

# Processes and Threads

Week 05 - Lecture

Interprocess Communication (IPC)

# Team

- Instructor
  - Giancarlo Succi
- Teaching Assistants
  - Nikita Lozhnikov (also Tutorial Instructor)
  - Manuel Rodriguez
  - Shokhista Ergasheva

# Sources

- These slides have been adapted from the original slides of the adopted book:
  - Tanenbaum & Bos, Modern Operating Systems: 4th edition, 2013  
Prentice-Hall, Inc.and customised for the needs of this course.
- Additional input for the slides are detailed later

# Interprocess Communication (1)

- Processes frequently need to communicate with other processes
- For example, in a shell pipeline, the output of the first process must be passed to the second process, and so on down the line
- **InterProcess Communication (or IPC)** should be carried out in a well-structured way not using interrupts

# Interprocess Communication (2)

- There are three issues here:
  - how one process can pass information to another
  - making sure two or more processes do not get in each other's way accessing the same resource
  - proper sequencing when dependencies are present:  
if process A produces data and process B prints them, B has to wait for A before starting to print
- The latter two of these issues apply equally well to threads

# Race Conditions (1/2)

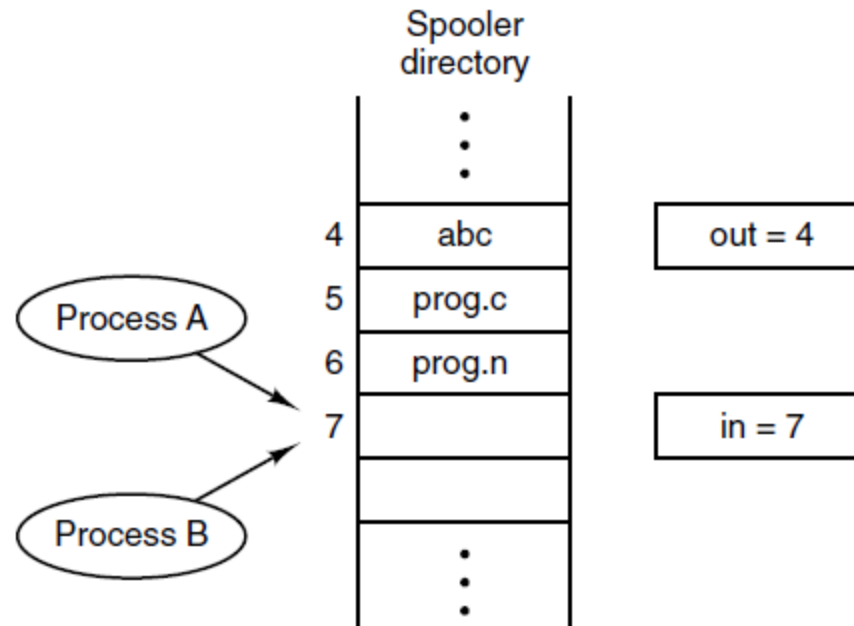


Figure 2-21. Two processes want to access shared memory at the same time.

# Race Conditions (2/2)

Process A  
file\_name = "a.txt"

	Spooler directory	
	.	
	.	
	.	
4	abc	out = 4
5	prog.c	
6	prog.n	
7		in = 7
8		
	.	
	.	
	.	

Process B  
file\_name = "b.txt"

# Race Conditions (2/2)

Process A  
file\_name = "a.txt"  
next\_free\_slot = 7

	Spooler directory	
	.	
	.	
	.	
4	abc	out = 4
5	prog.c	
6	prog.n	
7		in = 7
8		
	.	
	.	
	.	

Process B  
file\_name = "b.txt"



# Race Conditions (2/2)

Process A  
file\_name = "a.txt"  
next\_free\_slot = 7

Spooler directory		
	.	
	.	
	.	
4	abc	out = 4
5	prog.c	
6	prog.n	
7		in = 7
8		
	.	
	.	
	.	

Process B  
file\_name = "b.txt"

# Race Conditions (2/2)

Process A  
file\_name = "a.txt"  
next\_free\_slot = 7

Spooler directory		
	.	
	.	
	.	
4	abc	out = 4
5	prog.c	
6	prog.n	
7		in = 7
8		
	.	
	.	
	.	

Process B  
file\_name = "b.txt"  
next\_free\_slot = 7

# Race Conditions (2/2)

Process A  
file\_name = "a.txt"  
next\_free\_slot = 7

Spooler directory		
	.	
	.	
	.	
4	abc	out = 4
5	prog.c	
6	prog.n	
7	b.txt	
8		in = 8
	.	
	.	
	.	

Process B  
file\_name = "b.txt"  
next\_free\_slot = 7

# Race Conditions (2/2)

Process A  
file\_name = "a.txt"  
next\_free\_slot = 7

Spooler directory		
	.	
	.	
	.	
4	abc	out = 4
5	prog.c	
6	prog.n	
7	b.txt	
8		in = 8
	.	
	.	
	.	

Process B  
file\_name = "b.txt"  
next\_free\_slot = 7

# Race Conditions (2/2)

Process A  
file\_name = "a.txt"  
next\_free\_slot = 7

Spooler directory	
	.
	.
	.
4	abc
5	prog.c
6	prog.n
7	a.txt
8	
	.
	.
	.

out = 4

in = 8

in = next\_free\_slot + 1  
The spooler directory  
is now internally consistent

Process B  
file\_name = "b.txt"  
next\_free\_slot = 7

# Avoiding Race Conditions

- Mutual Exclusion with Busy Waiting
- Sleep and Wakeup
- Semaphores
- Mutexes
- Monitors
- Message Passing
- Barriers
- Avoiding Locks: Read-Copy-Update

# Critical Regions (1)

- A way to prohibit more than one process from reading and writing the shared data at the same time is called **mutual exclusion**
- The part of the program where the shared memory is accessed is called the **critical region** or **critical section** (Fig. 2-22)

