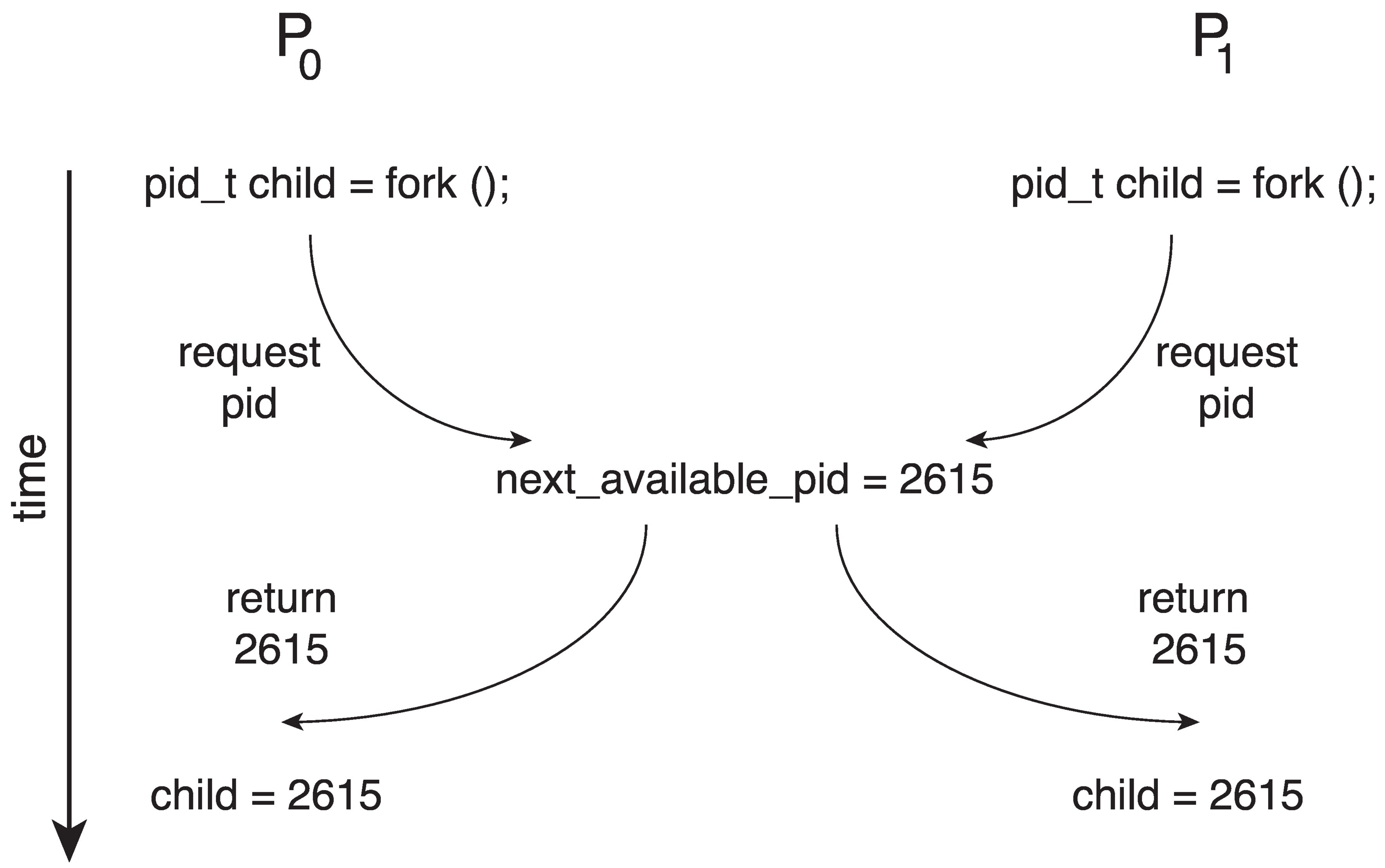
Concurrency : Critical Sections

03/09/2021

Professor Amanda Bienz

Textbook pages 257-275

Race Condition



- Unless there is a mechanism to prevent P0 and P1 from accessing the variable next_available_pid, the same pid could be assigned to two processes!

