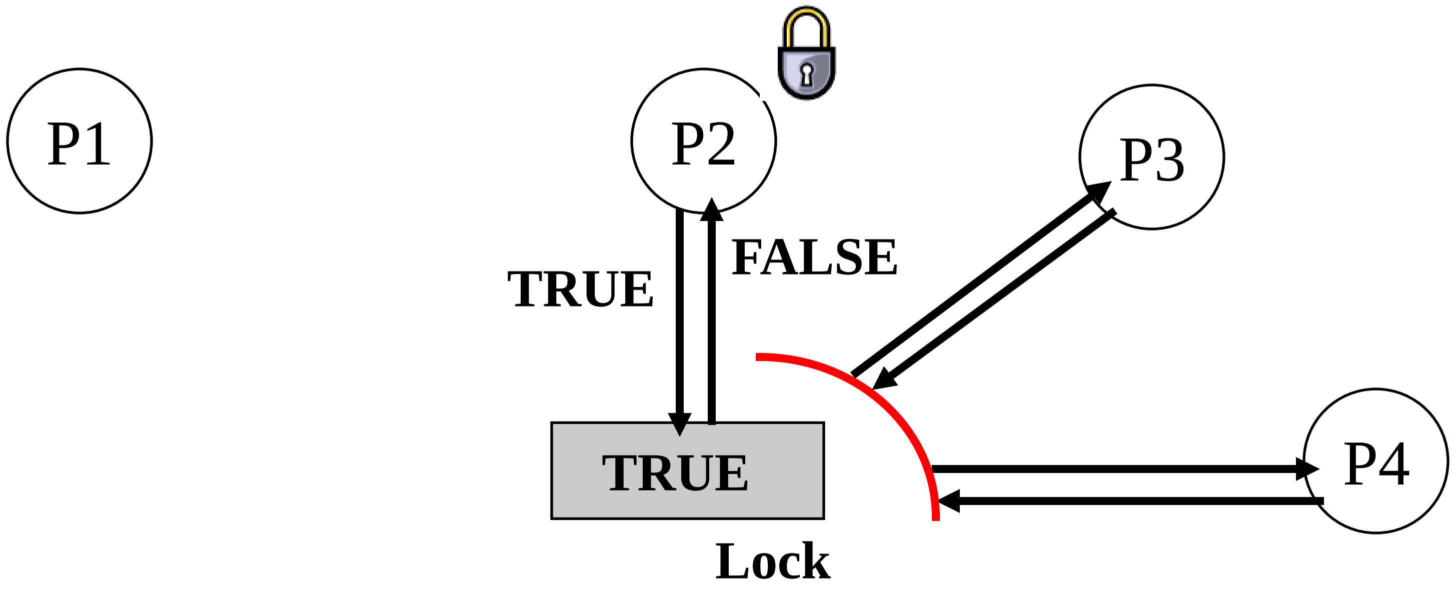
Test and set lock



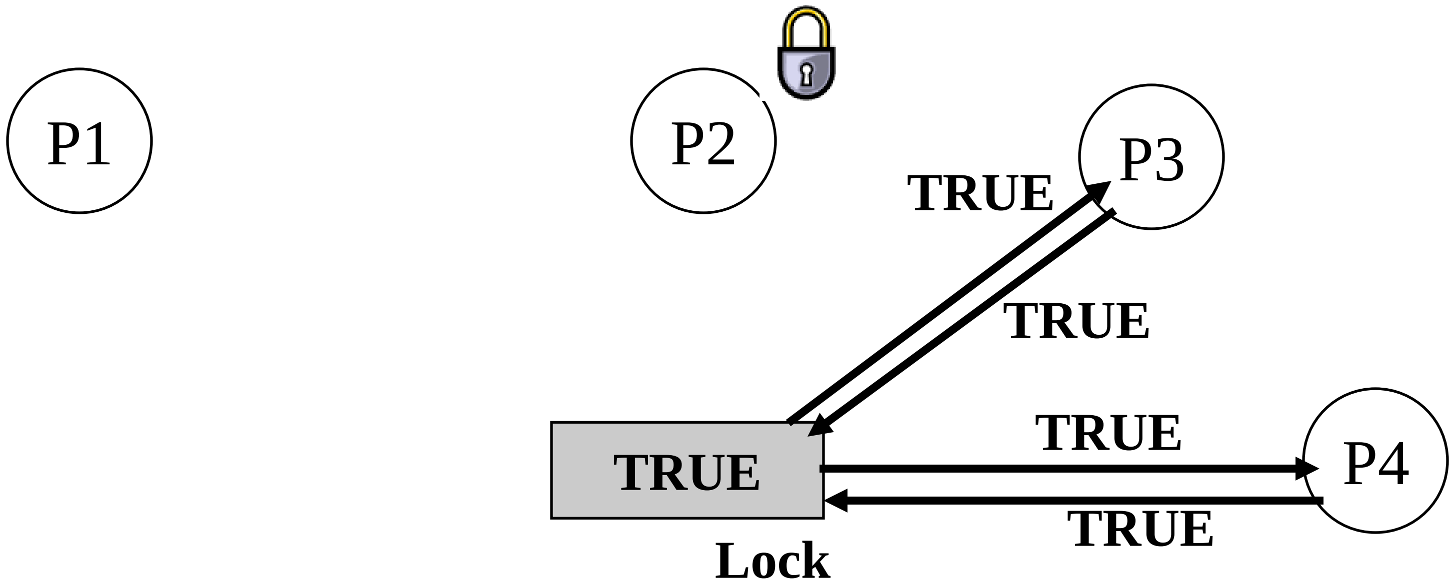
Test and set lock



Test and set lock



Test and set lock



Critical section entry code with TSL

I

```
repeat
  while(TSL(lock))
     no-op;
  critical section
  Lock = FALSE;
  remainder section
until FALSE
```

J

```
repeat
  while(TSL(lock))
     no-op;
  critical section
  Lock = FALSE;
  remainder section
until FALSE
```

- **Guarantees that only one thread at a time will enter its critical section**
- **Note that processes are busy while waiting**
  - Spin locks

Busy waiting

- **Also called polling or spinning**
  - *The thread consumes CPU cycles to evaluate when the lock becomes free !*
- **Shortcoming on a single CPU system...**
  - A busy-waiting thread can prevent the lock holder from running & completing its critical section & releasing the lock!
  - Why not block instead of busy wait ?

Quiz

- **What is the difference between a program and a process?**
- **Is the Operating System a program?**
- **Is the Operating System a process?**
  - Does it have a process control block?
  - How is its state managed when it is not running?
- **What is the difference between processes and threads?**
- **What tasks are involved in switching the CPU from one process to another?**
  - Why is it called a context switch?
- **What tasks are involved in switching the CPU from one thread to another?**
  - Why are threads “lightweight”?