Recap: Thread

- What is it?
  - Independent flow of control
- What does it need (thread private)?
  - Stack
- What for?
  - Lightweight programming construct for concurrent activities
- How to implement?
  - Kernel thread vs. user thread

Recap: Process vs. Thread

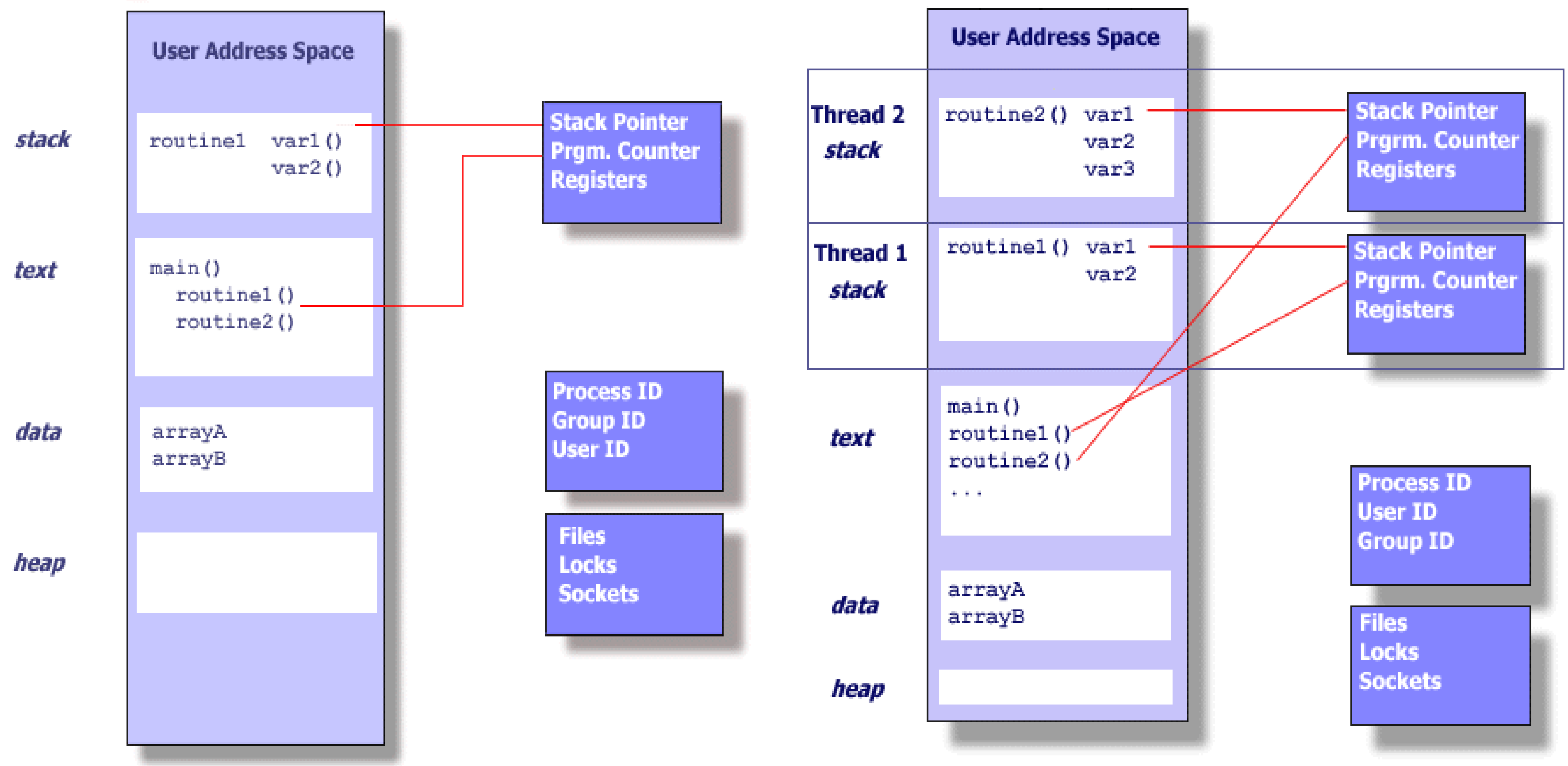


Figure source: https://computing.llnl.gov/tutorials/pthreads/

Multi-threads vs. Multi-processes

- Multi-processes
  - (+) protection
  - (-) performance (?)