Critical Section Problem

- Consider system of ***n*** processes {P0, P1, …, PN-1}
- Each process has **critical section** segment of code
    - Process mad 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 :** design protocol to solve this
- Each process must ask permission to enter critical section

Locks : The Basic Idea

- Ensure that any critical section executes as if it were a single **atomic instruction**
  - An example: the canonical update of a shared variable

```
balance = balance + 1;
```

  - Add some code around the critical section

```
lock_t mutex; // some globally-allocated lock 'mutex'
…
lock(&mutex);
balance = balance + 1;
unlock(&mutex);
```

Locks : The Basic Idea

- Lock variable holds **the state of** the lock
  - **Available** (unlocked, free) : no thread holds the lock
  - **Aquired** (locked, held) : exactly one thread holds the lock and presumably is in a critical section

The semantics of the lock()

- lock()
  - Try to acquire the lock
  - If **no other thread holds** the lock, the thread will acquire the lock
  - Enter the critical section
    - This thread is said to be **the owner of the lock**
  - Other threads are prevented from entering the critical section while the first thread that holds the lock is in there

Pthread Locks - mutex

- The name that the POSIX library uses for a lock
  - Used to provide mutual exclusion between threads

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

Pthread_mutex_lock(&lock); // wrapper for pthread_mutex_lock()
balance = balance + 1;
Pthread_mutex_unlock(&lock);
```

- We may be using different locks to protect different variables
  - Increase concurrency (a more fine-grained approach)

Building A Lock

- **Efficient locks** provide mutual exclusion at low cost
- Building a lock needs some help from hardware and the OS

Evaluating C.S. Solutions - Basic Criteria

- Correctness : Mutual exclusion and progress
    - **Mutual exclusion :** does the lock work, preventing multiple threads from entering a critical section?
    - **Progress :** if no one else is in the critical section, do we get in?
- Fairness : **Bounded Waiting**
    - If multiple threads are trying to get in, does each get in in a reasonable number of attempts? (Can a thread starve?)
- Also care about performance if we're going to do this often (and generally we are!)

Controlling Interrupts

- Disable interrupts for critical sections
  - One of the earliest solutions used to provide mutual exclusion
  - Invented for **single processor** systems

```
void lock() {
    DisableInterrupts();
}
void unlock() {
    EnableInterrupts();
}
```

Controlling Interrupts : Problems

- Requires too much **trust** in applications
  - Greedy (or malicious) program could monopolize the processor
- Does not work on **multiprocessors**
- Code that masks or unmasks interrupts is executed **slowing** by modern CPUs

Software Solution 1

- Two process solution
- Assume **load** and **store** instructions are ***atomic*** : cannot be interrupted
- The processes share the variable **int turn**
- The variable **turn** indicates whose turn it is to enter the critical section
- Initially, the value of **turn** is set to *i*

Algorithm for Process Pi

```
while (true){

    while (turn = = j);

    /* critical section */

    turn = j;

    /* remainder section */

}
```

Does this work well?

- Mutual exclusion?
- Progress requirement?
- Bounded-waiting requirement?

Peterson's Solution

- Two process solution, share two variables:
  - **int turn**
  - **boolean flag[2]**
- Variable **turn** indicates whose turn it is to enter critical section
- The **flag** array is used to indicate if a process is ready to enter the critical section
  - flag[i] = true implies Pi is ready

Algorithm for Process Pi

```
while (true){

   flag[i] = true;
   turn = j;
   while (flag[j] && turn = = j)
      ;

      /* critical section */

   flag[i] = false;

   /* remainder section */

}
```

Does this work well?

- Mutual exclusion?
- Progress requirement?
- Bounded-waiting requirement?

Why is hardware support needed?

- First attempt : using a flag denoting whether the lock is held or not
    - This code has problems

