Operating Systems

Christopher Kruegel
Department of Computer Science
UC Santa Barbara
http://www.cs.ucsb.edu/~chris/
Inter-process Communication and Synchronization

- Processes/threads may need to exchange information
- Processes/threads should not get in each other’s way
- Processes/threads should access resources in the right sequence
- Need to coordinate the activities of multiple threads
- Need to introduce the notion of *synchronization operations*
- These operations allow threads to control the timing of their events relative to events in other threads
Asynchrony and Race Conditions

• Threads need to deal with asynchrony
• Asynchronous events occur arbitrarily during thread execution:
  – An interrupt causes transfer being taken away from the current thread to the interrupt handler
  – A timer interrupt causes one thread to be suspended and another one to be resumed
  – Two threads running on different CPUs read and write the same memory
• Threads must be designed so that they can deal with such asynchrony
• (If not, the code must be protected from asynchrony)
Race Conditions

- Two threads, A and B, need to insert objects into a list, so that it can be processed by a third thread, C

- Both A and B
  - Check which is the first available slot in the list
  - Insert the object in the slot

- Everything seems to run fine until...
  - Thread A finds an available slot but gets suspended by the scheduler
  - Thread B finds the same slot and inserts its object
  - Thread B is suspended
  - Thread is resumed and inserts the object in the same slot

- B’s object is lost!
Critical Regions and Mutual Exclusion

- The part of the program where shared memory is accessed is called a *critical region* (or *critical section*)
- Critical regions should be accessed in *mutual exclusion*

- **Solution: Synchronization**
  - No two processes may be simultaneously inside the same critical region
  - No process running outside the critical region should block another process
  - No process should wait forever to enter its critical region
  - No assumptions can be made about speed/number of CPUs
Entering and Exiting Critical Regions

Entry Code

CS

Exit Code
Mutual Exclusion With Busy Waiting

• First solution: Disable interrupts when in critical region
  – What if the process “forgets” to re-enable interrupts?
  – What if there are multiple CPUs?

• Second solution: a lock variable
  – Test if lock is 0
  – If not, loop on check until 0
  – When lock is 0, set it to 1 and start critical region
  – Set it back to 0 when finished
  – ... do you see any problem?

• Third solution: strict alternation
Taking Turns...

Initially set to 0

```
turn = 0;
while (turn != 0) { }
```

```
turn = 1;
while (turn != 1) { }
```

```
turn = 0;
while (turn != 0) { }
```

```
turn = 1;
while (turn != 1) { }
```
Taking Turns...

• What if thread 0 is much faster than thread 1?
• Thread 0 may be waiting for its turn even if thread 1 is outside the critical region
• We said:
  – No process running outside the critical region should block another process

• Need for something better: Peterson’s algorithm
Peterson’s Algorithm

Process 0

interested_0 = TRUE;
turn = 0;
while (interested_1 == TRUE 
&& turn == 0) { };

interested_0 = FALSE;

Process 1

interested_1 = TRUE;
turn = 1;
while (interested_0 == TRUE 
&& turn == 1) { };

interested_1 = FALSE;
Test And Set Lock Instruction

- If the hardware (that is, the CPU) provides an atomic way of testing and setting a lock, life is easier
- TSL RX, LOCK
  - Reads contents of address LOCK into RX
  - Stores a nonzero value into location LOCK
- Now back to lock variables
  enter: TSL RX, LOCK
    CMP RX, #0
    JNE enter
    RET
  leave: MOV LOCK, #0
    RET
Sleep and Wakeup

- Busy waiting is a waste of CPU
- Need to provide a mechanism so that a thread can suspend when a critical region cannot be entered
  - Sleep() blocks the thread
  - Wakeup() resumes a thread
- Classical problem: Producer and Consumer communicating through a set of buffers
- Number of buffers (N) is limited
  - 0 buffers available  ⇒ consumer must wait
  - N buffers filled  ⇒ producer must wait
Producer/Consumer Problem

Count=0 → Consumer must wait

Count=N → Producer must wait
Producer/Consumer

in = 0;
out = 0;
count = 0;

Producer:
while (1) {
    item = produce_item();
    if (count == N) sleep();
    buff[in]=item;
    in=(in+1) % N;
    count=count+1;
    if (count == 1)
        wakeup(consumer)
}

