The Well-Tempered Semaphore: Theme with Variations

Kenneth A. Reek, Professor
Department of Computer Science
Rochester Institute of Technology
Rochester, NY 14623

kar@cs.rit.edu

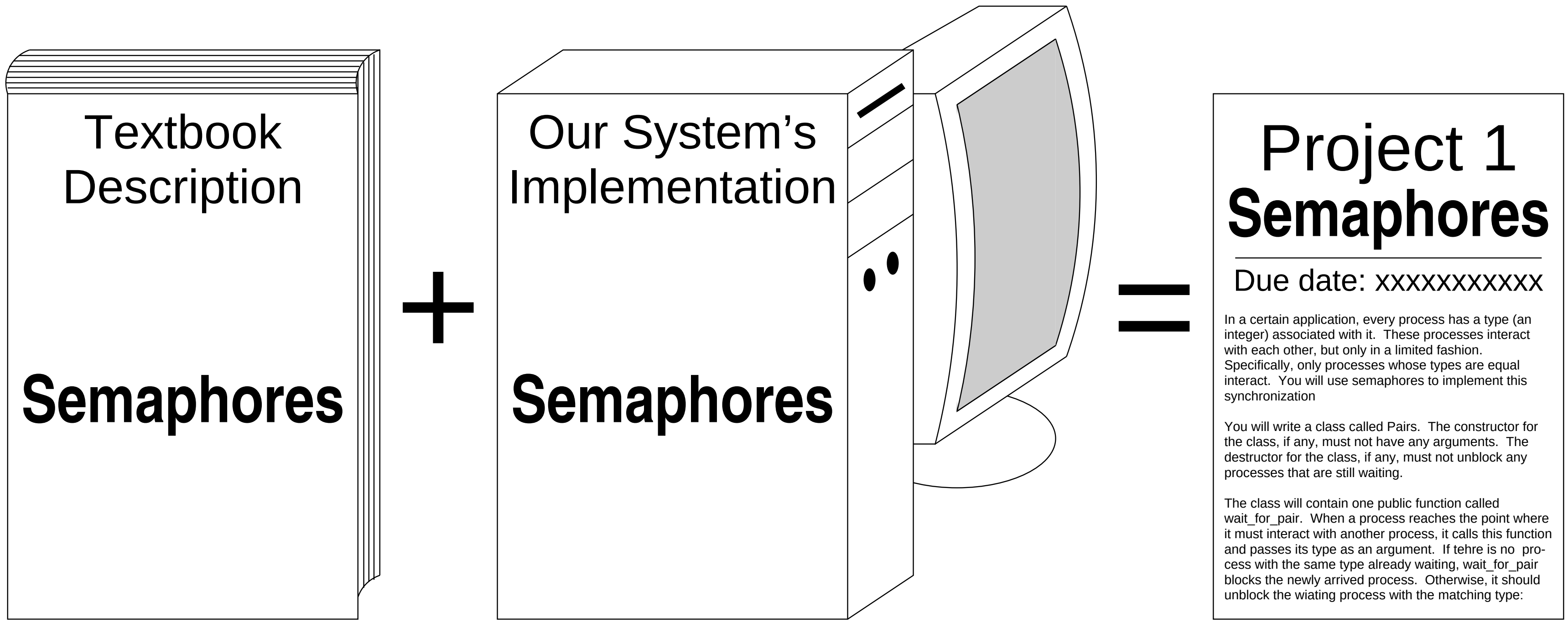
Introduction

Tell me and I know.
Show me and I remember.
Let me do it and I understand.