Process-per-tab

- Multi-threads
  - (+) performance
  - (-) protection

Single-process multi-threads

Threads: Advanced Topics

- Semantics of Fork/exec()
- Signal handling
- Thread pool
- Multicore

Semantics of fork()/exec()

- Remember fork(), exec() system calls?
  - Fork: create a child process (a copy of the parent)
  - Exec: replace the address space with a new pgm.

- Duplicate *all* threads or *the caller* only?
  - Linux: the calling thread only
  - Complicated. Don't do it!
    - Why? Mutex states, library, …
    - Exec() immediately after Fork() may be okay.

Signal Handling

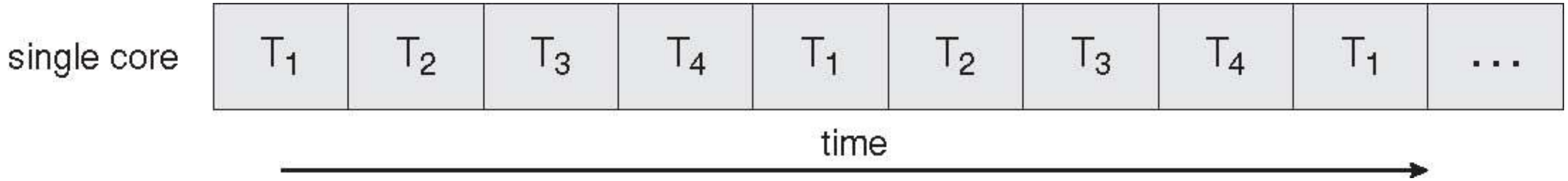
- What is *Singal*?
  - $ man 7 signal
  - OS to process notification
    - “hey, wake-up, you’ve got a packet on your socket,”
    - “hey, wake-up, your timer is just expired.”
- Which *thread* to deliver a signal?
  - Any thread
    - e.g., kill(pid)
  - Specific thread
    - E.g., pthread_kill(tid)

Thread Pool

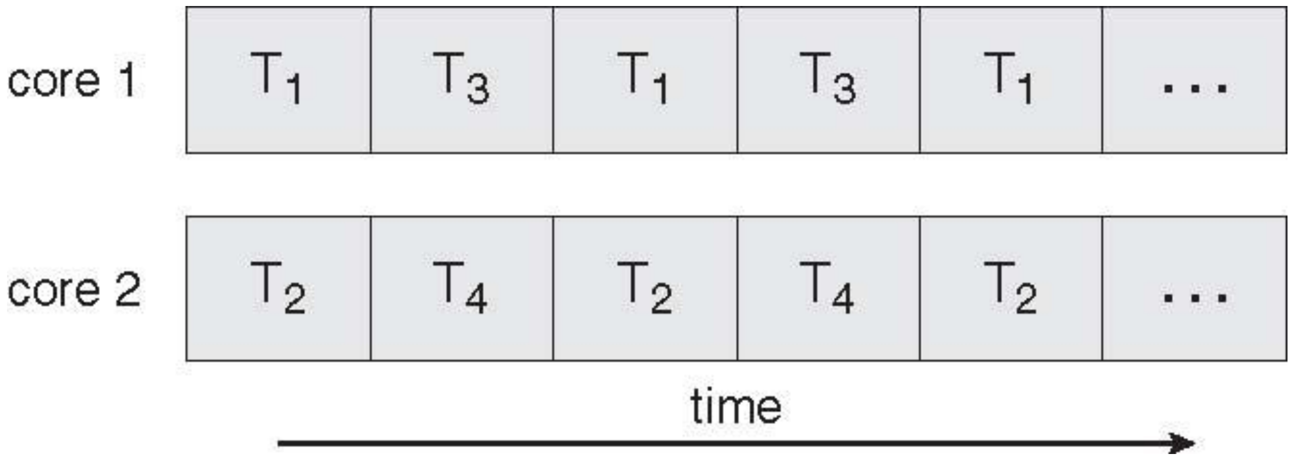
- Managing threads yourself can be cumbersome and costly
  - Repeat: create/destroy threads as needed.

