Synchronization

My formulation

- **OS = data structurs + synchronization**
- **Synchronization problems make writing OS code challenging**
- **Demand exceptional coding skills**

Race problem

```
long c = 0, c1 = 0, c2 = 0, run = 1;
void *thread1(void *arg) {
    while(run == 1) {
        c++;
        c1++;
    }
}
void *thread2(void *arg) {
    while(run == 1) {
        c++;
        c2++;
    }
}
```

```
int main() {
    pthread_t th1, th2;
    pthread_create(&th1, NULL, thread1, NULL);
    pthread_create(&th2, NULL, thread2, NULL);
    //fprintf(stdout, "Ending main\n");
    sleep(2);
    run = 0;
    fprintf(stdout, "c = %ld c1+c2 = %ld c1 = %ld c2 = %ld \n", c, c1+c2, c1, c2);
    fflush(stdout);
}
```

Race problem

- **On earlier slide**
  - **Value of c should be equal to c1 + c2, but it is not!**
  - **Why?**
- **There is a “race” between thread1 and thread2 for updating the variable c**
- **thread1 and thread2 may get scheduled in any order and *interrupted* any point in time**
- **The changes to c are not** atomic!
  - **What does that mean?**

Race problem

- **C++, when converted to assembly code, could be**

  mov c, r1

  add r1, 1

  mov r1, c

- **Now following sequence of instructions is possible among thread1 and thread2**

  thread1: mov c, r1

  thread2: mov c, r1

  thread1: add r1, 1

  thread1: mov r1, c

  thread2: add r1, 1

  thread2: mov r1, c

- **What will be value in c, if initially c was, say 5?**
  - **It will be 6, when it is expected to be 7. Other variations also possible.**

Races: reasons

- **Interruptible kernel**
  - **If entry to kernel code does not disable interrupts, then modifications to any kernel data structure can be left incomplete**
  - **This introduces concurrency**
- **Multiprocessor systems**
  - **On SMP systems: memory is shared, kernel and process code run on all processors**
  - **Same variable can be updated parallelly (not concurrently)**
- **What about non-interruptible kernel on multiprocessor systems?**
- **What about non-interruptible kernel on uniprocessor systems?**

Critical Section Problem

- Consider system of n processes {p0, p1, … pn-1}
- Each process has critical section segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- Critical section problem is to design protocol to solve this
- Each process must ask permission to enter critical section in entry section, may follow critical section with exit section, then remainder section
- Especially challenging with preemptive kernels

Critical Section problem

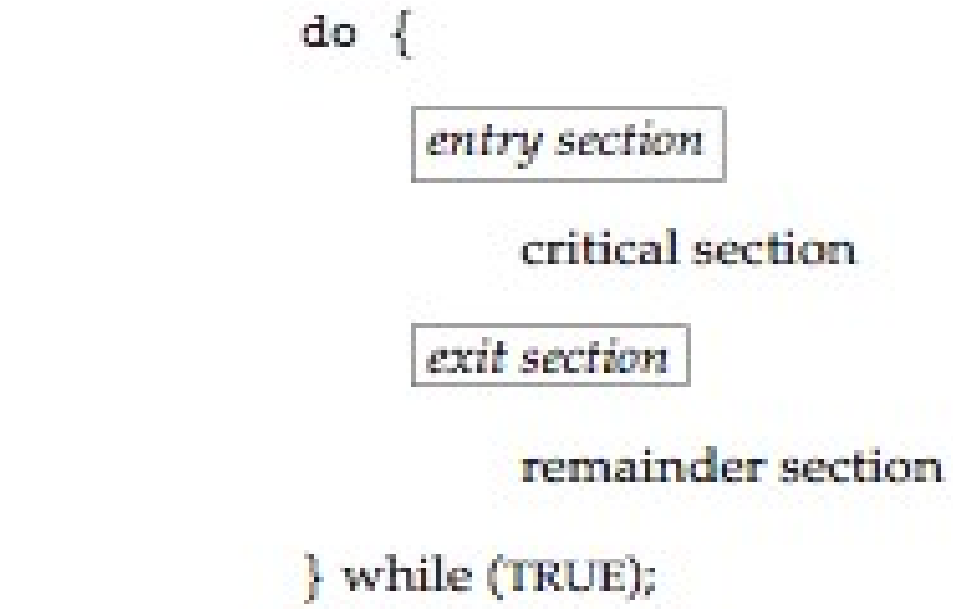


Figure 6.1 General structure of a typical process P_i.

Expected solution characteristics

- **1. Mutual Exclusion**
  - **If process Pi is executing in its critical section, then no other processes can be executing in their critical sections**
- **2. Progress**
  - **If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely**
- **3. Bounded Waiting**
  - **A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted**
    - **Assume that each process executes at a nonzero speed**
    - **No assumption concerning relative speed of the n processes**

suggested solution - 1

```
int flag = 1;
void *thread1(void *arg) {
    while(run == 1) {
        while(flag == 0)
        ;
        flag = 0;
        c++;
        flag = 1;
        c1++;
    }
}
```

- What's wrong here?
- Assumes that

  while(flag ==) ; flag = 0

  will be atomic

suggested solution - 2

```
int flag = 0;
void *thread1(void *arg) {
    while(run == 1) {
        if(flag)
            c++;
        else
            continue;
        c1++;
        flag  = 0;
    }
}
```

```
void *thread2(void *arg) {
    while(run == 1) {
        if(!flag)
            c++;
        else
            continue;
        c2++;
        flag = 1;
    }
}
```

Peterson's solution

- Two process solution
- Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted
- The two processes share two variables:

  ```
  int turn;

  Boolean flag[2]
  ```

- The variable turn indicates whose turn it is to enter the critical section
- The flag array is used to indicate if a process is ready to enter the critical section. flag[i] = true implies that process Pi is ready!

Peterson's solution

```
do {
   flag[i] = TRUE;
   turn = j;
   while (flag[j] && turn == j)
         ;
   critical section
   flag[i] = FALSE;
   remainder section
} while (TRUE);
```

- Provable that
  - Mutual exclusion is preserved
  - Progress requirement is satisfied
  - Bounded-waiting requirement is met

Hardware solution – the one actually implemented

- Many systems provide hardware support for critical section code
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
  - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
  - Atomic = non-interruptable
  - Either test memory word and set value
  - Or swap contents of two memory words
  - Basically two operations (read/write) done atomically in hardware

Solution using test-and-set

```
lock = false; //global

do {
    while ( TestAndSet (&lock ))
              ;   // do nothing
     //    critical section
    lock = FALSE;
    //      remainder section
  } while (TRUE);
```

Definition:

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

Solution using swap

```
lock = false; //global

do {
   key = true
    while ( key == true))
      swap(&lock, &key)
     //    critical section
    lock = FALSE;
    //      remainder section
  } while (TRUE);
```

Spinlock

- A lock implemented to do ‘busy-wait’
- Using instructions like T&S or Swap
- As shown on earlier slides

```
spinlock(int *lock){
    While(test-and-set(lock))
        ;
}
spinunlock(lock *lock) {
    *lock = false;
}
```