# Critical Regions (2)

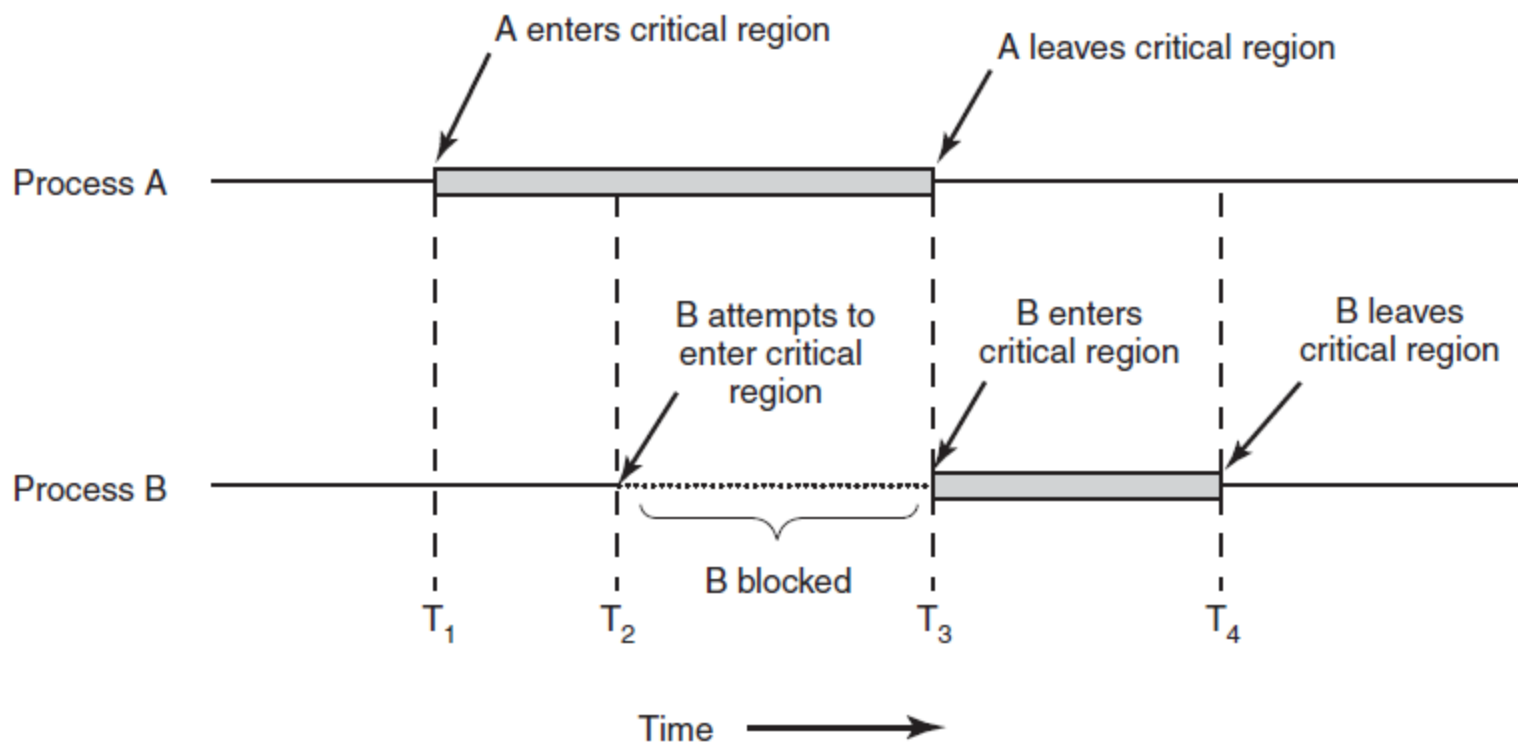


Figure 2-22. Mutual exclusion using critical regions.



# Requirements to Implement Critical Regions

- No two processes may be simultaneously inside their critical regions
- No assumptions may be made about speeds or the number of CPUs
- No process running outside its critical region may block other processes
- No process should have to wait forever to enter its critical region

# Proposals to Implement Critical Regions

- Disabling interrupts
- Lock variables
- Strict Alternation
- Peterson's Solution
- The TSL Instruction

# Mutual Exclusion with Busy Waiting: Disabling Interrupts

- Each process disables all interrupts after entering its critical region and re-enables them before leaving it
- Issues:
  - If a process will not re-enable interrupts, this will stop the system
  - Will not work in the multiprocessor systems. Interrupts for only one CPU will be stopped

# Mutual Exclusion with Busy Waiting: Lock Variables

- A possible software solution:
  - To have a shared variable (*lock*) that is initially 0
  - A process entering its critical region tests the lock first
  - Lock == 0? The process sets it to 1 and enters the critical region
  - Lock == 1? the process waits until it becomes 0
- This idea contains exactly the same fatal flaw that was described earlier

# Mutual Exclusion with Busy Waiting: Strict Alternation (1)

- To have an integer variable (*turn*) that is initially 0
- Process 0 finds it to be 0, and enters its critical region
- Process 1 also finds it to be 0 and sits in a tight loop waiting for its turn
  - Continuously testing a variable until some value appears is called **busy waiting**
  - A lock that uses busy waiting is called a **spin lock**

# Mutual Exclusion with Busy Waiting: Strict Alternation (2)

- This solution violates condition 3:

*No process running outside its critical region may block other processes*

- If one process is significantly slower, it will enter a critical region, exit it then will pass the turn to the second process and will continue executing in its noncritical region
- The second process will quickly finish its job, will pass the turn to the first process which might still not be done
- Thus, the second process will have to wait for a process in noncritical region (condition 3)

# Mutual Exclusion with Busy Waiting: Strict Alternation (3)

Process 0

while (TRUE) {
while (turn != 0) {
/* loop */
}
critical_region()
turn = 1
noncritical_region();
}

shared space

turn 0

Process 1

while (TRUE) {
while (turn != 1) {
/* loop */
}
critical_region()
turn = 0
noncritical_region();
}

# Mutual Exclusion with Busy Waiting: Strict Alternation (3)

Process 0

while (TRUE) {
while (turn != 0) {
/* loop */
}
critical_region()
turn = 1
noncritical_region();
}

shared space

turn 0

Process 1

while (TRUE) {
while (turn != 1) {
/* loop */
}
critical_region()
turn = 0
noncritical_region();
}



# Mutual Exclusion with Busy Waiting: Strict Alternation (3)

Process 0

while (TRUE) {
while (turn != 0) {
/* loop */
}
critical_region()
turn = 1
noncritical_region();
}

shared space

turn 0

Process 1

while (TRUE) {
while (turn != 1) {
/* loop */
}
critical_region()
turn = 0
noncritical_region();
}

# Mutual Exclusion with Busy Waiting: Strict Alternation (3)

Process 0

while (TRUE) {
while (turn != 0) {
/* loop */
}
critical_region()
turn = 1
noncritical_region();
}

shared space

turn 0

Process 1

while (TRUE) {
while (turn != 1) {
/* loop */
}
critical_region()
turn = 0
noncritical_region();
}

# Mutual Exclusion with Busy Waiting: Strict Alternation (3)

Process 0

while (TRUE) {
while (turn != 0) {
/* loop */
}
critical_region()
turn = 1
noncritical_region();
}

shared space

turn 0

Process 1

while (TRUE) {
while (turn != 1) {
/* loop */
}
critical_region()
turn = 0
noncritical_region();
}

# Mutual Exclusion with Busy Waiting: Strict Alternation (3)

Process 0

while (TRUE) {
while (turn != 0) {
/* loop */
}
critical_region()
turn = 1
noncritical_region();
}

shared space

turn 0

Process 1

while (TRUE) {
while (turn != 1) {
/* loop */
}
critical_region()
turn = 0
noncritical_region();
}

# Mutual Exclusion with Busy Waiting: Strict Alternation (3)

Process 0

while (TRUE) {
while (turn != 0) {
/* loop */
}
critical_region()
turn = 1
noncritical_region();
}

shared space

turn 0

Process 1

while (TRUE) {
while (turn != 1) {
/* loop */
}
critical_region()
turn = 0
noncritical_region();
}

# Mutual Exclusion with Busy Waiting: Strict Alternation (3)

Process 0

while (TRUE) {
while (turn != 0) {
/* loop */
}
critical_region()
turn = 1
noncritical_region();
}

shared space

turn 0

Process 1

while (TRUE) {
while (turn != 1) {
/* loop */
}
critical_region()
turn = 0
noncritical_region();
}