```
typedef struct __lock_t { int flag; } lock_t;

void init(lock_t *mutex) {
// 0 → lock is available, 1 → held
mutex->flag = 0;
}

void lock(lock_t *mutex) {
while (mutex->flag == 1)  // TEST the flag
    ;  // spin-wait (do nothing)
mutex->flag = 1;  // now SET it !
}

void unlock(lock_t *mutex) {
mutex->flag = 0;
}
```

Why is hardware support needed? (cont.)

- Problem 1 : No mutual exclusion (assume flag = 0 to begin)

Thread1	Thread2
call `lock()`	
`while (flag == 1)`	
interrupt: switch to Thread 2	
	call `lock()`
	`while (flag == 1)`
	`flag = 1;`
	interrupt: switch to Thread 1
`flag = 1;` // set flag to 1 (too!)	

- Problem 2 : Spin-waiting wastes time waiting for another thread
- So, we need an atomic instruction supported by Hardware!
  - Test-and-set instruction, also known as an atomic exchange

Test and Set (Atomic Exchange)

- An instruction to support the creation of simple locks

```
int TestAndSet(int *ptr, int new) {
int old = *ptr;   // fetch old value at ptr
*ptr = new;   // store 'new' into ptr
return old;   // return the old value
}
```

- Return(*testing*) old value pointed to by the ptr
- Simultaneously update(*setting*) said value to new
- This sequence of operations is **performed atomically**

A Simple Spin Lock using test-and-set

```
typedef struct __lock_t {
int flag;
} lock_t;

void init(lock_t *lock) {
// 0 indicates that lock is available,
// 1 that it is held
lock->flag = 0;
}

void lock(lock_t *lock) {
while (TestAndSet(&lock->flag, 1) == 1)
    ;    // spin-wait
}

void unlock(lock_t *lock) {
lock->flag = 0;
}
```

- Note : to work correctly on a **single processor**, it requires a preemptive scheduler

Evaluating Spin Locks

- Correctness : yes
  - The spin lock only allows a single thread to enter the critical section
  - If no one is waiting, a thread will get in
- Fairness : no
  - Spin locks don’t provide any fairness guarantees
  - Indeed, a thread spinning may spin forever
- Performance :
  - Single CPU : performance overheads can be quite painful
  - If number of threads roughly equals the number of CPUs, spin locks work reasonably well

Compare-And-Swap

- Test whether the value at the address(ptr) is equal to expected
  - If so, update the memory location pointed to by ptr with the new value
  - In either case, return the actual value at that memory location

```
int CompareAndSwap(int *ptr, int expected, int new) {
int actual = *ptr;
if (actual == expected)
    *ptr = new;
return actual;
}
```

Compare-and-Swap hardware atomic instruction (C-style)

```
void lock(lock_t *lock) {
while (CompareAndSwap(&lock->flag, 0, 1) == 1)
    ; // spin
}
```

Spin lock with compare-and-swap

Load-Linked and Store-Conditional

```c
int LoadLinked(int *ptr) {
return *ptr;
}

int StoreConditional(int *ptr, int value) {
if (no one has updated *ptr since the LoadLinked to this address) {
    *ptr = value;
    return 1; // success!
} else {
    return 0; // failed to update
}
}
```

Load-linked And Store-conditional

- The store-conditional only succeeds if no intermittent store to the address has taken place
  - Success : return 1 and update the value at ptr to value
  - Fail : the value at ptr is not updated and 0 is returned

Load-Linked and Store-Conditional (Cont.)