Bounded wait M.E. with T&S

```
do {
        waiting[i] = TRUE;
        key = TRUE;
        while (waiting[i] && key)
                key = TestAndSet(&lock);
        waiting[i] = FALSE;
                // critical section
        j = (i + 1) % n;
        while ((j != i) && !waiting[j])
                j = (j + 1) % n;
        if (j == i)
                lock = FALSE;
        else
                waiting[j] = FALSE;
                // remainder section
} while (TRUE);
```

Some thumb-rules of spinlocks

- **Never block a process holding a spinlock !**
- **Typical code:**

```
while(condition)
    { Spin-unlock()
    Schedule()
    Spin-lock()
}
```

- **Hold a spin lock for only a short duration of time**
  - **Spinlocks are preferable on multiprocessor systems**
  - **Cost of context switch is a concern in case of sleep-wait locks**
  - **Short = < 2 context switches**

sleep-locks

- Spin locks result in busy-wait
- CPU cycles wasted by waiting processes/threads
- Solution – threads keep waiting for the lock to be available
  - Move thread to wait queue
  - The thread holding the lock will wake up one of them

Sleep locks/mutexes

```
//ignore syntactical issues
typedef struct mutex {
    int islocked;
    int spinlock;
    waitqueue q;
    }mutex;
wait(mutex *m) {
    spinlock(m->spinlock);
    while(m->islocked)
        Block(m, m->spinlock)
    lk->islocked = 1;
    spinunlock(m->spinlock);
}
```

```
Block(mutex *m, spinlock *sl) {
    spinunlock(sl);
    currprocess->state = WAITING
    move current process to  m->q
    Sched();
    spinlock(sl);
}
release(mutex *m) {
    spinlock(m->spinlock);
    m->islocked = 0;
    Some process in m->queue
    =RUNNABLE;
    spinunlock(m->spinlock);
    }
```

Locks in xv6 code

struct spinlock

```
// Mutual exclusion lock.
struct spinlock {
  uint locked;       // Is the lock held?

  // For debugging:
  char *name;        // Name of lock.
  struct cpu *cpu;   // The cpu holding the lock.
  uint pcs[10];      // The call stack (an array of program counters)
                     // that locked the lock.
};
```

spinlocks in xv6 code

```
struct {
  struct spinlock lock;
  struct buf buf[NBUF];
  struct buf head;
} bcache;
struct {
  struct spinlock lock;
  struct file file[NFILE];
} ftable;
struct {
  struct spinlock lock;
  struct inode inode[NINODE];
} icache;
struct sleeplock {
  uint locked;       // Is the lock held?
  struct spinlock lk;
```

```
static struct spinlock idelock;
struct {
  struct spinlock lock;
  int use_lock;
  struct run *freelist;
} kmem;
struct log {
  struct spinlock lock;
...}
struct pipe {
  struct spinlock lock;
...}
struct {
  struct spinlock lock;
  struct proc proc[NPROC];
} ptable;
struct spinlock tickslock;
```

Spinlock on xv6

```
static inline uint
xchg(volatile uint *addr, uint newval)
{
  uint result;
   // The + in "+m" denotes a read-modify-write operand.
  asm volatile("lock; xchgl %0, %1" :
               "+m" (*addr), "=a" (result) :
               "1" (newval) :
               "cc");
  return result;
}
struct spinlock {
  uint locked;       // Is the lock held?


  // For debugging:
  char *name;        // Name of lock.
  struct cpu *cpu;   // The cpu holding the lock.
  uint pcs[10];      // The call stack (an array of program counters) that locked the lock.
};
```

```
void acquire(struct spinlock *lk)
{
  pushcli(); // disable interrupts to avoid deadlock.
 // The xchg is atomic.
  while(xchg(&lk->locked, 1) != 0)
    ;
//extra debugging code
}
void release(struct spinlock *lk)
{  //extra debugging code
  asm volatile("movl $0, %0" : "+m" (lk->locked) : );
  popcli();
}
```

```
Void acquire(struct spinlock *lk)
{
  pushcli(); // disable interrupts to avoid deadlock.
  if(holding(lk))
    panic("acquire");
    ......
void pushcli(void)
{
  int eflags;

  eflags = readeflags();
  cli();
  if(mycpu()->ncli == 0)
    mycpu()->intena = eflags & FL_IF;
  mycpu()->ncli += 1;
}
static inline uint
readeflags(void)
{
  uint eflags;
  asm volatile("pushfl; popl %0" : "=r" (eflags));
  return eflags;
}
```

spinlocks

- Pushcli() - disable interrupts on that processor
- One after another many acquire() can be called on different spinlocks
  - Keep a count of them in mycpu()->ncli

```
void
release(struct spinlock *lk)
{
...
    asm volatile("movl $0, %0" : "+m" (lk->locked) : );
    popcli();
}
.
Void popcli(void)
{
  if(readeflags()&FL_IF)
    panic("popcli - interruptible");
  if(--mycpu()->ncli < 0)
    panic("popcli");
  if(mycpu()->ncli == 0 && mycpu()->intena)
    sti();
}
```

spinlocks

- **Popcli()**
  - **Restore interrupts if last popcli() call restores ncli to 0 & interrupts were enabled before pushcli() was called**

spinlocks

- Always disable interrupts while acquiring spinlock
  - Suppose iderw held the idelock and then got interrupted to run ideintr.
  - Ideintr would try to lock idelock, see it was held, and wait for it to be released.
  - In this situation, idelock will never be released
  - Deadlock
- General OS rule: if a spin-lock is used by an interrupt handler, a processor must never hold that lock with interrupts enabled
- Xv6 rule: when a processor enters a spin-lock critical section, xv6 always ensures interrupts are disabled on that processor.

sleeplocks

- **Sleeplocks don’t spin. They move a process to a wait-queue if the lock can’t be acquired**
- **XV6 approach to “wait-queues”**
  - **Any memory address serves as a “wait channel”**
  - **The sleep() and wakeup() functions just use that address as a ‘condition’**
  - **There are no per condition process queues! Just one global queue of processes used for scheduling, sleep, wakeup etc. --> Linear search everytime !**
    - **costly, but simple**

sleep()

```
void
sleep(void *chan, struct spinlock *lk)
{
  struct proc *p = myproc();
....
  if(lk != &ptable.lock){
    acquire(&ptable.lock);
    release(lk);
  }
  p->chan = chan;
 p->state = SLEEPING;
  sched();
  // Reacquire original lock.
   if(lk != &ptable.lock){
    release(&ptable.lock);
    acquire(lk);
  }
```

- At call must hold lock on the resource on which you are going to sleep
- since you are going to change p-> values & call sched(), hold ptable.lock if not held
- p->chan = given address remembers on which condition the process is waiting
- call to sched() blocks the process

Calls to sleep() : examples of “chan” (output from cscope)

0 console.c
consoleread 251
sleep(&input.r, &cons.lock);

2 ide.c iderw
169 sleep(b, &idelock);

3 log.c begin_op
131 sleep(&log, &log.lock);

6 pipe.c piperead
111 sleep(&p->nread, &p->lock);

7 proc.c wait
317 sleep(curproc, &ptable.lock);

8 sleeplock.c
acquiresleep 28
sleep(lk, &lk->lk);