Confucius

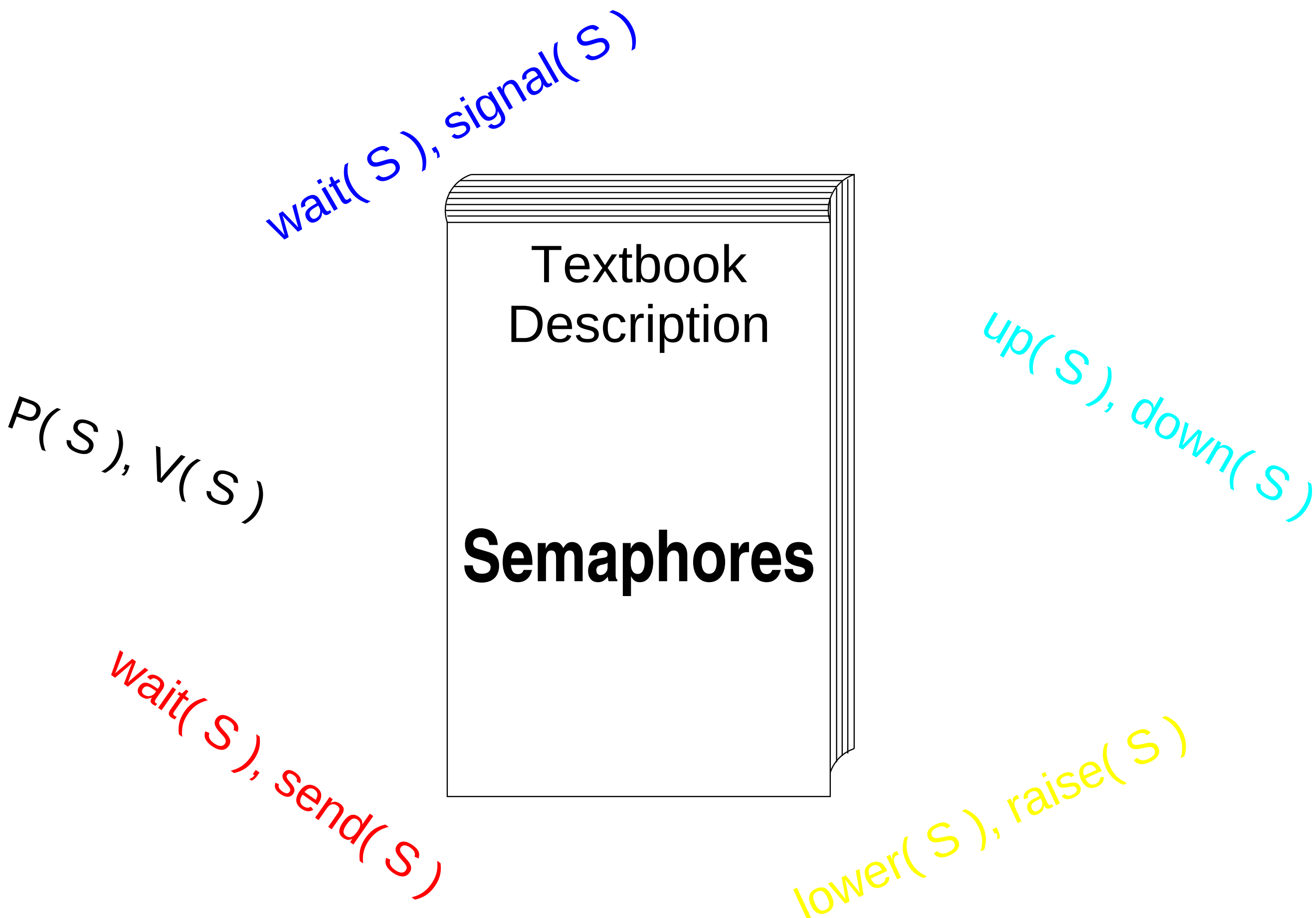
- Synchronization is difficult!
- Assigning real synchronization problems should improve student comprehension.

It Looks Easy Enough ...