Consumer:
while (1) {
    if (count == 0) sleep();
    item = buff[out];
    out=(out+1) % N;
    count = count-1;
    if (count == N-1)
        wakeup(producer)
    consume_item(item);
}
Missing the Wake Up Call

- Buffer is empty
- Consumer reads counter and gets 0
- Before falling asleep, there is a context switch to the Producer thread
- Producer inserts item and, since count==1, sends a wakeup
- Consumer is not sleeping and wakeup signal gets lost
- Control returns to Consumer that falls asleep (the check on count has been done before)
- Producer continues until count reaches N and then falls asleep: Game Over...
Semaphores

- Edward Dijkstra suggested to use an integer variable to count the number of wakeups issued
- New type, the Semaphore
  - Semaphore(count) creates and initializes to count
  - P() or down()
    - If the counter is greater than 0 then decrements the counter and returns
    - If counter = 0 the process suspends. When it wakes up decrements the counter and returns
  - V() or up()
    - Increments the counter
    - If there are any process waiting on the semaphore one is woken up
    - Returns
  - down() and up() are ATOMIC operations
Semaphores and Mutual Exclusion

Semaphore with count = 1, initial value 1

mutex down();

mutex down();

mutex up();

mutex up();
Threads - Revisited

```c
1: int i;
2: Semaphore sema;
3:
4: f()
5: {
6:     printf("i is %d\n", i);
7: }
8:
9: int main(int argc, char **argv)
10: {
11:     .. (do stuff here) ..
12:     P(sema);
13:     i = get_input();
14:     f();
15:     V(sema);
16:     return 0;
17: }
```
Threads - Revisited

1: int i;
2: Semaphore sema;
3:
4: f()
5: {
6:   printf("i is %d\n", i);
7: }
8:
9: int main(int argc, char **argv)
10: {
11:   .. (do stuff here) ..
12:   P(sema);
13:   i = get_input();
14:   f();
15:   V(sema);
16:   return 0;
17: }
Threads - Revisited

1: int i;
2: Semaphore sema;
3: 
4: f()
5: {
6:   printf(“i is %d\n”, i);
7: }
8: 
9: int main(int argc, char **argv)
10: {
11:   .. (do stuff here) ..
12:   P(sema);
13:   i = get_input();
14:   f();
15:   V(sema);
16:   return 0;
17: }
1: int i;
2: Semaphore sema;
3: 
4: f()
5: {
6:   printf("i is %d\n", i);
7: }
8: 
9: int main(int argc, char **argv)
10: {
11:   .. (do stuff here) ..
12:   P(sema);
13:   i = get_input();
14:   f();
15:   V(sema);
16:   return 0;
17: }

Threads - Revisited

Address Space

i = 42

sema
Counter: 0
Queue:

Registers

PC = 14
T1
PC = 11
T2

Stack

Running

Context
Threads - Revisited

1: int i;
2: Semaphore sema;
3:
4: f()
5: {
6:   printf("i is %d\n", i);
7: }
8:
9: int main(int argc, char **argv)
10: {
11:   .. (do stuff here) ..
12:   P(sema);
13:   i = get_input();
14:   f();
15:   V(sema);
16:   return 0;
17: }

Address Space

i = 42

Registers

PC = 14

Stack

PC = 11

Running

Context

sema

Counter: 0

Queue:
Threads - Revisited

1: int i;
2: Semaphore sema;
3: 
4: f()
5: {
6:   printf("i is %d\n", i);
7: } 
8: 
9: int main(int argc, char **argv)
10: {
11:   .. (do stuff here) ..
12:   P(sema);
13:   i = get_input();
14:   f();
15:   V(sema);
16:   return 0;
17: }

Address Space

Registers

Stack

Running

Context

sema
Counter: 0
Queue:
Threads - Revisited

1: int i;
2: Semaphore sema;
3:
4: f()
5: {
6:   printf("i is %d\n", i);
7: }
8:
9: int main(int argc, char **argv)
10: {
11:   .. (do stuff here) ..
12:   P(sema);
13:   i = get_input();
14:   f();
15:   V(sema);
16:   return 0;
17: }