9 sysproc.c
sys_sleep 74
sleep(&ticks, &tickslock);

```
void wakeup(void *chan)
{
  acquire(&ptable.lock);
  wakeup1(chan);
  release(&ptable.lock);
}
static void wakeup1(void *chan)
{
  struct proc *p;

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == SLEEPING && p->chan == chan)
      p->state = RUNNABLE;
}
```

Wakeup()

- Acquire ptable.lock since you are going to change ptable and p-> values
- just linear search in process table for a process where p->chan is given address
- Make it runnable

sleeplock

```
// Long-term locks for processes
struct sleeplock {
  uint locked;       // Is the lock held?
  struct spinlock lk; // spinlock protecting this sleep lock

  // For debugging:
  char *name;        // Name of lock.
  int pid;           // Process holding lock
};
```

Sleeplock acquire and release

```
void
acquiresleep(struct sleeplock *lk)
{
  acquire(&lk->lk);
  while (lk->locked) {
    /* Abhijit: interrupts are not disabled in sleep !*/
    sleep(lk, &lk->lk);
  }
  lk->locked = 1;
  lk->pid = myproc()->pid;
  release(&lk->lk);
}
```

```
void
releasesleep(struct sleeplock *lk)
{
  acquire(&lk->lk);
  lk->locked = 0;
  lk->pid = 0;
  wakeup(lk);
  release(&lk->lk);
}
```

Where are sleeplocks used?

- **struct buf**
  - **waiting for I/O on this buffer**
- **struct inode**
  - **waiting for I/o to this inode**

- **Just two !**

Sleeplocks issues

- **sleep-locks support yielding the processor during their critical sections.**
- **This property poses a design challenge:**
  - **if thread T1 holds lock L1 and has yielded the processor (waiting for some other condition),**
  - **and thread T2 wishes to acquire L1,**
  - **we have to ensure that T1 can execute**
  - **while T2 is waiting so that T1 can release L1.**
  - **T2 can't use the spin-lock acquire function here: it spins with interrupts turned off, and that would prevent T1 from running.**
- **To avoid this deadlock, the sleep-lock acquire routine (called acquiresleep) yields the processor while waiting, and does not disable interrupts.**

Sleep-locks leave interrupts enabled, they cannot be used in interrupt handlers.

More needs of synchronization

- Not only critical section problems
- Run processes in a particular order
- Allow multiple processes read access, but only one process write access
- Etc.

Semaphore

- **Synchronization tool that does not require busy waiting**
- **Semaphore S – integer variable**
- **Two standard operations modify S: wait() and signal()**
  - **Originally called P() and V()**
- **Less complicated**

- **Can only be accessed via two indivisible (atomic) operations**

```
wait (S) {
        while S <= 0
            ; // no-op
        S--;
    }
signal (S) {
      S++;
    }
```

--> Note this is Signal() on a semaphore, different froms signal system call

Semaphore for synchronization

- **Counting** semaphore – integer value can range over an unrestricted domain
- **Binary** semaphore – integer value can range only between 0 and 1; can be simpler to implement

  Also known as **mutex locks**
- Can implement a counting semaphore S as a binary semaphore
- Provides mutual exclusion

```
Semaphore mutex;    //  initialized to 1
do {
wait (mutex);
        // Critical Section
    signal (mutex);
// remainder section
} while (TRUE)
```

Semaphore implementation

```
Wait(sem *s) {
    while(s <=0)
        block(); // could be ";"
    s--;
}
signal(sem *s) {
    s++;
}
```

- Left side – expected behaviour
- Both the wait and signal should be atomic.
- This is the sematics of the semaphore.

Semaphore implementation? - 1

```
struct semaphore {
    int val;
    spinlock lk;
};
sem_init(semaphore *s, int initval) {
    s->val = initval;
    s->sl = 0;
}
wait(semaphore *s) {
    spinlock(&(s->sl));
    while(s->val <=0)
    ;
    (s->val)--;
    spinunlock(&(s->sl));
}
```

```
signal(seamphore *s) {
    spinlock(*(s->sl));
    (s->val)++;
    spinunlock(*(s->sl));
}
```

- suppose 2 processes trying wait.

val = 1;

Th1: spinlock Th2: spinlock-waits

Th1: while -> false, val-- => 0; spinulock;

Th2: spinlock success; while() -> true, loops;

Th1: is done with critical section, it calls signal. it calls spinlock() -> wait.

Who is holding spinlock-> Th2. Itis waiting for val > 0. Who can set value > 0 , ans: Th1, and Th1 is waiting for spinlock which is held by The2.

circular wait. Deadlock.

None of them will proceed.

Semaphore implementation? - 2

```
struct semaphore {
    int val;
    spinlock lk;
};
sem_init(semaphore *s, int initval) {
    s->val = initval;
    s->sl = 0;
}
signal(seamphore *s) {
    spinlock(*(s->sl));
    (s->val)++;
    spinunlock(*(s->sl));
}
```

```
wait(semaphore *s) {
    spinlock(&(s->sl));
    while(s->val <=0) {
        spinunlock(&(s->sl));
        spinlock(&(s->sl));
    }
    (s->val)--;
    spinunlock(&(s->sl));
}
```

Problem: race in spinlock of whille loop and signal's spinlock. Bounded wait not guaranteed.

Spinlocks are not good for a long wait.

Semaphore implementation? - 3, idea

```
struct semaphore {
    int val;
    spinlock lk;
};
sem_init(semaphore *s, int initval) {
    s->val = initval;
    s->sl = 0;
}
block() {
    put this current process on wait-q;
    schedule();
}
```

```
wait(semaphore *s) {
    spinlock(&(s->sl));
    while(s->val <=0) {
        Block();
    }
    (s->val)--;
    spinunlock(&(s->sl));
}
signal(seamphore *s) {
    spinlock(*(s->sl));
    (s->val)++;
    spinunlock(*(s->sl));
}
```

Semaphore implementation? - 3a

```
struct semaphore {
    int val;
    spinlock lk;
    list l;
};
sem_init(semaphore *s, int initval) {
    s->val = initval;
    s->sl = 0;
}
block(semaphore *s) {
    listappend(s->l, current);
    schedule();
}
```

problem is that block() will be called without holding the spinlock and the access to the list is not protected.

Note that - so far we have ignored changes to signal()

```
wait(semaphore *s) {
    spinlock(&(s->sl));
    while(s->val <=0) {
        spinunlock(&(s->sl));
        block(s);
    }
    (s->val)--;
    spinunlock(&(s->sl));
}
signal(seamphore *s) {
    spinlock(*(s->sl));
    (s->val)++;
    spinunlock(*(s->sl));
}
```

Semaphore implementation? - 3b

```
struct semaphore {
    int val;
    spinlock lk;
    list l;
};
sem_init(semaphore *s, int initval) {
    s->val = initval;
    s->sl = 0;
}
block(semaphore *s) {
    listappend(s->l, current);
    spinunlock(&(s->sl));
    schedule();
}
```

```
wait(semaphore *s) {
    spinlock(&(s->sl));
    while(s->val <=0) {
        block(s);
    }
    (s->val)--;
    spinunlock(&(s->sl));
}
signal(seamphore *s) {
    spinlock(*(s->sl));
    (s->val)++;
    x = dequeue(s->sl) and enqueue(readyq, x);
    spinunlock(*(s->sl));
}
```

Problem: after a blocked process comes out of the block, it does not hold the spinlock and it's goinng to change the s->sl;

Semaphore implementation? - 3c

```
struct semaphore {
    int val;
    spinlock lk;
    list l;
};
sem_init(semaphore *s, int initval) {
    s->val = initval;
    s->sl = 0;
}
block(semaphore *s) {
    listappend(s->l, current);
    spinunlock(&(s->sl));
    schedule();
}
```

```
wait(semaphore *s) {
    spinlock(&(s->sl)); // A
    while(s->val <=0) {
        block(s);
        spinlock(&(s->sl)); // B
    }
    (s->val)--;
    spinunlock(&(s->sl));
}
signal(seamphore *s) {
    spinlock(*(s->sl));
    (s->val)++;
    x = dequeue(s->sl) and enqueue(readyq, x);
    spinunlock(*(s->sl));
}
```

Question: there is race between A and B. Can we guarantee bounded wait ?