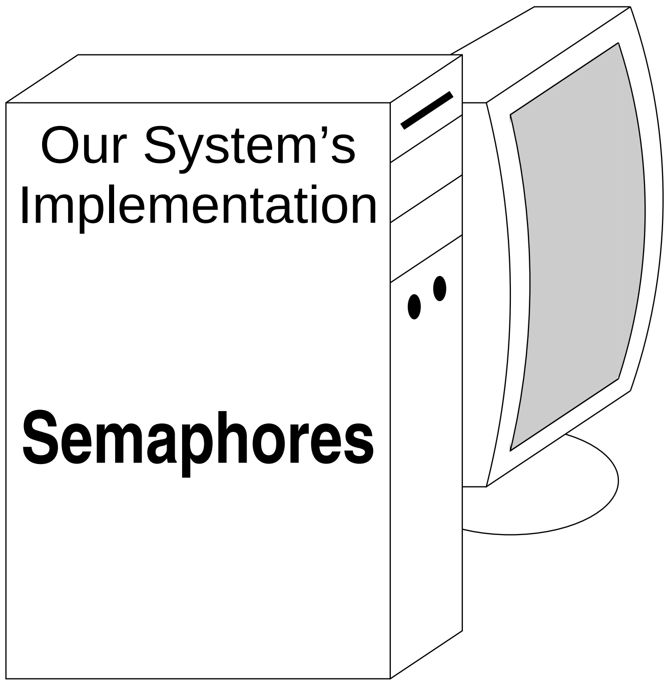
R·I·T

The Reality—Textbook Interfaces



Rochester Institute of Technology
Department of Computer Science

The Reality—Posix Interface



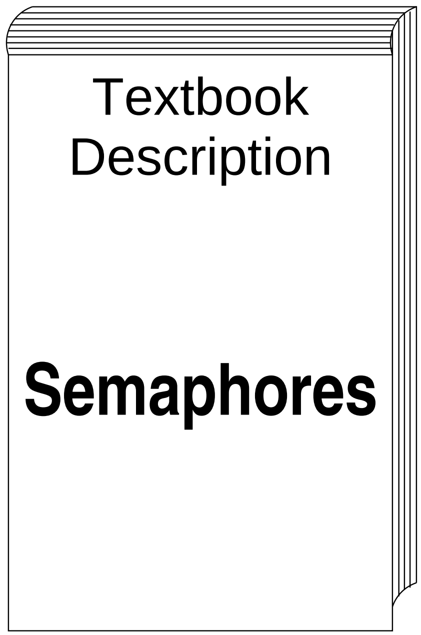

```
sem_t  S;

int sem_init( sem_t *S, int pshared,
              unsigned int value );

int sem_wait( sem_t *S );
```

(But if sem_wait returns -1
and errno == EINTR,
you MUST call it again!)

```
int sem_post( sem_t *S );

int sem_destroy( sem_t *S );
```

- SYS V semaphores are worse

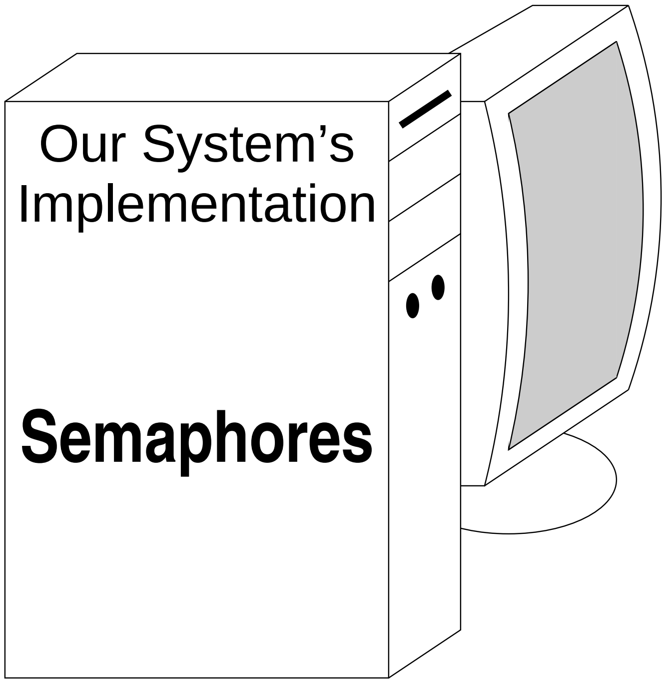
The Reality—Dijkstra's Semantics




```
P( s ):
        s = s - 1
        if s < 0 then
                wait on s
        endif

V( s ):
        s = s + 1
        if s <= 0 then
                unblock one process waiting on s
        endif
```