# Mutual Exclusion with Busy Waiting: Strict Alternation (3)

Process 0

while (TRUE) {
while (turn != 0) {
/* loop */
}
critical_region()
turn = 1
noncritical_region();
}

shared space

turn 0

Process 1

while (TRUE) {
while (turn != 1) {
/* loop */
}
critical_region()
turn = 0
noncritical_region();
}

# Mutual Exclusion with Busy Waiting: Strict Alternation (3)

Process 0

while (TRUE) {
while (turn != 0) {
/* loop */
}
critical_region()
turn = 1
noncritical_region();
}

shared space

turn 0

Process 1

while (TRUE) {
while (turn != 1) {
/* loop */
}
critical_region()
turn = 0
noncritical_region();
}



# Mutual Exclusion with Busy Waiting: Strict Alternation (3)

Process 0

while (TRUE) {
while (turn != 0) {
/* loop */
}
critical_region()
turn = 1
noncritical_region();
}

shared space

turn 0

Process 1

while (TRUE) {
while (turn != 1) {
/* loop */
}
critical_region()
turn = 0
noncritical_region();
}

# Mutual Exclusion with Busy Waiting: Strict Alternation (3)

Process 0

while (TRUE) {
while (turn != 0) {
/* loop */
}
critical_region()
turn = 1
noncritical_region();
}

shared space

turn 0

Process 1

while (TRUE) {
while (turn != 1) {
/* loop */
}
critical_region()
turn = 0
noncritical_region();
}

# Mutual Exclusion with Busy Waiting: Strict Alternation (3)

Process 0

while (TRUE) {
while (turn != 0) {
/* loop */
}
critical_region()
turn = 1
noncritical_region();
}

shared space

turn 0

Process 1

while (TRUE) {
while (turn != 1) {
/* loop */
}
critical_region()
turn = 0
noncritical_region();
}

# Mutual Exclusion with Busy Waiting: Strict Alternation (3)

Process 0

while (TRUE) {
while (turn != 0) {
/* loop */
}
critical_region()
turn = 1
noncritical_region();
}

shared space

turn 1

Process 1

while (TRUE) {
while (turn != 1) {
/* loop */
}
critical_region()
turn = 0
noncritical_region();
}

# Mutual Exclusion with Busy Waiting: Strict Alternation (3)

Process 0

while (TRUE) {
while (turn != 0) {
/* loop */
}
critical_region()
turn = 1
noncritical_region();
}

shared space

turn 1

Process 1

while (TRUE) {
while (turn != 1) {
/* loop */
}
critical_region()
turn = 0
noncritical_region();
}

# Mutual Exclusion with Busy Waiting: Strict Alternation (3)

Process 0

while (TRUE) {
while (turn != 0) {
/* loop */
}
critical_region()
turn = 1
noncritical_region();
}

shared space

turn 1

Process 1

while (TRUE) {
while (turn != 1) {
/* loop */
}
critical_region()
turn = 0
noncritical_region();
}

# Mutual Exclusion with Busy Waiting: Strict Alternation (3)

Process 0

while (TRUE) {
while (turn != 0) {
/* loop */
}
critical_region()
turn = 1
noncritical_region();
}

shared space

turn 1

Process 1

while (TRUE) {
while (turn != 1) {
/* loop */
}
critical_region()
turn = 0
noncritical_region();
}

# Mutual Exclusion with Busy Waiting: Strict Alternation (3)

Process 0

while (TRUE) {
while (turn != 0) {
/* loop */
}
critical_region()
turn = 1
noncritical_region();
}

shared space

turn 1

Process 1

while (TRUE) {
while (turn != 1) {
/* loop */
}
critical_region()
turn = 0
noncritical_region();
}



# Mutual Exclusion with Busy Waiting: Strict Alternation (3)

Process 0

while (TRUE) {
while (turn != 0) {
/* loop */
}
critical_region()
turn = 1
noncritical_region();
}

shared space

turn 1

Process 1

while (TRUE) {
while (turn != 1) {
/* loop */
}
critical_region()
turn = 0
noncritical_region();
}

# Mutual Exclusion with Busy Waiting: Strict Alternation (3)

Process 0

while (TRUE) {
while (turn != 0) {
/* loop */
}
critical_region()
turn = 1
noncritical_region();
}

shared space

turn 1

Process 1

while (TRUE) {
while (turn != 1) {
/* loop */
}
critical_region()
turn = 0
noncritical_region();
}

# Mutual Exclusion with Busy Waiting: Strict Alternation (3)

Process 0

while (TRUE) {
while (turn != 0) {
/* loop */
}
critical_region()
turn = 1
noncritical_region();
}

shared space

turn 1

Process 1

while (TRUE) {
while (turn != 1) {
/* loop */
}
critical_region()
turn = 0
noncritical_region();
}

# Mutual Exclusion with Busy Waiting: Strict Alternation (3)

Process 0

while (TRUE) {
while (turn != 0) {
/* loop */
}
critical_region()
turn = 1
noncritical_region();
}

shared space

turn 0

Process 1

while (TRUE) {
while (turn != 1) {
/* loop */
}
critical_region()
turn = 0
noncritical_region();
}

# Mutual Exclusion with Busy Waiting: Strict Alternation (3)

Process 0

while (TRUE) {
while (turn != 0) {
/* loop */
}
critical_region()
turn = 1
noncritical_region();
}

shared space

turn 0

Process 1

while (TRUE) {
while (turn != 1) {
/* loop */
}
critical_region()
turn = 0
noncritical_region();
}

# Mutual Exclusion with Busy Waiting: Peterson's Solution (1)

- Before entering its critical region, each process calls *enter\_region()* with its own process number, 0 or 1, as parameter
- This call will cause it to wait until it is safe to enter
- After it has finished with the shared variables, the process calls *leave\_region()* to indicate that it is done

# Mutual Exclusion with Busy Waiting: Peterson's Solution (2)

## Process 0

enter_region(int process) {
int other = 1 – process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

shared space

interested[process0]= FALSE
interested[process1]= FALSE
turn = 0

## Process 1

enter_region(int process) {
int other = 1 – process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

# Mutual Exclusion with Busy Waiting: Peterson's Solution (2)

## Process 0

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

## shared space

```
interested[process0]= FALSE  
interested[process1]= FALSE  
turn = 0
```

## Process 1

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}



# Mutual Exclusion with Busy Waiting: Peterson's Solution (2)

## Process 0

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

## shared space

```
interested[process0]= FALSE  
interested[process1]= FALSE  
turn = 0
```

## Process 1

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

# Mutual Exclusion with Busy Waiting: Peterson's Solution (2)

## Process 0

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

shared space

interested[process0]= FALSE
interested[process1]= FALSE
turn = 0

## Process 1

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

# Mutual Exclusion with Busy Waiting: Peterson's Solution (2)

## Process 0

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

shared space

interested[process0]= TRUE interested[process1]= FALSE turn = 0
---

## Process 1

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

# Mutual Exclusion with Busy Waiting: Peterson's Solution (2)

## Process 0

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

## shared space

```
interested[process0]= TRUE  
interested[process1]= FALSE  
turn = 0
```

## Process 1

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

# Mutual Exclusion with Busy Waiting: Peterson's Solution (2)

## Process 0

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

shared space

interested[process0]= TRUE interested[process1]= FALSE turn = 0
---

## Process 1

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

# Mutual Exclusion with Busy Waiting: Peterson's Solution (2)

Process 0

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

shared space

interested[process0]= TRUE interested[process1]= FALSE turn = 0
---

Process 1

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

# Mutual Exclusion with Busy Waiting: Peterson's Solution (2)

Process 0

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

shared space

```
interested[process0]= TRUE  
interested[process1]= FALSE  
turn = 0
```