Semaphore Implementation

- Must guarantee that no two processes can execute wait () and signal () on the same semaphore at the same time
- Thus, implementation becomes the critical section problem where the wait and signal code are placed in the critical section
  - Could now have busy waiting in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution

Semaphore in Linux

```
struct semaphore {
    raw_spinlock_t      lock;
    unsigned int        count;
    struct list_head    wait_list;
};
static noinline void __sched
__down(struct semaphore *sem)
{
    __down_common(sem,
TASK_UNINTERRUPTIBLE,
MAX_SCHEDULE_TIMEOUT);
}
```

```
void down(struct semaphore *sem)
{
    unsigned long flags;


    raw_spin_lock_irqsave(&sem->lock, flags);
    if (likely(sem->count > 0))
        sem->count--;
    else
        __down(sem);
    raw_spin_unlock_irqrestore(&sem->lock,
flags);
}
```

Semaphore in Linux

```
static inline int __sched
__down_common(struct semaphore
*sem, long state, long timeout)
{
    struct task_struct *task = current;
    struct semaphore_waiter waiter;
    list_add_tail(&waiter.list, &sem-
>wait_list);
    waiter.task = task;
    waiter.up = false;
    for (;;) {
        if (signal_pending_state(state, task))
            goto interrupted;
        if (unlikely(timeout <= 0))
            goto timed_out;
        __set_task_state(task, state);
        raw_spin_unlock_irq(&sem->lock);
        timeout = schedule_timeout(timeout);
        raw_spin_lock_irq(&sem->lock);
        if (waiter.up)
            return 0;
    }
    ....
}
```

Different uses of semaphores

For mutual exclusion

```
/*During inialization*/
semaphore sem;
initsem (&sem, 1);

/* On each use*/
P (&sem);
Use resource;
V (&sem);
```

Event-wait

```
/* During initialization */
semaphore event;
initsem (&event, 0); /* probably at boot time */

/* Code executed by thread that must wait on event */
P (&event); /* Blocks if event has not occurred */
/* Event has occurred */
V (&event); /* So that another thread may wake up */
/* Continue processing */

/* Code executed by another thread when event occurs */
V (&event); /* Wake up one thread */
```

Control countable resources

```
/* During initialization */
semaphore counter;
initsem (&counter, resourceCount);
/* Code executed to use the resource */
P (&counter); /* Blocks until resource is available */
Use resource; /* Guaranteed to be available now */
V (&counter); /* Release the resource */
```

Drawbacks of semaphores

- Need to be implemented using lower level primitives like spinlocks
- Context-switch is involved in blocking and signaling – time consuming
- Can not be used for a short critical section

Deadlocks

Deadlock

- two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let S and Q be two semaphores initialized to 1

P0	P1
wait (S);	wait (Q);
wait (Q);	wait (S);
. .	
. .	
. .	
signal (S);	signal (Q);
signal (Q);	signal (S);

Example of deadlock

- **Let’s see the pthreads program : deadlock.c**
- **Same programe as on earlier slide, but with pthread_mutex_lock();**

Non-deadlock, but similar situations

- **Starvation – indefinite blocking**
  - **A process may never be removed from the semaphore queue in which it is suspended**
- **Priority Inversion**
  - **Scheduling problem when lower-priority process holds a lock needed by higher-priority process (so it can not pre-empt lower priority process), and a medium priority process (that does not need the lock) pre-empts lower priority task, denying turn to higher priority task**
  - **Solved via priority-inheritance protocol : temporarily enhance priority of lower priority task to highest**

Livelock

- Similar to deadlock, but processes keep doing ‘useless work’
- E.g. two people meet in a corridor opposite each other
  - Both move to left at same time
  - Then both move to right at same time
  - Keep Repeating!
- No process able to progress, but each doing ‘some work’ (not sleeping/waiting), state keeps changing

Livelock example

```
#include <stdio.h>
#include <pthread.h>
struct person {
    int otherid;
    int otherHungry;
    int myid;
};
int main() {
    pthread_t th1, th2;
    struct person one, two;
    one.otherid = 2; one.myid = 1;
    two.otherid = 1; two.myid = 2;
    one.otherHungry = two.otherHungry = 1;
    pthread_create(&th1, NULL, eat, &one);
    pthread_create(&th2, NULL, eat, &two);
    printf("Main: Waiting for threads to get over\n");
    pthread_join(th1, NULL);
    pthread_join(th2, NULL);
    return 0;
}
```

```
/* thread two runs in this function */
int spoonWith = 1;
void *eat(void *param)
{
  int eaten = 0;
  struct person person= *(struct person *)param;
  while (!eaten) {
        if(spoonWith == person.myid)
            printf("%d going to eat\n", person.myid);
        else
            continue;
        if(person.otherHungry) {
            printf("You eat %d\n", person.otherid);
            spoonWith = person.otherid;
            continue;
        }
        printf("%d is eating\n", person.myid);
        break;
  }
}
```

More on deadlocks

- **Under which conditions they can occur?**
- **How can deadlocks be avoided/prevented?**
- **How can a system recover if there is a deadlock ?**

System model for understanding deadlocks

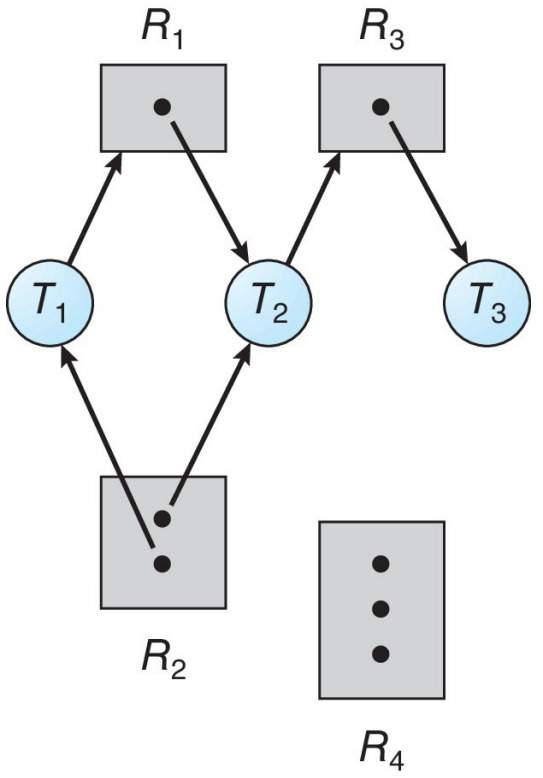
- System consists of resources
- Resource types R1, R2, . . ., Rm
  - CPU cycles, memory space, I/O devices
  - Resource: Most typically a lock, synchronization primitive
- Each resource type Ri has Wi instances.
- Each process utilizes a resource as follows:
  - request
  - use
  - release