An unblocked process has priority over newly arriving processes

The Reality—Posix Semantics




```
sem_wait( s ):
    lock mutex
    while count <= 0 do
        unlock mutex
        sleep
        lock mutex
    endwhile
    count = count - 1
    unlock mutex

sem_post( s ):
    lock mutex
    count = count + 1
    unlock mutex
    wake up one sleeping pocess
```

An unblocked process competes with newly arriving processes

R·I·T

The Semantic Difference

Process A:	Process B:
while true do	while true do
noncritical section	noncritical section
P(s)	P(s)
critical section	critical section
V(s)	V(s)
endwhile	endwhile

Rochester Institute of Technology
Department of Computer Science

Posix Semantics

Process A:

```
while true do
    noncritical section
    P( s )
    critical section
    V( s )
endwhile
```

A is here

Process B:

```
while true do
    noncritical section
    P( s )
    critical section
    V( s )
endwhile
```

B is blocked in the P

Rochester Institute of Technology
Department of Computer Science

Posix Semantics

Process A:

```
while true do
    noncritical section
    P( s )
    critical section
    V( s )
endwhile
```

Process B:

```
while true do
    noncritical section
    P( s )
    critical section
    V( s )
endwhile
```

A leaves the CS

B is awakened but not yet running

- If A continues to run, it can sneak back into the CS before B ever resumes executing

Rochester Institute of Technology
Department of Computer Science

R·I·T

Dijkstra's Semantics

Process A:	Process B:
while true do	while true do
noncritical section	noncritical section
P(s)	P(s)
critical section	critical section
V(s)	V(s)
endwhile	endwhile

A is blocked next time around ***B will continue***

- A cannot sneak back into the CS before B
- In fact, B must be viewed as entering the CS as soon as the V is done.

Rochester Institute of Technology
Department of Computer Science

The Semantic Difference

- With Posix semantics, one process could indefinitely postpone the other.
- Dijkstra's semantics guarantee that this does not happen.

Therefore:

- Posix semantics are ***weak*** [Stark]
- Dijkstra's semantics are ***strong***

However:

- With three or more processes, any two of them can indefinitely postpone the others
- Neither Dijkstra nor Posix specify the order in which sleeping processes are unblocked (though some implementations do).

Monitors

A higher-level synchronization technique that ensures mutual exclusion.

- ***wait(condition)*** always blocks the process doing it
- ***signal(condition)*** unblocks one waiting process (ignored if there are none)

Mutual Exclusion in a Monitor

To maintain mutual exclusion, signal must temporarily delay one process until the other leaves or waits.

- Hoare's semantics: the process doing the signal is delayed, the one that received the signal runs first.
- Brinch Hansen's semantics: the process that received the signal is delayed, the one doing the signal continues.

The Semantic Difference

With Hoare's semantics, execution is constantly switching from the signalling process to the signalled one.

- This code looks bad but may be perfectly good:

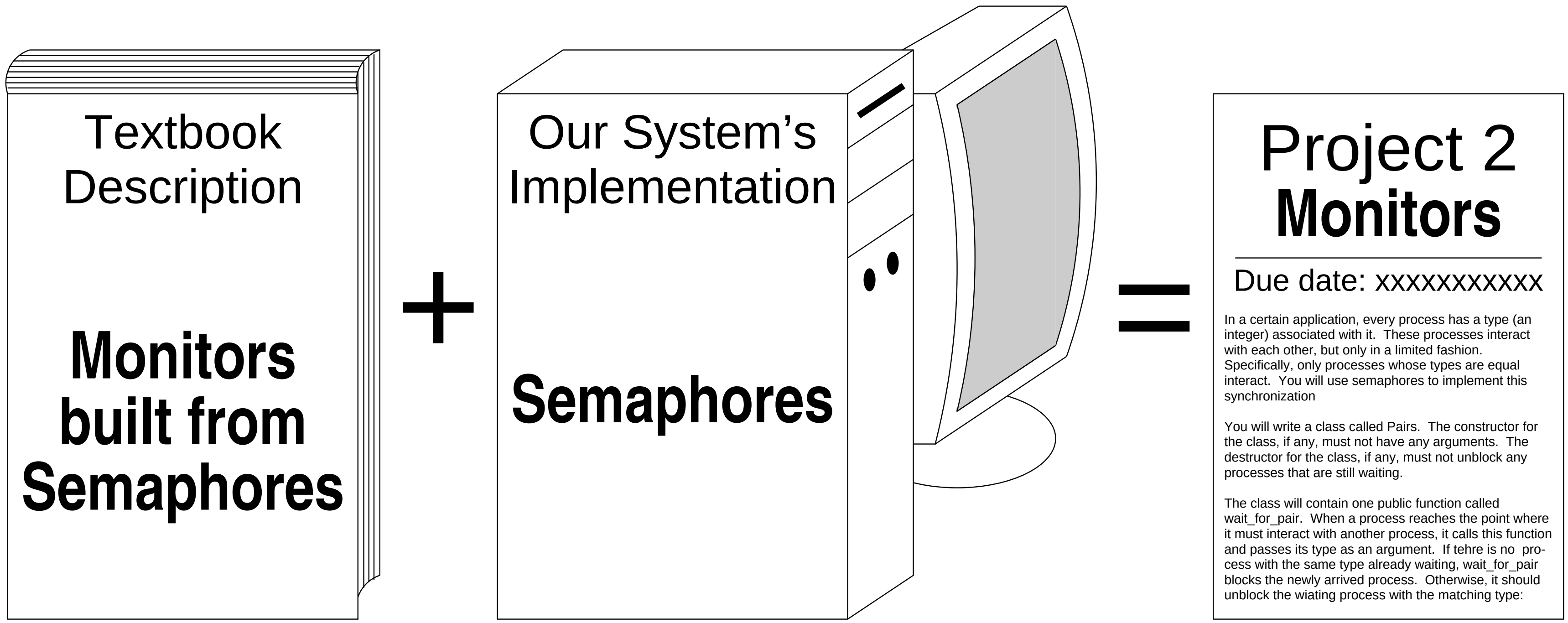
```
while count > 0 do
    signal( condition )
endwhile
```

Brinch Hansen's semantics are more ``natural´´

- The running process keeps running until it blocks itself or leaves

R·I·T

This Looks Pretty Easy, Too ...



Monitors and Semaphores

Monitors are built from semaphores. Will using strong vs. weak semaphores make any difference?

- Let's see ...

Rochester Institute of Technology
Department of Computer Science

A Monitor to Enforce Alternation

```
condition other_guy

alternate():
    signal( other_guy )
    wait( other_guy )
```

R·I·T

Brinch Hansen's Wait and Signal

```
wait( x ):
    x.count = x.count + 1
    if urgentcount > 0 then
        urgentcount = urgentcount - 1
        V( urgent )
    else
        V( gate )
    endif
    P( x.semaphore )
    P( urgent )

signal( x ):
    if x.count > 0 then
        x.count = x.count - 1
        urgentcount = urgentcount + 1
        V( x.semaphore )
    endif
```

Process A calls alternate

```
condition other_guy

alternate():
      signal( other_guy )
      wait( other_guy )
```

other_guy.count: 0

other_guy.semaphore: 0

urgentcount: 0

urgent: 0

```
wait( x ):
      x.count = x.count + 1
      if urgentcount > 0 then
            urgentcount = urgentcount - 1
            V( urgent )
      else
            V( gate )
      endif
      P( x.semaphore )
      P( urgent )

signal( x ):
      if x.count > 0 then
            x.count = x.count - 1
            urgentcount = urgentcount + 1
            V( x.semaphore )
      endif
```

Process B calls alternate: signal

```
condition other_guy

alternate():
    signal( other_guy )
    wait( other_guy )
```

other_guy.count: ~~0~~ 1

other_guy.semaphore: 0

urgentcount: 0

urgent: 0

```
wait( x ):
    x.count = x.count + 1
    if urgentcount > 0 then
        urgentcount = urgentcount - 1
        V( urgent )
    else
        V( gate )
    endif
    P( x.semaphore )  <-- A blocked
    P( urgent )

signal( x ):
    if x.count > 0 then
        x.count = x.count - 1
        urgentcount = urgentcount + 1
        V( x.semaphore )
    endif
```