Process 1

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

# Mutual Exclusion with Busy Waiting: Peterson's Solution (2)

Process 0

enter_region(int process) {
int other = 1 – process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

shared space

interested[process0]= TRUE interested[process1]= FALSE turn = 0
---

Process 1

enter_region(int process) {
int other = 1 – process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}



# Mutual Exclusion with Busy Waiting: Peterson's Solution (2)

Process 0

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

shared space

interested[process0]= TRUE interested[process1]= FALSE turn = 0
---

Process 1

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

# Mutual Exclusion with Busy Waiting: Peterson's Solution (2)

Process 0

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

shared space

interested[process0]= TRUE interested[process1]= TRUE turn = 0
--

Process 1

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

# Mutual Exclusion with Busy Waiting: Peterson's Solution (2)

Process 0

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

shared space

interested[process0]= TRUE interested[process1]= TRUE turn = 0
--

Process 1

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

# Mutual Exclusion with Busy Waiting: Peterson's Solution (2)

Process 0

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

shared space

```
interested[process0]= TRUE
interested[process1]= TRUE
turn = 1
```

Process 1

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

# Mutual Exclusion with Busy Waiting: Peterson's Solution (2)

Process 0

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

shared space

interested[process0]= TRUE interested[process1]= TRUE turn = 1
--

Process 1

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

# Mutual Exclusion with Busy Waiting: Peterson's Solution (2)

Process 0

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

shared space

```
interested[process0]= TRUE  
interested[process1]= TRUE  
turn = 1
```

Process 1

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

# Mutual Exclusion with Busy Waiting: Peterson's Solution (2)

Process 0

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

shared space

```
interested[process0]= TRUE  
interested[process1]= TRUE  
turn = 1
```

Process 1

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

# Mutual Exclusion with Busy Waiting: Peterson's Solution (2)

Process 0

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

shared space

```
interested[process0]= TRUE
interested[process1]= TRUE
turn = 1
```

Process 1

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}



# Mutual Exclusion with Busy Waiting: Peterson's Solution (2)

Process 0

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

shared space

```
interested[process0]= TRUE
interested[process1]= TRUE
turn = 1
```

Process 1

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

# Mutual Exclusion with Busy Waiting: Peterson's Solution (2)

Process 0

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

shared space

```
interested[process0]= TRUE  
interested[process1]= TRUE  
turn = 1
```

Process 1

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

# Mutual Exclusion with Busy Waiting: Peterson's Solution (2)

Process 0

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

shared space

```
interested[process0]= TRUE
interested[process1]= TRUE
turn = 1
```

Process 1

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

# Mutual Exclusion with Busy Waiting: Peterson's Solution (2)

Process 0

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

shared space

```
interested[process0]= TRUE  
interested[process1]= TRUE  
turn = 1
```

Process 1

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

# Mutual Exclusion with Busy Waiting: Peterson's Solution (2)

Process 0

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

shared space

```
interested[process0]= TRUE  
interested[process1]= TRUE  
turn = 1
```

Process 1

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

# Mutual Exclusion with Busy Waiting: Peterson's Solution (2)

Process 0

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

shared space

```
interested[process0]= TRUE
interested[process1]= TRUE
turn = 1
```

Process 1

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

# Mutual Exclusion with Busy Waiting: Peterson's Solution (2)

Process 0

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

shared space

interested[process0]= TRUE interested[process1]= TRUE turn = 1
--

Process 1

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

# Mutual Exclusion with Busy Waiting: Peterson's Solution (2)

Process 0

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

shared space

```
interested[process0]= FALSE  
interested[process1]= TRUE  
turn = 1
```

Process 1

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}



# Mutual Exclusion with Busy Waiting: Peterson's Solution (2)

Process 0

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

shared space

interested[process0]= FALSE interested[process1]= TRUE turn = 1
---

Process 1

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

# Mutual Exclusion with Busy Waiting: Peterson's Solution (2)

Process 0

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

shared space

interested[process0]= FALSE interested[process1]= TRUE turn = 1
---

Process 1

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

# Mutual Exclusion with Busy Waiting: Peterson's Solution (2)

Process 0

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

shared space

```
interested[process0]= FALSE  
interested[process1]= TRUE  
turn = 1
```

Process 1

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

# Mutual Exclusion with Busy Waiting: Peterson's Solution (2)

Process 0

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

shared space

```
interested[process0]= FALSE  
interested[process1]= TRUE  
turn = 1
```

Process 1

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

# Mutual Exclusion with Busy Waiting: Peterson's Solution (2)

Process 0

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

shared space

interested[process0]= FALSE interested[process1]= TRUE turn = 1
---

Process 1

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

# Mutual Exclusion with Busy Waiting: Peterson's Solution (2)

Process 0

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

shared space

interested[process0]= FALSE interested[process1]= TRUE turn = 1
---

Process 1

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

# Mutual Exclusion with Busy Waiting: Peterson's Solution (2)

Process 0

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

shared space

interested[process0]= FALSE interested[process1]= TRUE turn = 1
---

Process 1

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

# Mutual Exclusion with Busy Waiting: Peterson's Solution (2)

Process 0

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

shared space

```
interested[process0]= FALSE
interested[process1]= FALSE
turn = 1
```

Process 1

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}



# Mutual Exclusion with Busy Waiting: Peterson's Solution (2)

Process 0

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

shared space

```
interested[process0]= FALSE  
interested[process1]= FALSE  
turn = 1
```

Process 1

enter_region(int process) {
int other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process && interested[other] == TRUE) {
/* loop */
}
critical_region();
leave_region(int process) {
interested[process] = FALSE;
}

# Mutual Exclusion with Busy Waiting: The TSL Instruction (1)

- Hardware-assisted approach; works with multiple CPUs
- An instruction like ***TSL RX,LOCK*** (Test and Set Lock) reads the contents of the memory word *lock* into register *RX* and then sets *lock* to be a nonzero value
- The operations of reading the word and storing into it are guaranteed to be indivisible (**the memory bus is locked**)
- Different from disabling interrupts - doing it on processor 1 has no effect at all on processor 2