Deadlock characterisation

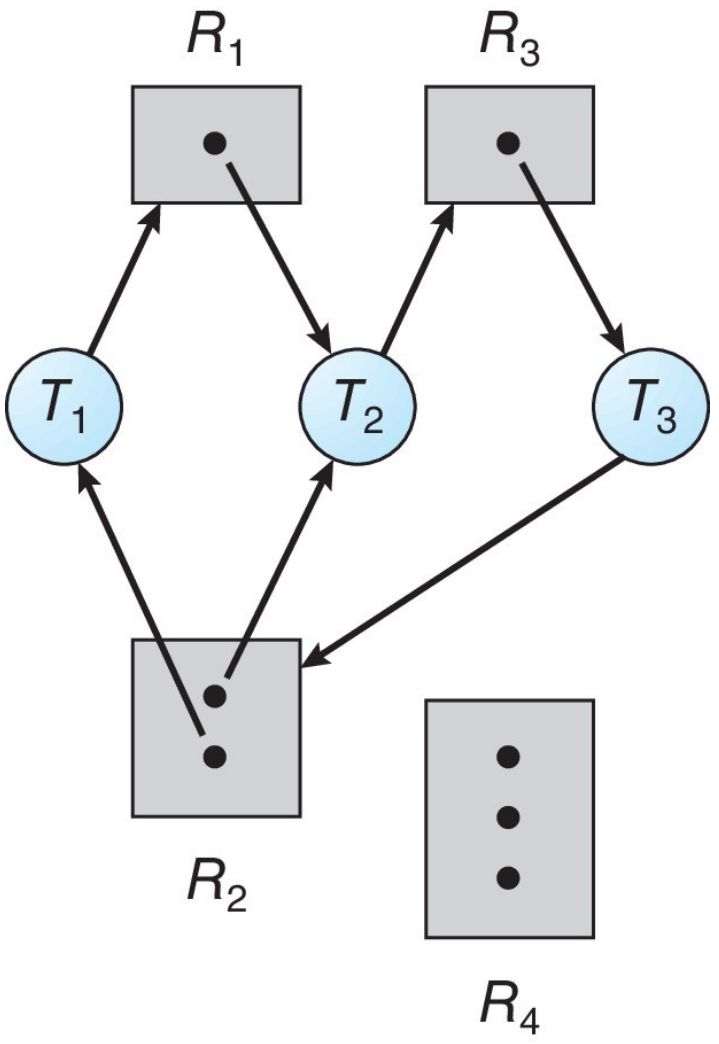
- **Deadlock is possible only if ALL of these conditions are TRUE at the same time**
  - **Mutual exclusion:** only one process at a time can use a resource
  - **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes
  - **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task
  - **Circular wait:** there exists a set {P0, P1, …, Pn} of waiting processes such that P0 is waiting for a resource that is held by P1, P1 is waiting for a resource that is held by P2, …, Pn–1 is waiting for a resource that is held by Pn, and Pn is waiting for a resource that is held by P0.

Resource Allocation Graph Example

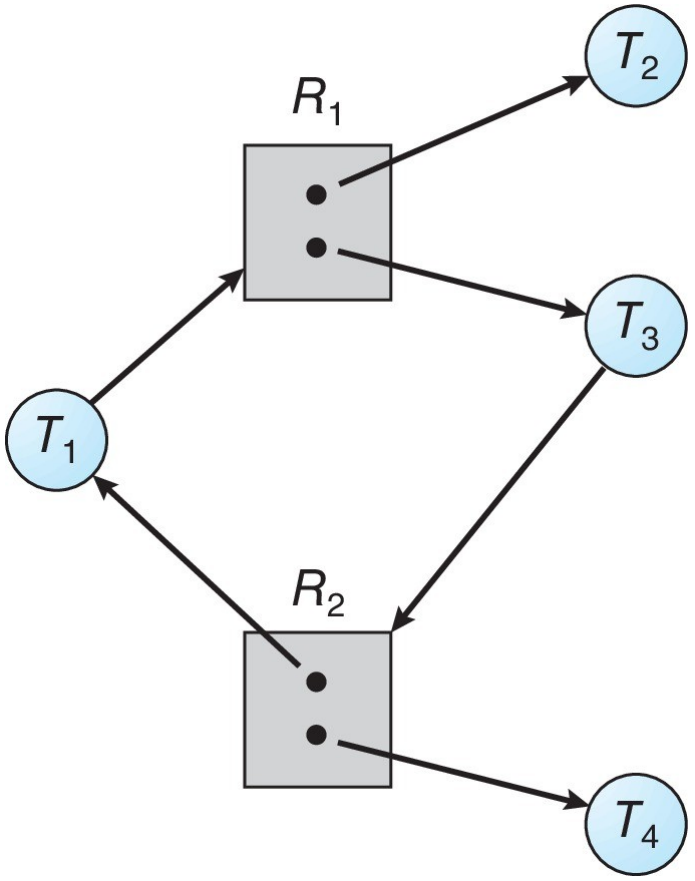
- One instance of R1
- Two instances of R2
- One instance of R3
- Three instance of R4
- T1 holds one instance of R2 and is waiting for an instance of R1
- T2 holds one instance of R1, one instance of R2, and is waiting for an instance of R3
- T3 is holds one instance of R3



Resource Allocation Graph with a Deadlock



Graph with a Cycle But no Deadlock



Basic Facts

- **If graph contains no cycles -> no deadlock**
- **If graph contains a cycle :**
  - **if only one instance per resource type, then deadlock**
  - **if several instances per resource type, possibility of deadlock**

Methods for Handling Deadlocks

- **Ensure that the system will never enter a deadlock state:**
  - **1) Deadlock prevention**
  - **2) Deadlock avoidance**
  - **3) Allow the system to enter a deadlock state and then recover**
  - **4) Ignore the problem and pretend that deadlocks never occur in the system.**

(1) Deadlock Prevention

- **Invalidate one of the four necessary conditions for deadlock:**
- **Mutual Exclusion – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources**
- **Hold and Wait – must guarantee that whenever a process requests a resource, it does not hold any other resources**
  - **Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none allocated to it.**
  - **Low resource utilization; starvation possible**

(1) Deadlock Prevention (Cont.)