Process B calls alternate: wait (1st part)

```
condition other_guy

alternate():
    signal( other_guy )
    wait( other_guy )
```

other_guy.count: ~~0~~ ~~1~~ 0

other_guy.semaphore: ~~0~~ 1

urgentcount: ~~0~~ 1

urgent: 0

```
wait( x ):
    x.count = x.count + 1
    if urgentcount > 0 then
        urgentcount = urgentcount - 1
        V( urgent )
    else
        V( gate )
    endif
    P( x.semaphore ) <-- A awakened
    P( urgent )

signal( x ):
    if x.count > 0 then
        x.count = x.count - 1
        urgentcount = urgentcount + 1
        V( x.semaphore )
    endif
```

Rochester Institute of Technology
Department of Computer Science

Process B calls alternate: wait (2nd part)

```
condition other_guy

alternate():
      signal( other_guy )
      wait( other_guy )
```

other_guy.count: ~~0~~ ~~1~~ ~~0~~ 1

other_guy.semaphore: ~~0~~ 1

urgentcount: ~~0~~ ~~1~~ 0

urgent: ~~0~~ 1

```
wait( x ):
      x.count = x.count + 1
      if urgentcount > 0 then
            urgentcount = urgentcount - 1
            V( urgent )
      else
            V( gate )
      endif                          B is here
      P( x.semaphore )               A awakened
      P( urgent )

signal( x ):
      if x.count > 0 then
            x.count = x.count - 1
            urgentcount = urgentcount + 1
            V( x.semaphore )
      endif
```

R·I·T

Process B calls alternate: Grand Finale

condition other_guy

alternate():
    signal(other_guy)
    wait(other_guy)

other_guy.count: ~~*0*~~ ~~*1*~~ ~~*0*~~ *1*

other_guy.semaphore: ~~*0*~~ ~~*1*~~ *0*

urgentcount: ~~*0*~~ ~~*1*~~ *0*

urgent: ~~*0*~~ ~~*1*~~ *0*

```
wait( x ):
    x.count = x.count + 1
    if urgentcount > 0 then
        urgentcount = urgentcount - 1
        V( urgent )
    else
        V( gate )
    endif
    P( x.semaphore )   <-- A reblocked!
    P( urgent )
                   <-- B departs!

signal( x ):
    if x.count > 0 then
        x.count = x.count - 1
        urgentcount = urgentcount + 1
        V( x.semaphore )
    endif
```

Rochester Institute of Technology
Department of Computer Science

Analysis

- A was blocked when B called alternate
- B *should have* blocked, A *should have* continued
- The opposite happened ...
- Conclusion: *IT DIDN'T WORK!* (duh)

Monitors and Semaphores

Does using strong vs. weak semaphores make any difference?

- Brinch Hansen’s semantics: more intuitive, but weak semaphores can cause a failure
- Hoare’s semantics: less intuitive, but works correctly with either type of semaphore
- Weak semaphores are not as useful as strong semaphores

Weak semaphores can cause problems when implementing synchronization projects!

Solution: Hide Them with a Wrapper

Rochester Institute of Technology
Department of Computer Science

Wrapper: Dijkstra's Counting Semaphores

Object Oriented implementation in C++ enforces semaphore rules:

- The only operations defined are P and V
- Data is private
- Constructor *requires* an initializer

Option to reverse the unblocking order

- Helps find programs that incorrectly depend on the order

Implemented using Posix (or Solaris) condition variables

- Should port easily to other Posix systems
- Might port to Win32 (perhaps using Events)?