Address Space

i = 42

Registers

PC = 14

PC = 13

Stack

T1

T2

Blocked

sema

Counter: -1

Queue: T2

Context
1: int i;
2: Semaphore sema;
3: 
4: f()
5: {
6:     printf("i is %d\n", i);
7: }
8: 
9: int main(int argc, char **argv)
10: {
11:     .. (do stuff here) ..
12:     P(sema);
13:     i = get_input();
14:     f();
15:     V(sema);
16:     return 0;
17: }
Threads - Revisited

1: int i;
2: Semaphore sema;
3: 
4: f()
5: {
6:     printf("i is %d\n", i);
7: }
8: 
9: int main(int argc, char **argv)
10: {
11:     .. (do stuff here) ..
12:     P(sema);
13:     i = get_input();
14:     f();
15:     V(sema);
16:     return 0;
17: }

Address Space

i = 42

Registers

PC = 6
PC = 13

Stack

15

Running

Blocked

Counter: -1
Queue: T2

sema

T1

T2

T1

T2

UC Santa Barbara
Threads - Revisited

1: int i;
2: Semaphore sema;
3: 
4: f()
5: {
6:     printf(“i is %d\n”, i);
7: }
8: 
9: int main(int argc, char **argv)
10: {
11:     .. (do stuff here) ..
12:     P(sema);
13:     i = get_input();
14:     f();
15:     V(sema);
16:     return 0;
17: }

Address Space

Registers

Stack

Running

Blocked

Context

sema

Counter: -1

Queue: T2

PC = 7

PC = 13

T1

T1

T2

T2

i is 42

i is 42

i = 42

15

Running

Blocked

UC Santa Barbara
Threads - Revisited

1: int i;
2: Semaphore sema;
3:
4: f()
5: {
6:   printf("i is %d\n", i);
7: }
8:
9: int main(int argc, char **argv)
10: {
11:   .. (do stuff here) ..
12:   P(sema);
13:   i = get_input();
14:   f();
15:   V(sema);
16:   return 0;
17: }

Address Space

Registers

Stack

Running

Blocked

Context
Threads - Revisited

```c
1: int i;
2: Semaphore sema;
3:
4: f()
5: {
6:     printf("i is %d\n", i);
7: } 
8:
9: int main(int argc, char **argv)
10: {
11:     .. (do stuff here) ..
12:     P(sema);
13:     i = get_input();
14:     f();
15:     V(sema);
16:     return 0;
17: }
```
Threads - Revisited

```c
1: int i;
2: Semaphore sema;
3: 
4: f()
5: {
6:     printf("i is %d\n", i);
7: }
8: 
9: int main(int argc, char **argv)
10: {
11:     .. (do stuff here) ..
12:     P(sema);
13:     i = get_input();
14:     f();
15:     V(sema);
16:     return 0;
17: }
```

Address Space

i = 42

Registers

PC = 16

PC = 13

Stack

T1

T2

Running

T1

T2

Context

sema

Counter: 0

Queue:
1: int i;
2: Semaphore sema;
3: 
4: f()
5: {
6:   printf("i is %d\n", i);
7: }
8: 
9: int main(int argc, char **argv)
10: {
11:   .. (do stuff here) ..
12:   P(sema);
13:   i = get_input();
14:   f();
15:   V(sema);
16:   return 0;
17: }

Threads - Revisited
1: int i;
2: Semaphore sema;
3:
4: f()
5: {
6:     printf("i is %d\n", i);
7: }
8:
9: int main(int argc, char **argv)
10: {
11:     .. (do stuff here) ..
12:     P(sema);
13:     i = get_input();
14:     f();
15:     V(sema);
16:     return 0;
17: }
Threads - Revisited

1: int i;
2: Semaphore sema;
3:
4: f()
5: {
6:     printf("i is %d\n", i);
7: }
8:
9: int main(int argc, char **argv)
10: {
11:     .. (do stuff here) ..
12:     P(sema);
13:     i = get_input();
14:     f();
15:     V(sema);
16:     return 0;
17: }

Address Space

i = 17

Registers

PC = 16
T1

PC = 7
T2

Stack

15

Running

Sema

Counter: 0

Queue:

Context
## Threads - Revisited

```c
1: int i;
2: Semaphore sema;
3: 
4: f()
5: {
6:     printf("i is %d\n", i);
7: }
8: 
9: int main(int argc, char **argv)
10: {
11:     .. (do stuff here) ..
12:     P(sema);
13:     i = get_input();
14:     f();
15:     V(sema);
16:     return 0;
17: }
```