- **No Preemption:**
  - **If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released**
  - **Preempted resources are added to the list of resources for which the process is waiting**
  - **Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting**
- **Circular Wait:**
  - **Impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration**

(1) Deadlock prevention: Circular Wait

- **Invalidating the circular wait condition is most common.**
- **Simply assign each resource (i.e., mutex locks) a unique number.**
- **Resources must be acquired in order.**
- **If:**

  first_mutex = 1

  second_mutex = 5

  code for thread_two could not be written as follows:

```
/* thread_one runs in this function */
void *do_work_one(void *param)
{
   pthread_mutex_lock(&first_mutex);
   pthread_mutex_lock(&second_mutex);
   /**
    * Do some work
    */
   pthread_mutex_unlock(&second_mutex);
   pthread_mutex_unlock(&first_mutex);

   pthread_exit(0);
}

/* thread_two runs in this function */
void *do_work_two(void *param)
{
   pthread_mutex_lock(&second_mutex);
   pthread_mutex_lock(&first_mutex);
   /**
    * Do some work
    */
   pthread_mutex_unlock(&first_mutex);
   pthread_mutex_unlock(&second_mutex);

   pthread_exit(0);
}
```

(1) Preventing deadlock: cyclic wait

- **Locking hierarchy : Highly preferred technique in kernels**
  - **Decide an ordering among all ‘locks’**
  - **Ensure that on ALL code paths in the kernel, the locks are obtained in the decided order!**
  - **Poses coding challenges!**
  - **A key differentiating factor in kernels**
  - **Do not look at only the current lock being taken, look at all the locks the code may be holding at any given point in code!**

(1) Prevention in Xv6: Lock Ordering

- lock on the directory, a lock on the new file's inode, a lock on a disk block buffer, idelock, and ptable.lock.

(2) Deadlock avoidance

- **Requires that the system has some additional a priori information available**
  - **Simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need**
  - **The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition**
  - **Resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes**

(2) Deadlock avoidance

- **Please see: concept of safe states, unsafe states, Banker's algorithm**

(3) Deadlock detection and recovery

- How to detect a deadlock in the system?
- The Resource-Allocation Graph is a graph. Need an algorithm to detect cycle in a graph.
- How to recover?
  - Abort all processes or abort one by one?
  - Which processes to abort?
    - Priority ?
    - Time spent since forked()?
    - Resources used?
    - Resources needed?
    - Interactive or not?
    - How many need to be terminated?

“Condition” Synchronization Tool

What is condition variable?

- A variable with a sleep queue
- Threads can sleep on it, and wake-up all remaining

```
Struct condition {
   Proc *next
   Proc *prev
   Spinlock *lock
}
```

Different variables of this type can be used as different ‘conditions

Code for condition variables

```
//Spinlock s is held before calling wait
void wait (condition *c, spinlock_t *s)
(
   spin_lock (&c->listLock);
   add self to the linked list;
   spin_unlock (&c->listLock);
   spin_unlock (s); /* release
   spinlock before blocking */
    swtch(); /* perform context switch */
   /* When we return from swtch, the
event has occurred */
   spin_lock (s); /* acquire the spin
lock again */
   return;
)
```

```
void do_signal (condition *c)
/*Wakeup one thread waiting on the  condition*/
{
   spin_lock (&c->listLock);
   remove one thread from linked list, if it is nonempty;
   spin_unlock (&c->listLock);
   if a thread was removed from the list, make it
      runnable;
   return;
}
void do broadcast (condition *c)
/*Wakeup al lthreads waiting on the condition*/
{
  spin_lock (&c->listLock);
  while (linked list is nonempty) {
    remove a thread from linked list;
    make it runnable;
  }
  spin_unlock (&c->listLock);
}
```

Semaphore implementation using condition variables?

- Is this possible?
- Can we try it?

```
typedef struct semaphore {
  //something
  condition c;
}semaphore;
```

- Now write code for semaphore P() and V()

Classical Synchronization Problems

Bounded-Buffer Problem

- Producer and consumer processes
  - N buffers, each can hold one item
- Producer produces ‘items’ to be consumed by consumer , in the bounded buffer
- Consumer should wait if there are no items
- Producer should wait if the ‘bounded buffer’ is full

Bounded-Buffer Problem: solution with semaphores

- Semaphore mutex initialized to the value 1
- Semaphore full initialized to the value 0
- Semaphore empty initialized to the value N

Bounded-buffer problem

The structure of the producer process

```
do  {
        //   produce an item in nextp
       wait (empty);
       wait (mutex);
       //  add the item to the  buffer
        signal (mutex);
        signal (full);
} while (TRUE);
```

The structure of the Consumer process

```
do  {
        wait (full);
        wait (mutex);
        //  remove an item from
          //  buffer to nextc
         signal (mutex);
         signal (empty);
        // consume item in nextc
} while (TRUE);
```

Bounded buffer problem

- **Example : pipe()**
- **Let’s see code of pipe in xv6 – a solution using sleeplocks**

Readers-Writers problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do not perform any updates
  - Writers – can both read and write
- Problem – allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time
- Several variations of how readers and writers are treated – all involve priorities
- Shared Data
  - Data set
  - Semaphore mutex initialized to 1
  - Semaphore wrt initialized to 1
  - Integer readcount initialized to 0

The structure of a writer process

```
do {
      wait (wrt) ;
//    writing is performed
     signal (wrt) ;
} while (TRUE);
```

Readers-Writers problem

The structure of a reader process

```
do {
                wait (mutex) ;
                readcount ++ ;
                if (readcount == 1)
                      wait (wrt) ;
                signal (mutex)
               // reading is performed
                wait (mutex) ;
                readcount  - - ;
                if (readcount  == 0)
                       signal (wrt) ;
                signal (mutex) ;

 } while (TRUE);
```

Readers-Writers Problem Variations

- First variation – no reader kept waiting unless writer has permission to use shared object
- Second variation – once writer is ready, it performs write asap
- Both may have starvation leading to even more variations
- Problem is solved on some systems by kernel providing reader-writer locks

Reader-write lock

- A lock with following operations on it
  - Lockshared()
  - Unlockshared()
  - LockExcl()
  - UnlockExcl()
- Possible additions
  - Downgrade() -> from excl to shared
  - Upgrade() -> from shared to excl

Code for reader-writer locks

```
struct rwlock {
   int nActive; /* num of active readers, or-1 if a writer is active */
   int nPendi ngReads;
   int nPendingWrites;
   spinlock_t sl;
   condition canRead;
   condition canWrite;
);
```

```
void lockShared {struct rwlock *r)
(
   spin_lock {&r->sl);
   r->nPendingReads++;
   if (r->nPendingWrites > O)
      wait (&r->canRead, &r->sl ); /*don'tstarve
         writers */
   while {r->nActive < 0) /* someone has
        exclusive lock */
        wait (&r->canRead, &r->sl);
   r->nActive++;
   r->nPendingReads--;
   spin_unlock (&r->sl);
)
```

Code for reader-writer locks

```
void unlockShared (struct rwlock
*r)
{
    spin_lock (&r->sl);
    r->nActive--;
    if (r->nActive == O) {
        spin_unlock (&r->sl);
        do signal (&r->canWrite);
    } else
        spin_unlock (&r->M);
)
```

```
void lockExclusive (struct rwlock
*r)
(
    spin_lock (&r->sl);
    r->nPendingWrltes++;
    while (r->nActive)
        wait (&r->canWrite, &r->sl);
    r->nPendingWrites--;
    r->nActive = -1;
    spin_unlock (&r->sl);
}
```