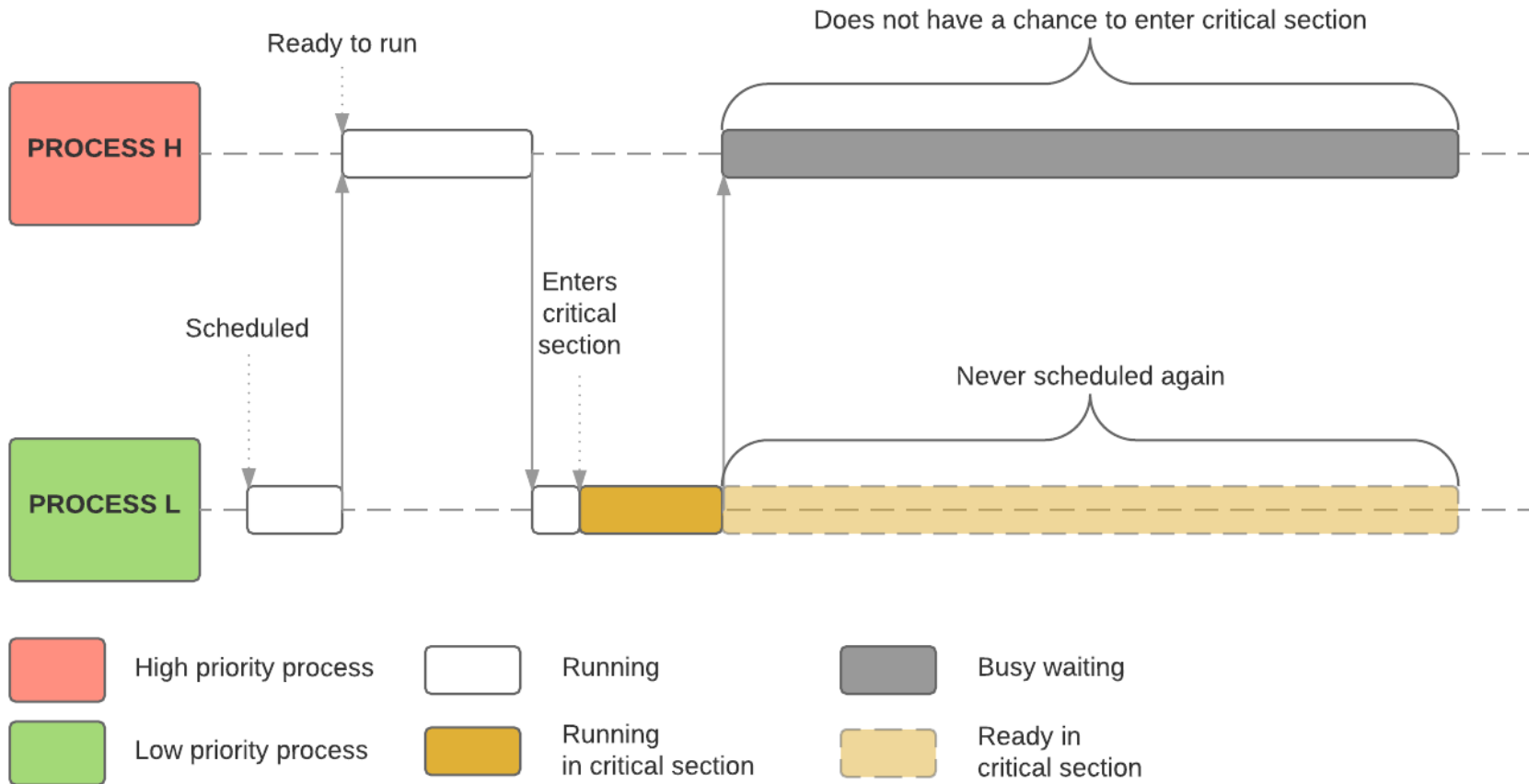
# Mutual Exclusion with Busy Waiting: The TSL Instruction (2)

- The first instruction copies the old value of lock to the register and then sets lock to 1
- The old value is compared with 0:
  - If it is nonzero, the lock was already set by another program
  - Sooner or later it will become 0 and the subroutine returns, with the lock set
- To clear the lock the program just stores a 0 in *lock*

# Sleep and Wakeup (1)

- Both Peterson's solution and the solutions using TSL or XCHG are correct, but both have the **defect of requiring busy waiting**
- It can also have unexpected effects:
  - Consider two processes -  $H$  (high priority) and  $L$  (low priority)
  - According to the scheduling rules  $H$  is run whenever it is in ready state
  - Eventually,  $H$  becomes ready to run, but  $L$  is in its critical region,
  - $H$  begins busy waiting,  $L$  is never scheduled while  $H$  is running
  - Both  $L$  and  $H$  never get the chance to proceed

# Sleep and Wakeup (2)



## Priority inversion problem

# Sleep and Wakeup (3)

- This situation is sometimes referred to as the **priority inversion problem**
- One of the simplest interprocess communication primitives that block instead of wasting CPU time is the pair of system calls **sleep** and **wakeup**:
  - **Sleep** causes the caller to be suspended (blocked) until another process wakes it up
  - **Wakeup** accepts the process to be awakened as a parameter and wakes this process up (supposedly, changes its state to *Ready*)

# Sleep and Wakeup

## The Producer-Consumer Problem (1)

- The producer-consumer problem (bounded-buffer problem):
  - Two processes share a common, fixed-size buffer
  - One of them, the producer, puts information into the buffer
  - The other one, the consumer, takes it out
  - When the producer wants to put a new item in the buffer, but it is already full, it goes to sleep
  - Similarly, if the consumer wants to remove an item from the buffer and sees that the buffer is empty, it goes to sleep too

# Sleep and Wakeup

## The Producer-Consumer Problem (2)

- To keep track of the number of items in the buffer, we will need a variable, *count*
- The producer's code will first test to see if *count* is  $N$  (the maximum number of items in the buffer)
- If it is, the producer will go to sleep; if it is not, the producer will add an item and increment *count*
- Consumer first tests *count* to see if it is 0
- If it is, go to sleep; if it is nonzero, remove an item and decrement the counter
- Each of the processes also tests to see if the other should be awakened, and if so, wakes it up



# Sleep and Wakeup

## The Producer-Consumer Problem (3)



count = 0



Buffer

# Sleep and Wakeup

## The Producer-Consumer Problem (3)



count = 0



Buffer

# Sleep and Wakeup

## The Producer-Consumer Problem (3)



count = 0



Buffer

# Sleep and Wakeup

## The Producer-Consumer Problem (3)



count = 8



Buffer

# Sleep and Wakeup

## The Producer-Consumer Problem (3)



count = 8



Buffer

# Sleep and Wakeup

## The Producer-Consumer Problem (3)



count = 0



Buffer

# Sleep and Wakeup

## The Producer-Consumer Problem (4)

- Race condition can occur because access to *count* is unconstrained
- The following situation could possibly occur:
  - The buffer is empty and the consumer has just read **count** to see if it is 0 (it hasn't fallen asleep yet)
  - The scheduler decides to stop running the consumer temporarily and start running the producer
  - The producer inserts an item in the buffer, increments count, and notices that it is now 1
  - The producer calls wakeup to wake the consumer up

# Sleep and Wakeup

## The Producer-Consumer Problem (5)

- Since the consumer is not yet logically asleep, the wakeup signal will be lost
- When the consumer next runs, it will test the value of count it previously read, find it to be 0, and go to sleep
- Sooner or later the producer will fill up the buffer and also go to sleep
- Both will sleep forever