```
void lock(lock_t *lock) {
while (1) {
    while (LoadLinked(&lock->flag) == 1)
        ; // spin until it's zero
    if (StoreConditional(&lock->flag, 1) == 1)
        return; // if set-it-to-1 was a success: all done
            otherwise: try it all over again
}
}

void unlock(lock_t *lock) {
    lock->flag = 0;
}
```

Using LL/SC To Build A Lock

```
void lock(lock_t *lock) {
while (LoadLinked(&lock->flag)||!StoreConditional(&lock->flag, 1))
    ; // spin
}
```

A more concise form of the `lock()` using LL/SC

Fetch-and-Add

- Atomically increment a value while returning the old value at a particular address

```
int FetchAndAdd(int *ptr) {
    int old = *ptr;
    *ptr = old + 1;
    return old;
}
```

Fetch-And-Add Hardware atomic instruction (C-style)

Ticket Lock

- Ticket lock can be built with fetch-and-add
  - Ensure progress for all threads : **fairness**

```
typedef struct __lock_t {
    int ticket;
    int turn;
} lock_t;

void lock_init(lock_t *lock) {
    lock->ticket = 0;
    lock->turn = 0;
}

void lock(lock_t *lock) {
    int myturn = FetchAndAdd(&lock->ticket);
    while (lock->turn != myturn)
            ; // spin
}
void unlock(lock_t *lock) {
    FetchAndAdd(&lock->turn);
}
```

So Much Spinning

- Hardware-based spin locks are simple and they work
- In some cases, these solution can be quite inefficient
  - Any time a thread gets caught spinning, it wastes and entire time splice doing nothing but checking a value

How To Avoid *Spinning*?
We'll need OS Support too!

A Simple Approach : Just Yield

- When you are going to spin, give up the CPU to another thread
  - OS system call moves the caller from the running state to the ready state
  - Cost of a **context switch** can be substantial and the **starvation** problem exists

```
void init() {
    flag = 0;
}

void lock() {
    while (TestAndSet(&flag, 1) == 1)
        yield(); // give up the CPU
}

void unlock() {
    flag = 0;
}
```

Lock with Test-and-set and Yield

Using Queues : Sleeping Instead of Spinning

- Queue to keep track of which threads are waiting to enter the lock
- park() : put a calling thread to sleep
- unpark(threadID) : wake a particular thread as designated the threadID

Using Queues : Sleeping Instead of Spinning

```
typedef struct __lock_t { int flag; int guard; queue_t *q; } lock_t;

void lock_init(lock_t *m) {
    m->flag = 0;
    m->guard = 0;
    queue_init(m->q);
}

void lock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1)
        ; // acquire guard lock by spinning
    if (m->flag == 0) {
        m->flag = 1; // lock is acquired
        m->guard = 0;
    } else {
        queue_add(m->q, gettid());
        m->guard = 0;
        park();
    }
}
…
```

Lock With Queues, Test-and-set, Yield, And Wakeup

Using Queues : Sleeping Instead of Spinning

```
•   void unlock(lock_t *m) {
•       while (TestAndSet(&m->guard, 1) == 1)
•           ; // acquire guard lock by spinning
•       if (queue_empty(m->q))
•           m->flag = 0; // let go of lock; no one wants it
•       else
•           unpark(queue_remove(m->q)); // hold lock (for next thread!)
•       m->guard = 0;
•   }
```

Lock With Queues, Test-and-set, Yield, And Wakeup (Cont.)

Wakeup / Waiting Race

- In case of releasing the lock (thread A) just before the call to park() (thread B) : thread B would potentially sleep forever
- Solaris solves this problem by adding a third system call : setpark()
  - By calling this routine, a thread can indicate it is about to park
  - If it happens to be interrupted and another thread calls unpack before park is actually called, the subsequent park returns immediately instead of sleeping

```
queue_add(m->q, gettid());
setpark(); // new code
m->guard = 0;
park();
```

Code modification inside of `lock()`

Futex

- Linux provides a futex (similar to Solaris's park and unpack)
  - futex_wait(address, expected)
    - Put calling thread to sleep
    - If value at address is not equal to expected, call returns immediately
  - futex_wake(address)
    - Wake one thread that is waiting on the queue

Futex (Cont.)

- Snippet from lowlevellock.h in the nptl library
  - The high bit of the integer v : track whether the lock is held or not
  - All other bits : number of waiters

```
void mutex_lock(int *mutex) {
    int v;
    /* Bit 31 was clear, we got the mutex (this is the fastpath) */
    if (atomic_bit_test_set(mutex, 31) == 0)
        return;
    atomic_increment(mutex);
    while (1) {
        if (atomic_bit_test_set(mutex, 31) == 0) {
            atomic_decrement(mutex);
            return;
        }
        /* We have to wait now. First make sure the futex value
           we are monitoring is truly negative (i.e. locked). */
        v = *mutex;
        …
```

Linux-based Futex Locks

Futex (Cont.)