Code for reader-writer locks

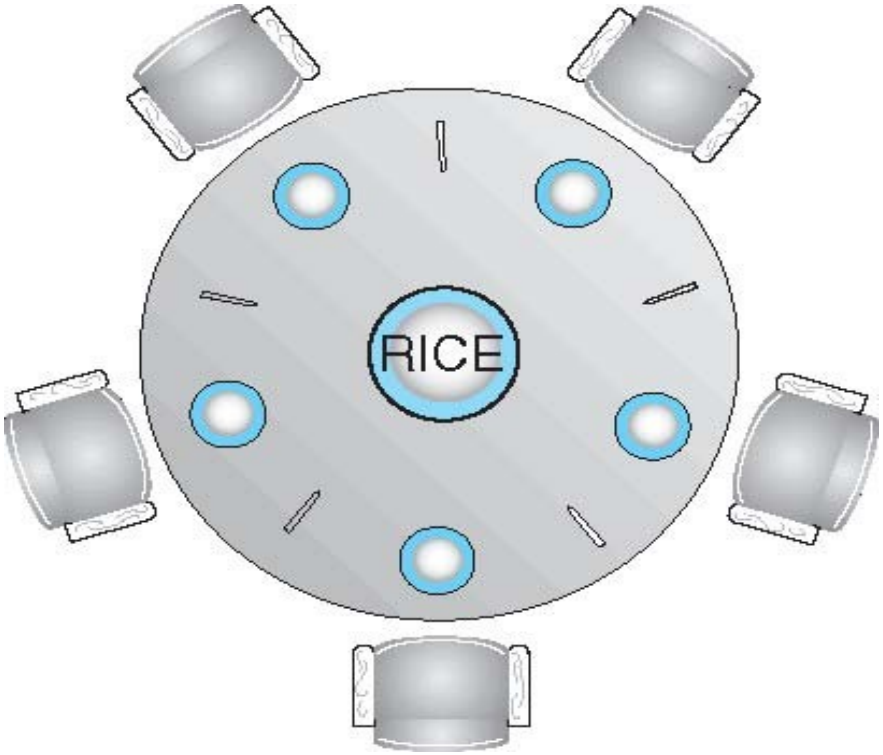
```
void unlockExclusive (struct rwlock *r){
   boolean t wakeReaders;
   spin_lock (&r->sl);
   r->nActive = O;
   wakeReaders = (r->nPendingReads != 0);
   spin_unlock (&r->sl);
   if (wakeReaders)
        do broadcast (&r->canRead); /* wake
allreaders */
  else
      do_signal (&r->canWrite);
       /*wakeasinglewri  r */
}
```

Try writing code for downgrade and upgrade

Try writing a reader-writer lock using semaphores!

Dining-Philosophers Problem

- Philosophers spend their lives thinking and eating
- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
- In the case of 5 philosophers
  - Shared data
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1



Dining philosophers: One solution

The structure of Philosopher i:

```
do  {
        wait ( chopstick[i] );
       wait ( chopStick[ (i + 1) % 5] );
       //  eat
       signal ( chopstick[i] );
       signal (chopstick[ (i + 1) % 5] );
       //  think
} while (TRUE);
```

What is the problem with this algorithm?

Dining philosophers: Possible approaches

- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up her chopsticks only if both chopsticks are available
  - to do this, she must pick them up in a critical section
- Use an asymmetric solution
  - that is, an odd-numbered philosopher picks up first her left chopstick and then her right chopstick
  - whereas an even-numbered philosopher picks up her right chopstick and then her left chopstick.

Other solutions to dining philosopher's problem

- **Using higher level synchronization primitives like 'monitors'**
-

Practical Problems

Lost Wakeup problem

- **The sleep/wakeup mechanism does not function correctly on a multiprocessor.**
- **Consider a potential race:**
  - **Thread T1 has locked a resource R1.**
  - **Thread T2, running on another processor, tries to acquire the resource, and finds it locked.**
  - **T2 calls sleep() to wait for the resource.**
  - **Between the time T2 finds the resource locked and the time it calls s]eep (), T1 frees the resource and proceeds to wake up all threads blocked on it.**
  - **Since T2 has not yet been put on the sleep queue, it will miss the wakeup.**
  - **The end result is that the resource is not locked, but T2 is blocked waiting for it to be unlocked.**
  - **If no one else tries to access the resource, T2 could block indefinitely.**
  - **This is known as the lost wakeup problem,**
- **Requires some mechanism to combine the test for the resource and the call to sleep () into a single atomic operation.**

Lost Wakeup problem

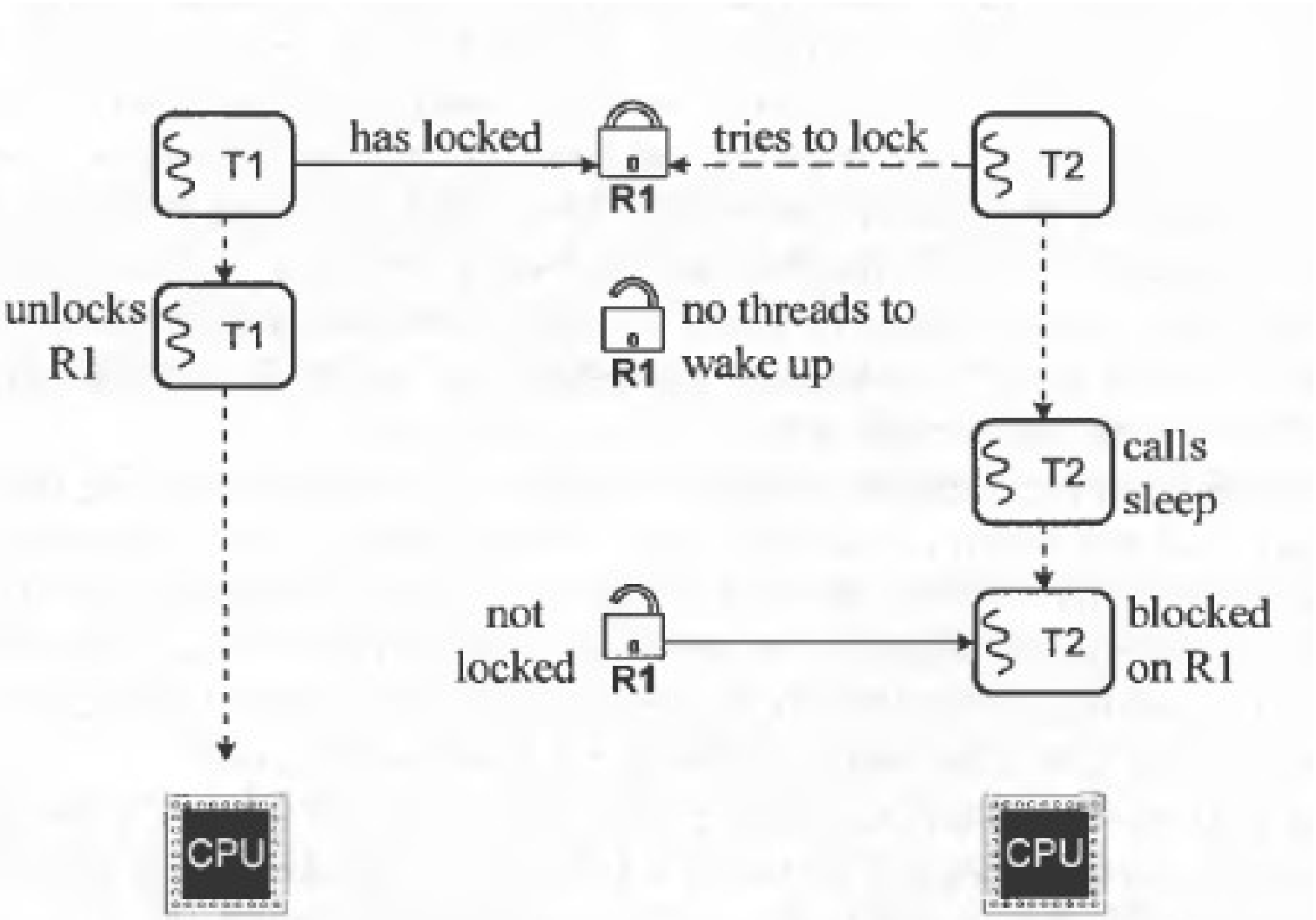


Figure 7-6. The lost wakeup problem.