- Let’s create a set of threads ahead of time, and just ask them to execute my functions
  - #of thread ~ #of cores
  - No need to create/destroy many times
  - Many high-level parallel libraries use this.
    - e.g., Intel TBB (threading building block), …

Single Core Vs. Multicore Execution



Single core execution



Multiple core execution

Challenges for Multithreaded Programming in Multicore

- How to divide activities?
- How to divide data?
- **How to synchronize accesses to the shared data?**
- How to test and debug?

EECS750

Summary

- Thread
  - What is it?
    - Independent flow of control.
  - What for?
    - Lightweight programming construct for concurrent activities
  - How to implement?
    - Kernel thread vs. user thread
- Next class
  - How to synchronize?

Synchronization

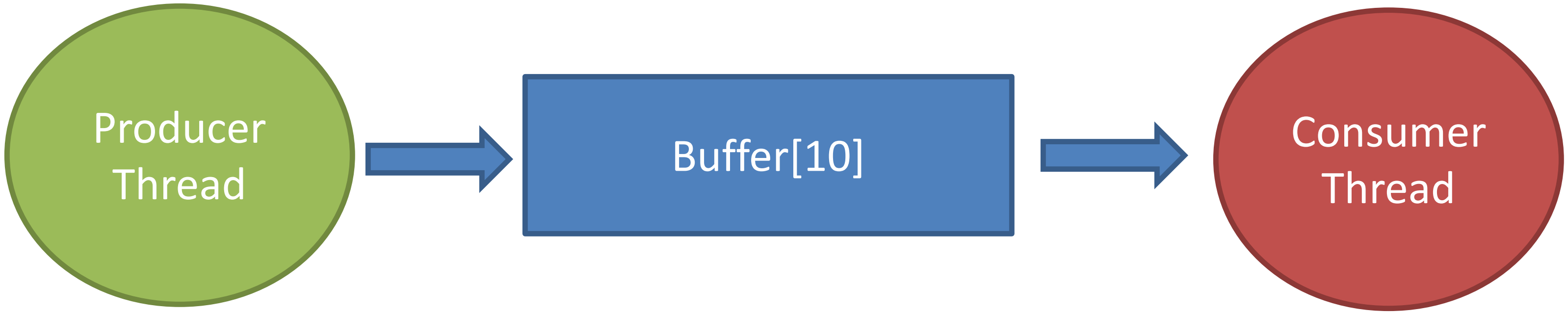
Disclaimer: some slides are adopted from the book authors' slides with permission

Agenda

- **Mutual exclusion**
  - **Peterson's algorithm (Software)**
  - **Synchronization instructions (Hardware)**

- High-level synchronization mechanisms
  - Mutex
  - Semaphore
  - Monitor

Producer/Consumer



Producer/Consumer

Producer

```
while (true){

   /* wait if buffer full */
   while (counter == 10);

   /* produce data */
   buffer[in] = sdata;
   in = (in + 1) % 10;

   /* update number of
      items in buffer */
   counter++;
}
```

Consumer

```
while (true){

   /* wait if buffer empty */
   while (counter == 0);

   /* consume data */
   sdata = buffer[out];
   out = (out + 1) % 10;

   /* update number of
      items in buffer */
   counter--;
}
```

Producer/Consumer

Producer

```
while (true){

   /* wait if buffer full */
   while (counter == 10);

   /* produce data */
   buffer[in] = sdata;
   in = (in + 1) % 10;

   /* update number of
      items in buffer */
   R1 = load (counter);
   R1 = R1 + 1;
   counter = store (R1);
}
```

Consumer

```
while (true){

   /* wait if buffer empty */
   while (counter == 0);

   /* consume data */
   sdata = buffer[out];
   out = (out + 1) % 10;

   /* update number of
      items in buffer */
   R2 = load (counter);
   R2 = R2 – 1;
   counter = store (R2);
}
```

Check Yourself

```
int count = 0;
int main()
{
	count = count + 1;
	return count;
}

$ gcc –O2 –S sync.c
```

```
…
movl    count(%rip), %eax
addl    $1, %eax
movl    %eax, count(%rip)
…
```

Race Condition

Initial condition: *counter = 5*

Thread 1

R1 = load (counter);
R1 = R1 + 1;
counter = store (R1);

Thread 2

R2 = load (counter);
R2 = R2 – 1;
counter = store (R2);

- What are the possible outcome?