# Sleep and Wakeup

## The Producer-Consumer Problem (6)



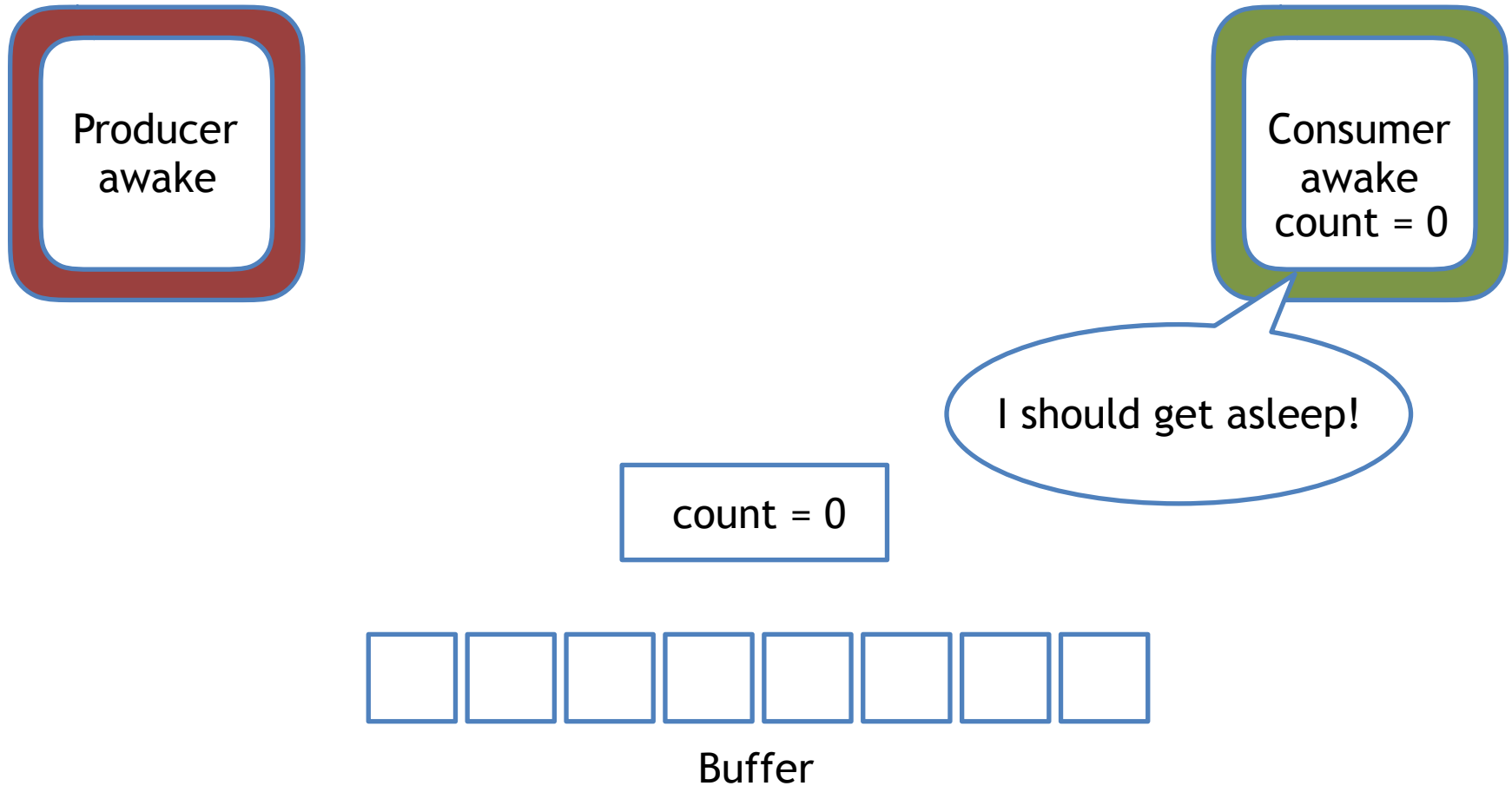
count = 0



Buffer

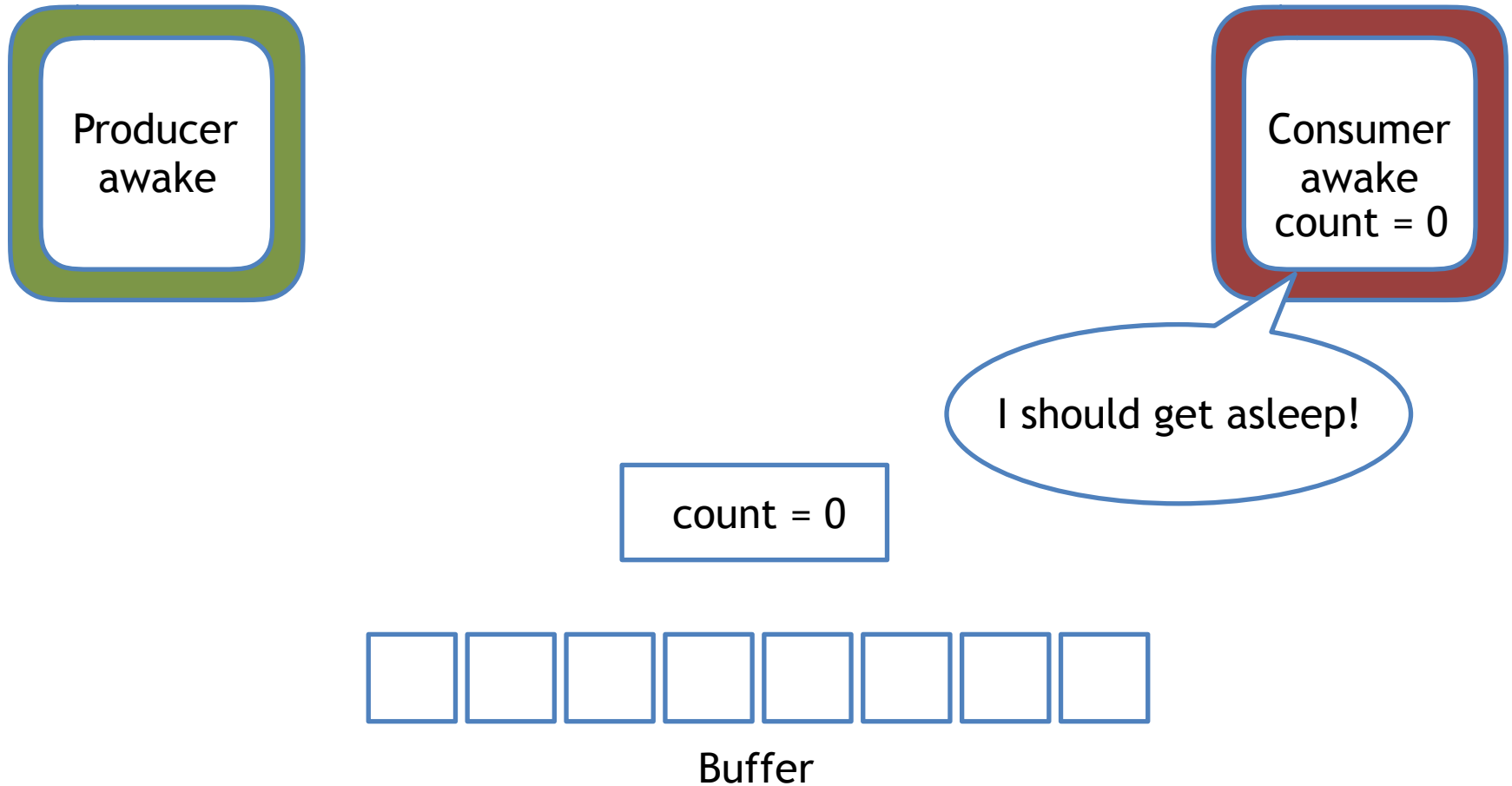
# Sleep and Wakeup

## The Producer-Consumer Problem (6)



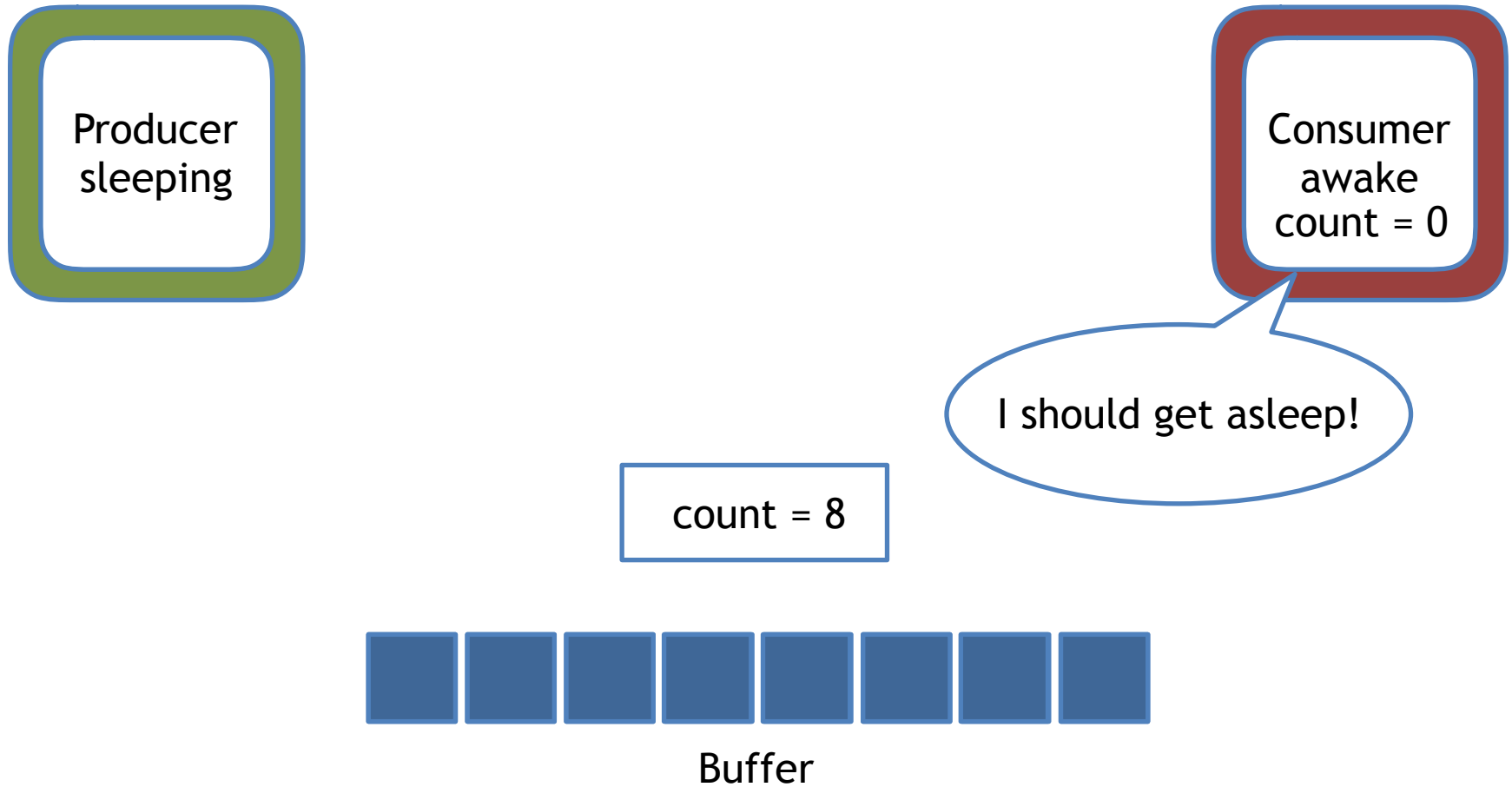
# Sleep and Wakeup

## The Producer-Consumer Problem (6)



# Sleep and Wakeup

## The Producer-Consumer Problem (6)



# Sleep and Wakeup

## The Producer-Consumer Problem (6)



count = 8



Buffer

# Sleep and Wakeup

## The Producer-Consumer Problem (7)

- To fix this situation, it is possible to add a **wakeup waiting bit**:
  - When a wakeup is sent to a process that is still awake, this bit is set
  - When the process tries to go to sleep, if the wakeup waiting bit is on, it will be turned off, but the process will stay awake
  - The consumer clears the wakeup waiting bit in every iteration of the loop
- However, it is easy to construct examples with three or more processes in which one wakeup waiting bit is insufficient. Even when using more bits, the essence of the problem is still here

# Semaphores (1)

- Semaphore:
  - Proposed by Dijkstra In 1965
  - an integer variable to count the number of wake-ups saved for future use - a **semaphore**
  - has a positive value if one or more wakeups were pending, 0 otherwise
  - two operations: **down** and **up** (generalizations of **sleep** and **wakeup**, respectively)
  - the down operation checks if the value is greater than 0. If so, it decrements the value (i.e., uses up one stored wakeup) and just continues

# Semaphores (2)

- If the value is 0, the process is put to sleep without completing the down for the moment