### Address Space
- `i = 17`

### Registers
- PC = 16
- PC = 15
- T1
- T2

### Stack
- T1
- T2

### Running Context
- `sema`
- Counter: 0
- Queue:
Threads - Revisited

1: int i;
2: Semaphore sema;
3:
4: f()
5: {
6:   printf("i is %d\n", i);
7: }
8:
9: int main(int argc, char **argv)
10: {
11:   .. (do stuff here)..
12:   P(sema);
13:   i = get_input();
14:   f();
15:   V(sema);
16:   return 0;
17: }
Producer/Consumer with Semaphores

Three semaphores

1. full: counts the number of slots that are full
2. empty: keeps track of the empty slots
3. mutex: makes sure produce and consumer do not access the buffers at the same time

• Initially:
  – full = 0
  – empty = N
  – mutex = 1
Producer/Consumer with Semaphores

item = produce_item();

empty.down();
mutex.down();

insert_item(item)

mutex.up();
full.up();

full.down();
mutex.down();

item=remove_item()

mutex.up();
empty.up();

consume_item(item);
Producer/Consumer with a Mistake...

```plaintext
item = produce_item();
mutex.down();
empty.down();
insert_item(item)
mutex.up();
full.up();
full.down();
mutex.down();
item=remove_item()
mutex.up();
empty.up();
consume_item(item);
```
Monitors

- A monitor is a collection of procedures, variables, and data structures grouped together in a special module.
- Only one thread can be active in a monitor at any instant.
- Mutual exclusion is enforced by the compiler and therefore it is less prone to errors.
- Monitors introduce the concept of condition variables.

```plaintext
monitor example
    integer i;
    condition c;

procedure producer();
.
.
.
end;

procedure consumer();
.
.
.
end;
end monitor;
```
Condition Variables

- Condition variables support two operations
  - Wait
  - Signal
- `wait(condition)`: the calling thread blocks and allows another thread to enter the monitor
- `signal(condition)`: the calling thread wakes up a thread blocked on the condition variable
  - If more than one thread is waiting, only one is selected by the scheduler
  - The signal operation must be the last statement executed, so that the caller immediately exits the monitor
- Condition variables do not keep track of signals as semaphores do
Producer/Consumer with Monitors

```
monitor ProducerConsumer
  condition full, empty;
  integer count;
  procedure insert(item: integer);
  begin
    if count = N then wait(full);
    insert_item(item);
    count := count + 1;
    if count = 1 then signal(empty)
  end;
function remove: integer;
  begin
    if count = 0 then wait(empty);
    remove = remove_item;
    count := count - 1;
    if count = N - 1 then signal(full)
  end;
  count := 0;
end monitor;

procedure producer;
begin
  while true do
    begin
      item = produce_item;
      ProducerConsumer.insert(item)
    end
end;

procedure consumer;
begin
  while true do
    begin
      item = ProducerConsumer.remove;
      consume_item(item)
    end
end;
```
A Little Problem

monitor M
  condition cond1, cond2;
  function sub1();
  begin
    ...
    wait(cond1);
  end;
  function sub2();
  begin
    ...
    signal(cond1);
    ...
    wait(cond2);
  end;
  function sub3();
  begin
    ...
    signal(cond2);
    signal(cond2);
  end;
end;

• Process A is waiting on cond1
• Process B is waiting on cond2
• At time t0 process C calls M.sub2()
• At time t1 > t0 process D calls M.sub2()
• At time t2 > t1 process E calls M.sub3()
• Assume that all waiting queues are FIFO
• Assuming that Q has been waiting for condition "x" and P performs "signal(x)", consider two possible policies:
  – P waits until Q either leaves the monitor, or waits for another condition; or
  – Q waits until P either leaves the monitor, or waits for another condition
• Determine the order of execution of the processes
Solution

Policy 1
• C executes signal(cond1) and wakes up A
• C suspends and A starts executing sub1()
• A exits the monitor
• C restarts
• C waits on cond2 (after B)
• D enters the monitor with sub2()
• D executes signal(cond1) and nothing happens
• D waits on cond2 after (B and C)
• E enters the monitor with sub3()
• E executes the first signal on cond2 and wakes B
• E suspends and B starts
• B exits the monitor and E restarts
• E executes the second signal and wakes C
• E suspends and C starts
• C exits the monitor and E restarts
• E exits the monitor