Race Condition

Initial condition: *counter = 5*

R1 = load (counter);	R1 = load (counter);	R1 = load (counter);
R1 = R1 + 1;	R1 = R1 + 1;	R1 = R1 + 1;
counter = store (R1);	R2 = load (counter);	R2 = load (counter);
R2 = load (counter);	R2 = R2 – 1;	R2 = R2 – 1;
R2 = R2 – 1;	counter = store (R1);	counter = store (R2);
counter = store (R2);	counter = store (R2);	counter = store (R1);
counter = 5	*counter = 4*	*counter = 6*

- Why this happens?

Race Condition

- A situation when two or more threads **read and write** shared data at the same time
- Correctness depends on the execution order

Thread 1

R1 = load (**counter**);
R1 = R1 + 1;
counter = store (R1);

read

write

Thread 2

R2 = load (**counter**);
R2 = R2 – 1;
counter = store (R2);

- How to prevent race conditions?

Critical Section

- Code sections of potential race conditions

Thread 1	Thread 2
Do something	Do something
..	..
R1 = load (**counter**);	R2 = load (**counter**);
R1 = R1 + 1;	R2 = R2 – 1;
counter = store (R1);	**counter** = store (R2);
...	..
Do something	Do something

Critical sections

Solution Requirements

- Mutual Exclusion
  - If a thread executes its critical section, *no other threads* can enter their critical sections
- Progress
  - If no one executes a critical section, someone can enter its critical section
- Bounded waiting
  - Waiting (time/number) must be bounded

Simple Solution (?): Use a Flag

```
// wait
while (in_cs)
    ;
// enter critical section
in_cs = true;

Do something

// exit critical section
in_cs = false;
```

T1

```
while(in_cs);


in_cs = true;
//enter
```

T2

```

while(in_cs);
in_cs = true;

//enter
```

- Mutual exclusion is not guaranteed

KU THE UNIVERSITY OF KANSAS

Peterson's Solution

- Software solution (no h/w support)
- Two process solution
  - Multi-process extension exists
- The two processes share two variables:
  - int turn;
    - The variable turn indicates whose turn it is to enter the critical section
  - Boolean flag[2]
    - The flag array is used to indicate if a process is ready to enter the critical section.

Peterson's Solution

Thread 1

```
do {
    flag[0] = TRUE;
    turn = 1;
    while (flag[1]  &&  turn==1)
        ;
    // critical section

    flag[0] = FALSE;

    // remainder section
} while (TRUE)
```

Thread 2

```
do {
    flag[1] = TRUE;
    turn = 0;
    while (flag[0]  &&  turn==0)
        ;
    // critical section

    flag[1] = FALSE;

    // remainder section
} while (TRUE)
```

- Solution meets all three requirements
  - Mutual exclusion: P0 and P1 cannot be in the critical section at the same time
  - Progress: if P0 does not want to enter critical region, P1 does no waiting
  - Bounded waiting: process waits for at most one turn

Peterson's Solution

- Only supports two processes
  - generalizing for more than two processes has been achieved, but not very efficient
- Assumes that the LOAD and STORE instructions are atomic
- Assumes that memory accesses are not reordered
  - your compiler re-orders instructions (gcc –O2, -O3, …)
  - your processor re-orders instructions (memory consistency models)

Reordering by the CPU

Initially X = Y = 0

Thread 0	Thread 1
X = 1	Y = 1
R1 = Y	R2 = X

Thread 0	Thread 1
R1 = Y	
	R2 = X
X = 1	
	Y = 1

- Possible values of R1 and R2?
  - 0,1
  - 1,0
  - 1,1
  - **0,0 ← possible on PC**

KU THE UNIVERSITY OF KANSAS

Lock-Based Solutions

- General solution to the critical section problem
  - critical sections are protected by locks
  - process must acquire lock before entry
  - process releases lock on exit

```
do {
    acquire lock;

        critical section

    release lock;

        remainder section

} while(TRUE);
```

Hardware Support for Lock-Based Solutions – Uniprocessors

- For uniprocessor systems
  - concurrent processes cannot be overlapped, only *interleaved*
  - process runs until it is *interrupted*
- Disable interrupts !
  - active process will run without preemption

```
do {

    disable interrupts;
        critical section
    enable interrupts;

        remainder section
} while(TRUE);
```