```
                if (v >= 0)
                        continue;
                futex_wait(mutex, v);
        }
}

void mutex_unlock(int *mutex) {
        /* Adding 0x80000000 to the counter results in 0 if and only if
           there are not other interested threads */
        if (atomic_add_zero(mutex, 0x80000000))
                return;
        /* There are other threads waiting for this mutex,
           wake one of them up */
        futex_wake(mutex);
}
```

Linux-based Futex Locks (Cont.)

Two-Phase Locks

- A two-phase lock realizes that spinning can be useful if the lock is about to be released
    - First phase :
        - Lock spins for awhile, hoping that it can acquire the lock
        - If the lock is not acquired during the first spin phase, a second spin phase is entered
    - Second phase :
        - The caller is put to sleep
        - The caller is only woken up when the lock becomes free later

Semaphores

- More sophisticated way to synchronize processes than mutex locks
- Semaphore **S** : integer variable
- Can only be accessed via two atomic operations
  - **wait()**
  - **signal()**

Wait and Signal Operations

- Defintion of wait() operation:

```
wait(S)
{
    while (S <= 0)
        ; // busy wait
    S—;
}
```

- Definition of signal() operation:

```
signal(S)
{
    S++;
}
```

Semaphore (Cont.)

- **Counting semaphore :** integer value can range over an unrestricted domain
- **Binary semaphore :** integer value can range only between 0 and 1
  - Same as a **mutex lock**
- Can implement a counting semaphore S as a binary semaphore
- With semaphores, we can solve various synchronization problems

Critical Sections and Semaphores

- Create a semaphore **“mutex”** initialized to 1

  wait(mutex);
  Critical Section
  signal(mutex);

Semaphores and Synchronization

- Two processes : P1 and P2
- Two statements : S1 (by P1) and S2 (by P2)
- S1 must happen before S2

Semaphores and Synchronization

- Two processes : P1 and P2
- Two statements : S1 (by P1) and S2 (by P2)
- S1 must happen before S2

P1 :
    S1;
    signal(synch);
P2 :
    wait(synch);
    S2;

Semaphore Implementation

- Must guarantee that no two processes can execute the ***wait()*** and ***signal()*** on the same semaphore at the same time
- Thus, implementation becomes the critical section problem where **wait** and **signal** code are place 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 Implementation with no Busy Waiting

- With each semaphore, there is an associated waiting queue
- Each entry in waiting queue has two data items:
  - Value (integer)
  - Pointer to next record in list
- Two operations:
  - **Block :** place process invoking the operating on the appropriate waiting queue
  - **Wakeup :** remove one of the processes in the waiting queue and place it in the ready queue

Implementation with no Busy Waiting

- Waiting queue

```
typedef struct
{
    int value;
    struct process * list;
} semaphore;
```

Implementation with no Busy Waiting

- *Wait (semaphore * S)*

```
{
    S->value--;
    if (S->value < 0)
    {
        add this process to S->list;
        block();
    }
}
```

Implementation with no Busy Waiting

- *Signal (semaphore* S)*

```
{
    S->value++;
    if (S->value <= 0)
    {
        remove a process P from S->list;
        wakeup(P);
    }
}
```

Problems with Semaphores

- Incorrect use of semaphore operations:
  - signal(mutex) … wait(mutex)
  - wait(mutex) … wait(mutex)
  - Omitting wait(mutex) and or single(mutex)

Reading

- Deadlocks : pg 283
- Synchronization problems : pg 289-294
- POSIX : pg 299-303
- Alternative approaches : pg 311-313