Policy 2
• C executes signal(cond1) and wakes up A
• C continues until it waits on cond2 (after B)
• C suspends and A starts executing sub1()
• A exits the monitor
• D enters the monitor with sub2()
• D executes signal(cond1) and nothing happens
• D waits on cond2 after (B and C)
• E enters the monitor with sub3()
• E executes the first signal on cond2 and wakes B
• E executes the second signal on cond2 and wakes C
• E exits the monitor
• B starts
• B exits the monitor and C starts
• C exits the monitor
The Readers and Writers Problem

- Multiple threads can read from a database at the same time
- If one thread is writing data into the db, no process should be reading or writing at the same time
- First reader gets a hold of a lock on the db
- Subsequent readers just increment the reader counter (critical section with a mutex)
- When they are finished they decrement the counter (critical section with a mutex)
- Last reader does an up() on the database lock letting the writer access the db
- Writer may starve if readers are too “active”
Reader/Writer Solution

reader() {
    mutex.down();
    readerCount++;
    if (readerCount==1) db.down();
    mutex.up();

    read_db();

    mutex.down();
    readerCount--;
    if (readerCount==0) db.up();
    mutex.up();
    use_db_data();
}

writer() {
    prepare_db_data();
    db.down();
    write_db_data();
    db.up();
}
Dining Philosophers Problem
First Solution

```c
philosopher(int i) {
    while (1) {
        think();
        take_chopstick(i);
        take_chopstick((i + 1) % N);
        eat();
        put_chopstick(i);
        put_chopstick((i + 1) % N);
    }
}
```

• If all the philosopher take their left chopsticks they get stuck
Second Solution

philosopher(int i) {
    while (1) {
        think();
        take_chopstick(i);
        if (!available((i + 1) % N)) {
            put_chopstick(i);
            continue();
        }
        take_chopstick((i + 1) % N);
        eat();
        put_chopstick(i);
        put_chopstick((i + 1) % N);
    }
}

• It is possible that all the philosophers put down and pick up their chopsticks at the same time, leading to starvation

• think() should be randomized
Third Solution

- Use one mutex
  - Do a down() when acquiring chopsticks
  - Do an up() when releasing chopsticks

- Problem: only one philosopher can eat at once
Fourth Solution

• Maintain state of philosophers
  – Switch to HUNGRY when ready to eat
  – Sleep if no chopsticks available
  – When finished wake up your neighbors

• Use one semaphore for each philosopher, to be used to suspend in case no chopsticks are available

• Use one mutex for critical regions

• Use take_chopsticks/put_chopsticks to acquire both chopsticks
Fourth Solution

philosopher(i) {
    think();
    take_chopsticks(i);
    eat();
    put_chopsticks(i);
}

take_chopsticks(i) {
    mutex.down();
    state[i] = HUNGRY;
    test(i);
    mutex.up();
    philosopher[i].down();
}

put_chopsticks(i) {
    mutex.down();
    state[i] = THINKING;
    test((i + 1) % N);
    test((i + N - 1) % N);
    mutex.up();
}

test(i) {
    if (state[i] == HUNGRY && state[(i + 1) % N] != EATING &&
        state[(i + N - 1) % N] != EATING)
    {
        state[i] = EATING;
        philosopher[i].up();
    }
}

The Sleeping Barber Problem

- Hair Salon with finite capacity (N chairs in the waiting room).
- Barber’s life:
  - Get the next customer
  - Give him/her haircut
- Customer’s life:
  - Grow hair
  - Enter the Hair Salon if possible (chairs are available)
  - Get haircut
  - Leave the Hair Salon
The Sleeping Barber Problem

- Three semaphores
  - Customers: counts the waiting customers, initially = 0
  - Barber: available barbers (0 or 1), initially = 0
  - Mutex: critical section control, initially = 1

- Variables
  - waiting: keeps track of how many customers, initially = 0
    - Needed because the value of a semaphore cannot be read
The Sleeping Barber Problem

barber() {
    while (1) {
        customers.down();
        mutex.down();
        waiting--;
        barber.up();
        mutex.up();
        cut_hair();
    }
}

customer() {
    mutex.down();
    if (waiting < CHAIRS) {
        waiting++;
        customers.up();
        mutex.up();
        barber.down();
        get_haircut();
    } else {
        mutex.up();
    }
}