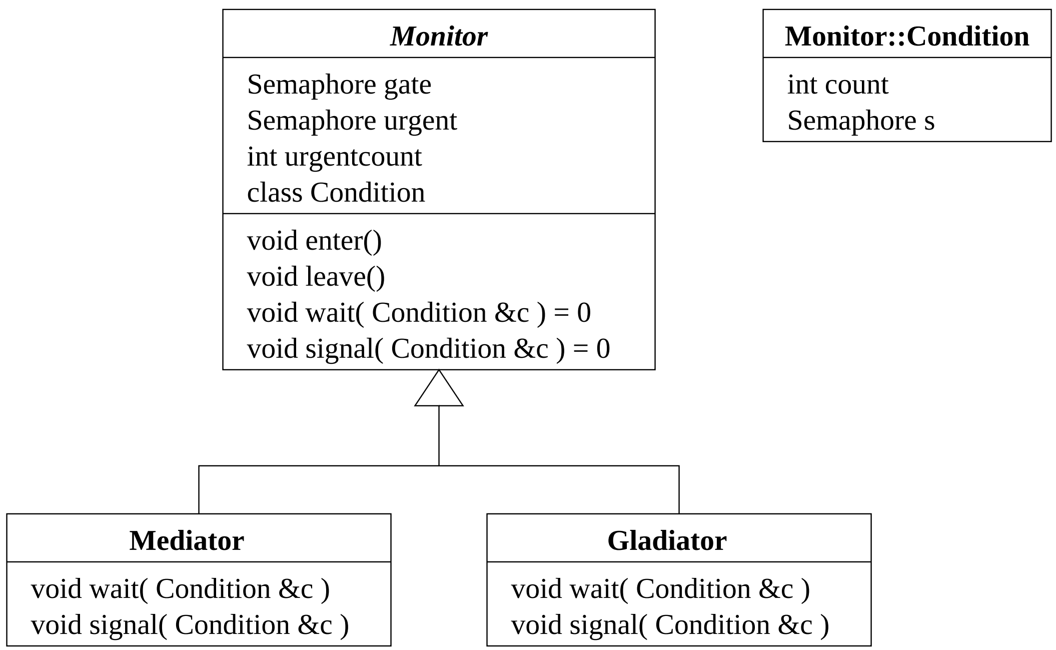
Using the Wrapper: Semaphores

```
#include "Semaphore.h"

Semaphore mutex( 1 ); // Initialize count to 1

while( true ){
        // non-critical section
        mutex.P();
        // critical section
        mutex.V();
}
```

- Wrapper defines simpler interface to create threads

Wrapper: Monitors

- Both Hoare's semantics (called Mediator) and Brinch Hansen's semantics (called Gladiator) are implemented



Rochester Institute of Technology
Department of Computer Science

R·I·T

Using the Wrapper: Monitors

```
#include "Monitor.h"

class TwoThreadAlternate: public Gladiator {
public:
        void    alternate();

private:
        Condition other_guy;
};

void TwoThreadAlternate::alternate(){
        enter();        // enter mutual exclusion
        signal( other_guy );
        wait( other_guy );
        leave();        // leave mutual exclusion
}
```

R·I·T

Nifty Assignments: Semaphores

Lucky 7's

- Every thread has a number from 1-6
- Incoming threads are blocked until there is a group whose numbers add up to 7; those in the group are unblocked