- Checking the value, changing it, and possibly going to sleep, are all done as a single, indivisible atomic action
- It is guaranteed that once a semaphore operation has started, no other process can access the semaphore until the operation has completed or blocked
- This atomicity is absolutely essential to solving synchronization problems and avoiding race conditions



# Semaphores (3)

- Semaphores solve the lost-wakeup problem
- To make them work correctly, it is essential that they be implemented in an indivisible way
- The normal way is to implement *up()* and *down()* as system calls, with the OS briefly disabling all interrupts while it is testing the semaphore, updating it, and putting the process to sleep, if necessary
- All of these actions take only a few instructions, so no harm is done in disabling interrupts
- If multiple CPUs are being used, each semaphore should be protected by a *lock variable* with the TSL or XCHG instructions

# Semaphores (4)

- The solution uses three semaphores
  - one called *full* for counting the number of slots that are full
  - one called *empty* for counting the number of slots that are empty
  - one called *mutex* to make sure the producer and consumer do not access the buffer at the same time
- *Full* is initially 0, *empty* is initially equal to the number of slots in the buffer, and *mutex* is initially 1
- Semaphores that are initialized to 1 and used by two or more processes to ensure that only one of them can enter its critical region at the same time are called **binary semaphores**
- If each process does a down just before entering its critical region and an up just after leaving it, mutual exclusion is guaranteed