Thundering herd problem

- **Thundering Herd problem**
  - On a multiprocessor, if several threads were locked the resource
  - Waking them all may cause them to be simultaneously scheduled on different processors
  - and they would all fight for the same resource again.
- **Starvation**
  - Even if only one thread was blocked on the resource, there is still a time delay between its waking up and actually running.
  - In this interval, an unrelated thread may grab the resource causing the awakened thread to block again. If this happens frequently, it could lead to starvation of this thread.
  - This problem is not as acute on a uniprocessor, since by the time a thread runs, whoever had locked the resource is likely to have released it.

Case Studies

Linux Synchronization

- Prior to kernel Version 2.6, disables interrupts to implement short critical sections
- Version 2.6 and later, fully preemptive
- Linux provides:
  - semaphores
  - spinlocks
  - reader-writer versions of both
  - Atomic integers
- On single-cpu system, spinlocks replaced by enabling and disabling kernel preemption

Linux Synchronization

- **Atomic variables**

  atomic_t is the type for atomic integer

- **Consider the variables**

  atomic_t counter;

  int value;

Atomic Operation	*Effect*
`atomic_set(&counter,5);`	`counter = 5`
`atomic_add(10,&counter);`	`counter = counter + 10`
`atomic_sub(4,&counter);`	`counter = counter - 4`
`atomic_inc(&counter);`	`counter = counter + 1`
`value = atomic_read(&counter);`	`value = 12`

Pthreads synchronization

- **Pthreads API is OS-independent**
- **It provides:**
  - **mutex locks**
  - **condition variables**
- **Non-portable extensions include:**
  - **read-write locks**
  - **spinlocks**

Synchronization issues in xv6 kernel

Difference approaches

- **Pros and Cons of locks**
  - **Locks ensure serialization**
  - **Locks consume time !**
- **Solution – 1**
  - **One big kernel lock**
  - **Too enefficient**
- **Solution – 2**
  - **One lock per variable**
  - **Often un-necessary, many data structures get manipulated in once place, one lock for all of them may work**
- **Problem: ptable.lock for the entire array and every element within**
  - **Alternatively: one lock for array, one lock per array entry**

Three types of code

- **System calls code**
  - **Can it be interruptible?**
  - **If yes, when?**
- **Interrupt handler code**
  - **Disable interrupts during interrupt handling or not?**
  - **Deadlock with iderw ! - already seen**
- **Process's user code**
  - **Ignore. Not concerned with it now.**

Interrupts enabling/disablilng in xv6

- Holding every spinlock disables interrupts!
- System call code or Interrupt handler code won't be interrupted if
  - The code path followed took at least once spinlock !
  - Interrupts disabled only on that processor!
- Acquire calls pushcli() before xchg()
- Release calls popcli() after xchg()

Memory ordering

- Compiler may generate machine code for out-of-order execution !
- Processor pipelines can also do the same!
- This often improves performance
- Compiler may reorder 4 after 6 --> Troble!
- Solution: Memory barrier
  - __sync_synchronize(), provided by GCC
  - Do not reorder across this line
  - Done only on acquire and release()

- **Consider this**

```
1)l = malloc(sizeof *l);
2)l->data = data;
3)acquire(&listlock);
4)l->next = list;
5)list = l;
6)release(&listlock);
```

Lost Wakeup?

- Do we have this problem in xv6?
- Let's analyze again!
  - The race in acquiresleep()'s call to sleep() and releasesleep()
- T1 holding lock, T2 willing to acquire lock
  - Both running on different processor
  - Or both running on same processor
  - What happens in both scenarios?
- Introduce a T3 and T4 on each of two different processors. Now how does the scenario change?
- See page 69 in xv6 book revision-11.

Code of sleep()

```
if(lk != &ptable.lock){
   acquire(&ptable.lock);
   release(lk);
}
```

- Why this check?
- Deadlock otherwise!
- Check: wait() calls with ptable.lock held!

Exercise question : 1

Sleep has to check lk != &ptable.lock to avoid a deadlock

Suppose the special case were eliminated by replacing

```
if(lk != &ptable.lock){
acquire(&ptable.lock);
release(lk);
}
```

with

```
release(lk);
acquire(&ptable.lock);
```

Doing this would break sleep. How?

`

bget() problem

- bget() panics if no free buffers!
- Quite bad
- Should sleep !
- But that will introduce many deadlock problems. Which ones ?

iget() and ilock()

- iget() does no hold lock on inode
- Ilock() does
- Why this separation?
  - Performance? If you want only “read” the inode, then why lock it?
- What if iget() returned the inode locked?

Interesting cases in namex()

```
while((path = skipelem(path, name)) !
= 0){
  ilock(ip);
  if(ip->type != T_DIR){
    iunlockput(ip);
    return 0;
  }
  if(nameiparent && *path == '\0'){
    // Stop one level early.
    iunlock(ip);
    return ip;
  }
```

```
  if((next = dirlookup(ip, name, 0)) == 0){
    iunlockput(ip);
    return 0;
  }
  iunlockput(ip);
  ip
 }
```

--> only after obtaining next from dirlookup() and iget() is the lock released on ip;

-> lock on next obtained only after releasing the lock on ip. Deadlock possible if next was “.”

Xv6
Interesting case of holding and releasing ptable.lock in scheduling

One process acquires, another releases!

Giving up CPU

- A process that wants to give up the CPU
  - must acquire the process table lock ptable.lock
  - release any other locks it is holding
  - update its own state (proc->state),
  - and then call sched()
- Yield follows this convention, as do sleep and exit
- Lock held by one process P1, will be released another process P2 that starts running after sched()
  - remember P2 returns either in yield() or sleep()
  - In both, the first thing done is releasing ptable.lock

Interesting race if ptable.lock is not held

- Suppose P1 calls yield()
- Suppose yield() does not take ptable.lock
  - Remember yield() is for a process to give up CPU
- Yield sets process state of P1 to RUNNABLE
- Before yield's sched() calls swtch()
- Another processor runs scheduler() and runs P1 on that processor
- Now we have P1 running on both processors!
- P1 in yield taking ptable.lock prevents this

Homework

- **Read the version-11 textbook of xv6**
- **Solve the exercises!**