THE UNIVERSITY OF KANSAS

Hardware Support for Lock-Based Solutions – Multiprocessors

- In multiprocessors
  - several processes share memory
  - processors behave independently in a peer manner
- Disabling interrupt based solution will not work
  - too inefficient
  - OS using this not broadly scalable
- Provide hardware support in the form of atomic instructions
  - atomic *test-and-set* instruction
  - atomic *compare-and-swap* instruction
- Atomic execution of a set of instructions means that the instructions are treated as a single step that cannot be interrupted.
  - **All or nothing** property

TestAndSet Instruction

- Pseudo code definition of *TestAndSet*

```
boolean TestAndSet (boolean *target)
{
    boolean rv = *target;
    *target = TRUE;
    return rv:
}
```

Mutual Exclusion using *TestAndSet*

```
int mutex;
init_lock (&mutex);

do {

    lock (&mutex);
        critical section
    unlock (&mutex);

        remainder section
} while(TRUE);
```

```
void init_lock (int *mutex)
{
    *mutex = 0;
}

void lock (int *mutex)
{
    while(TestAndSet(mutex))
        ;
}

void unlock (int *mutex)
{
    *mutex = 0;
}
```

CAS (Compare & Swap) Instruction

- Psuedo code definition of CAS instruction

```
int CAS(int *value, int oldval, int newval)
{
        int temp = *value;
        if (*value == oldval)
          *value = newval;
        return temp;
 }
```

Mutual Exclusion using *CAS*

```
int mutex;
init_lock (&mutex);

do {

    lock (&mutex);
        critical section
    unlock (&mutex);

        remainder section
} while(TRUE);
```

```
void init_lock (int *mutex) {
    *mutex = 0;
}

void lock (int *mutex) {
    while(CAS(&mutex, 0, 1) != 0);
}

void unlock (int *mutex) {
    *mutex = 0;
}
```

Fairness not guaranteed by any implementation !

Preemptive Vs. Non-preemptive Kernels

- Kernel is full of important shared data
  - structures for maintaining file systems, memory allocation, interrupt handling, etc.
- How to ensure the OS is free from race conditions ?
- Non–preemptive kernels
  - process executing in kernel mode cannot be preempted
  - disable interrupts when process is in kernel mode
  - what about multiprocessor systems ?
- Preemptive kernels
  - process executing in kernel mode can be preempted
  - suitable for real-time programming
  - more responsive

Roadmap

- Solutions for mutual exclusion
  - Peterson’s algorithm (Software)
  - Synchronization instructions (Hardware)

- **High-level synchronization mechanisms**
  - **Mutex**
  - **Semaphore**
  - **Monitor**

KU THE UNIVERSITY OF KANSAS

Spinlock using TestAndSet

```
void init_lock (int *mutex)
{
   *mutex = 0;
}

void lock (int *mutex)
{
   while(TestAndSet(mutex))
      ;
}

void unlock (int *mutex)
{
   *mutex = 0;
}
```

What’s Wrong With **Spin**locks?

- Very wasteful
  - Waiting thread continues to use CPU cycles
  - While doing absolutely nothing but wait
  - 100% CPU utilization, but no useful work done
  - Power consumption, fan noise, …
- Useful when
  - You hold the lock only *briefly*
- Otherwise
  - A better solution is needed

Mutex – Blocking Lock

- Instead of spinning
  - Let the thread sleep
    - There can be multiple waiting threads
  - In the meantime, let other threads use the CPU
  - When the lock is released, wake-up one thread
    - Pick one if there multiple threads were waiting

More reading: [mutex.c in Linux]()

```
void mutex_init (mutex_t *lock)
{
    lock->value = 0;
    list_init(&lock->wait_list);
    spin_lock_init(&lock->wait_lock);
}

void  mutex_lock (mutex_t *lock)
{
…
    while(TestAndSet(&lock->value)) {
…
…
…
…
…
    }
…
}

void mutex_unlock (mutex_t *lock)
{
…
    lock->value = 0;
…
…
…
}
```

Thread waiting list

To protect waiting list

More reading: mutex.c in Linux

```
void mutex_init (mutex_t *lock)
{
    lock->value = 0;
    list_init(&lock->wait_list);
    spin_lock_init(&lock->wait_lock);
}

void  mutex_lock (mutex_t *lock)
{
...
    while(TestAndSet(&lock->value)) {
        current->state = WAITING;
        list_add(&lock->wait_list, current);
...
        schedule();
...
    }
...
}

void mutex_unlock (mutex_t *lock)
{
...
    lock->value = 0;
...
...
...
}
```