# Semaphores (5)

- We used semaphores in two different ways:
  - The *mutex* semaphore is used for mutual exclusion. It is designed to guarantee that only one process at a time will be reading or writing the buffer and the associated variables
  - The other use of semaphores is for **synchronization**. The *full* and *empty* semaphores are needed to guarantee that certain event sequences do or do not occur. In this case, they ensure that the producer stops running when the buffer is full, and that the consumer stops running when it is empty

# Mutexes (1)

- A **mutex** is a shared variable that can be in one of two states: unlocked or locked
- Only 1 bit is required to represent it, but in practice an integer often is used, with 0 meaning unlocked and all other values meaning locked
- Mutexes are good only for managing mutual exclusion to some shared resource or piece of code
- They are easy and efficient to implement, which makes them especially useful in thread packages that are implemented entirely in user space

# Mutexes (2)

- Two procedures are used with mutexes:
  - When a thread (or process) needs access to a critical region, it calls ***mutex\_lock***. If the mutex is currently unlocked, the call succeeds and the calling thread is free to enter the critical region
  - If the mutex is already locked, the calling thread is blocked until the thread in the critical region is finished and calls ***mutex\_unlock***. If multiple threads are blocked on the mutex, one of them is chosen at random
- Mutexes can easily be implemented in user space provided that a TSL or XCHG instruction is available (Fig. 2-29)

# Mutexes (3)

- *mutex\_lock* is similar to the code of *enter\_region* that was provided as an example of TSL instruction (Fig. 2-25), but there is a crucial difference:
  - When *enter\_region* fails to enter the critical region, it keeps testing the lock repeatedly (busy waiting)
  - With (user) threads, the situation is different because there is no clock that stops threads that have run too long
  - A thread that tries to acquire a lock by busy waiting will loop forever and never acquire the lock because it never allows any other thread to run and release the lock

# Mutexes (4)

- When the *mutex\_lock* fails to acquire a lock, it calls *thread\_yield* to give up the CPU to another thread - there is no busy waiting
- Since *thread\_yield* is just a call to the thread scheduler in user space, it is very fast.
- Neither mutex lock nor mutex unlock requires any kernel calls
- Using them, user-level threads can synchronize entirely in user space using procedures that require only a handful of instructions

# Mutexes (5)

- The mutex system just described is a bare-bones set of calls
- With all software, there is always a demand for more features, and synchronization primitives are no exception
- For example, sometimes a thread package offers a call *mutex\_trylock* that either acquires the lock or returns a code for failure, but does not block



# Mutexes (6)

- When we work with user-space threads, there is no problem with multiple threads having access to the same mutex
- If processes have disjoint address spaces, how can they share the *turn* variable, semaphores or a common buffer?
  - Some of the shared data structures, such as the semaphores, can be stored in the kernel and accessed only by means of system calls
  - Most modern OSs offer a way for processes to share some portion of their address space with other processes. In this way, buffers and other data structures can be shared

# Monitors (1)

- A **monitor** is a higher-level synchronization primitive - it is a collection of procedures, variables, and data structures that are all grouped together in a special kind of module or package (Fig. 2-33)
- Processes may call the procedures in a monitor whenever they want to, but they cannot directly access the monitor's internal data structures from procedures declared outside the monitor

# Monitors (2)

```
monitor example
    integer i;
    condition c;

    procedure producer( );
    .
    .
    end;

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

Figure 2-33. A monitor.

# Monitors (3)

```
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;
```




Figure 2-34. An outline of the producer-consumer problem with monitors. Only one monitor procedure at a time is active. The buffer has  $N$  slots.

# Monitors (4)

```
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;
```

Figure 2-34. An outline of the producer-consumer problem with monitors. Only one monitor procedure at a time is active. The buffer has N slots.

# Monitors (5)

- Only one process can be active in a monitor at any moment
- When a process calls a monitor procedure, the first few instructions of the procedure will check to see if any other process is currently active within the monitor
- If no other process is using the monitor, the calling process may enter
- It is up to the compiler to implement mutual exclusion on monitor entries, but a common way is to use a mutex or a binary semaphore

# Monitors (6)

- We also need a way for processes to block when they cannot proceed
- The solution lies in the introduction of condition variables, along with two operations on them, *wait* and *signal*
- When a monitor procedure discovers that it cannot continue (e.g., the producer finds the buffer full), it does a *wait* on some condition variable. This action causes the calling process to block
- The other process, for example, the consumer, can wake up its sleeping partner by doing a *signal* on the condition variable that its partner is waiting on

# Monitors (7)

- To avoid having two active processes in the monitor at the same time, we need a rule telling what happens after a signal:
  - One solution is letting the newly awakened process run, suspending the other one
  - The second one is requiring that a process doing a signal must exit the monitor immediately (a signal statement may appear only as the final statement in a monitor procedure)
  - The third one is to let the signaler continue to run and allow the waiting process to start running only after the signaler has exited the monitor



# Monitors (8)

- The producer and consumer threads are functionally identical to their counterparts in all previous examples
- The interesting part of this program is the class *our\_monitor*, which holds the buffer, the administration variables, and two synchronized methods
- When the producer is active inside *insert*, it knows for sure that the consumer cannot be active inside *remove*, making it safe to update the variables and the buffer without fear of race conditions
- The variable *count* (any value from 0 to  $N - 1$ ) keeps track of how many items are in the buffer
- The variable *lo* is the index of the buffer slot where the next item is to be fetched
- *hi* is the index of the buffer slot where the next item is to be placed
- It is permitted that  $lo = hi$ , which means that either 0 items or  $N$  items are in the buffer. The value of *count* tells which case holds

# Monitors (9)

- By making the mutual exclusion of critical regions automatic, monitors make parallel programming much less error prone than using semaphores
- Nevertheless, they too have some drawbacks:
  - Monitors are a **programming-language concept**. C, Pascal, and most other languages do not have monitors, so it is unreasonable to expect their compilers to enforce any mutual exclusion rules
  - Another problem with monitors, and also with semaphores, is that they were designed for solving the mutual exclusion problem on one or more CPUs that all **have access to a common memory**. In a distributed system consisting of multiple CPUs, each with its own private memory, these primitives become inapplicable

# Message Passing

- **message passing** is the method of IPC that uses two primitives, *send* and *receive*, which, like semaphores and unlike monitors, are system calls rather than language constructs (Fig. 2-36)
- They can easily be put into library procedures, such as
  - `send(destination, &message);`
  - `receive(source, &message);`
- The former call sends a message to a given destination and the latter one receives a message from a given source (or from ANY, if the receiver does not care)
- If no message is available, the receiver can block until one arrives. Alternatively, it can return immediately with an error code
- The main issue: **messages can be lost by the network**. The receiver might not be able to distinguish a new message from the retransmission of an old one

End

Week 05 - Lecture

# References

- Tanenbaum & Bos, Modern Operating Systems: 4th edition, 2013  
Prentice-Hall, Inc.