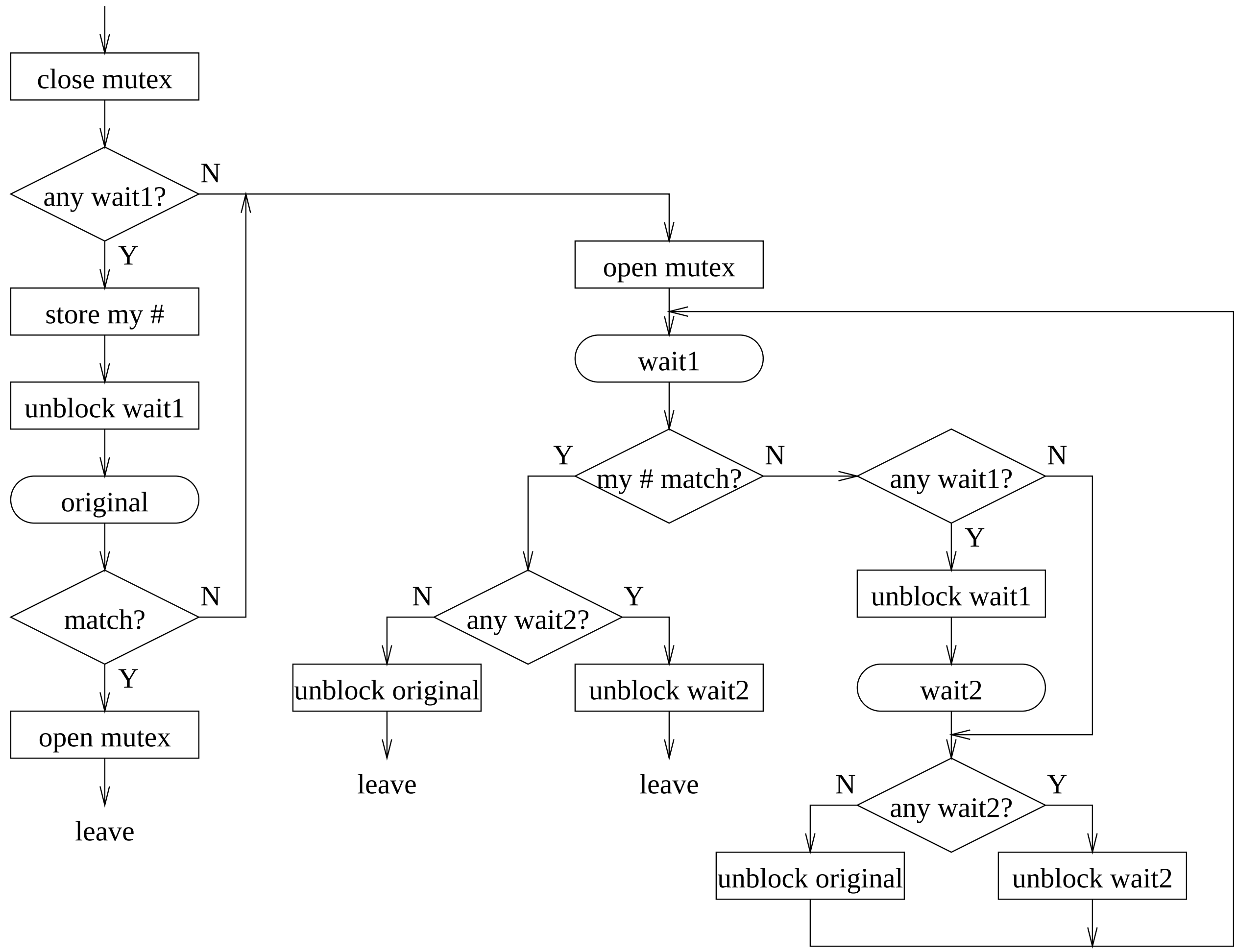
Thread groups

- Incoming threads are blocked until there are enough to form a group, which is then unblocked
- Group size can be changed, which may unblock groups too

Pairs

- Every thread has an arbitrary integer id
- Each thread is blocked until another with the same id arrives

Pairs Logic using Semaphores



Rochester Institute of Technology
Department of Computer Science

Nifty Assignments: Monitors

Messages

- Pass a single integer message among threads
- Blocking & non-blocking send, blocking & non-blocking receive

Lucky 7's

Pairs

Rochester Institute of Technology
Department of Computer Science

Pairs Logic using Brinch Hansen's Monitor

- New arrival saves his ID and unblocks all waiting threads
- Waiting threads compare their ID one by one
- If they don't match, they reblock
- If one matches, it continues
- Last unblocked thread signals the new arrival

This Stuff is Great! Can We Use It?

Sure!

But please share any NIFTY ASSIGNMENTS you dream up with the rest of us!

From the World Wide Web:

- Go to www.cs.rit.edu/~kar, click *``Various papers and colloquia´´*

You'll find:

- All of the source code and handouts describing it
- Handouts showing semaphore and monitor solutions to several classic synchronization problems
- The paper and these slides

R·I·T

Thank you for your attention!

Kenneth A. Reek, Professor
Department of Computer Science
Gollisano College of Computing and Information Sciences
Rochester Institute of Technology

www.cs.rit.edu/~kar
kar@cs.rit.edu