Thread waiting list

To protect waiting list

Thread state change

Add the current thread to the waiting list

Sleep or schedule another thread

More reading: [mutex.c in Linux]()

```
void mutex_init (mutex_t *lock)
{
    lock->value = 0;
    list_init(&lock->wait_list);
    spin_lock_init(&lock->wait_lock);
}

void  mutex_lock (mutex_t *lock)
{
    spin_lock(&lock->wait_lock);
    while(TestAndSet(&lock->value)) {
        current->state = WAITING;
        list_add(&lock->wait_list, current);
...
        schedule();
...
    }
   spin_unlock(&lock->wait_lock);
}

void mutex_unlock (mutex_t *lock)
{
...
    lock->value = 0;
...
...
...
}
```

Thread waiting list

To protect waiting list

Thread state change

Add the current thread to the waiting list

Sleep or schedule another thread

More reading: mutex.c in Linux

```
void mutex_init (mutex_t *lock)
{
    lock->value = 0;
    list_init(&lock->wait_list);
    spin_lock_init(&lock->wait_lock);
}

void  mutex_lock (mutex_t *lock)
{
    spin_lock(&lock->wait_lock);
    while(TestAndSet(&lock->value)) {
        current->state = WAITING;
        list_add(&lock->wait_list, current);
        spin_unlock(&lock->wait_lock);
        schedule();
        spin_lock(&lock->wait_lock);
    }
   spin_unlock(&lock->wait_lock);
}

void mutex_unlock (mutex_t *lock)
{
...
    lock->value = 0;
...
...
...
}
```

Thread waiting list

To protect waiting list

Thread state change

Add the current thread to the waiting list

Sleep or schedule another thread

More reading: [mutex.c in Linux]()

```
void mutex_init (mutex_t *lock)
{
    lock->value = 0;
    list_init(&lock->wait_list);          <- Thread waiting list
    spin_lock_init(&lock->wait_lock);     <- To protect waiting list
}

void  mutex_lock (mutex_t *lock)
{
    spin_lock(&lock->wait_lock);
    while(TestAndSet(&lock->value)) {
        current->state = WAITING;              <- Thread state change
        list_add(&lock->wait_list, current);   <- Add the current thread to the waiting list
        spin_unlock(&lock->wait_lock);
        schedule();                            <- Sleep or schedule another thread
        spin_lock(&lock->wait_lock);
    }
   spin_unlock(&lock->wait_lock);
}

void mutex_unlock (mutex_t *lock)
{
...
    lock->value = 0;
    if (!list_empty(&lock->wait_list))         <- Someone is waiting for the lock
        wake_up_process(&lock->wait_list)      <- Wake-up a waiting thread
...
}
```

More reading: [mutex.c in Linux]()

```
void mutex_init (mutex_t *lock)
{
   lock->value = 0;
   list_init(&lock->wait_list);
   spin_lock_init(&lock->wait_lock);
}

void  mutex_lock (mutex_t *lock)
{
   spin_lock(&lock->wait_lock);
   while(TestAndSet(&lock->value)) {
      current->state = WAITING;
      list_add(&lock->wait_list, current);
      spin_unlock(&lock->wait_lock);
      schedule();
      spin_lock(&lock->wait_lock);
   }
  spin_unlock(&lock->wait_lock);
}

void mutex_unlock (mutex_t *lock)
{
   spin_lock(&lock->wait_lock);
   lock->value = 0;
   if (!list_empty(&lock->wait_list))
      wake_up_process(&lock->wait_list)
   spin_unlock(&lock->wait_lock);
}
```

Thread waiting list

To protect waiting list

Thread state change

Add the current thread to the waiting list

Sleep or schedule another thread

Someone is waiting for the lock

Wake-up a waiting thread

High-level Synchronization Primitives

- Lock (mutex) is great, but…
  - Too low-level primitive
  - Sometimes we need more **powerful primitives**

- Semaphore
  - Binary/integer semaphore
- Monitor
  - Condition variable

Semaphore

- High-level synchronization primitive
  - Designed by Dijkstra in 1960’
- Definition
  - Semaphore is an integer variable
  - Only two operations are possible:
    - P() or wait() or down()
      - **Wait** until the semaphore value to become > 0, then **decrements** it by 1.
    - V() or signal() or up()
      - **Increments** the semaphore value by 1

KU THE UNIVERSITY OF KANSAS