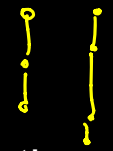
Operating Systems Design
6. Synchronization

Paul Krzyzanowski
pxk@cs.rutgers.edu

Concurrency



Concurrent threads/processes (informal)

- Two processes are concurrent if they run at the same time or if their execution is interleaved *in any order*

Asynchronous

- The processes require occasional synchronization

Independent

- They do not have any reliance on each other

Synchronous

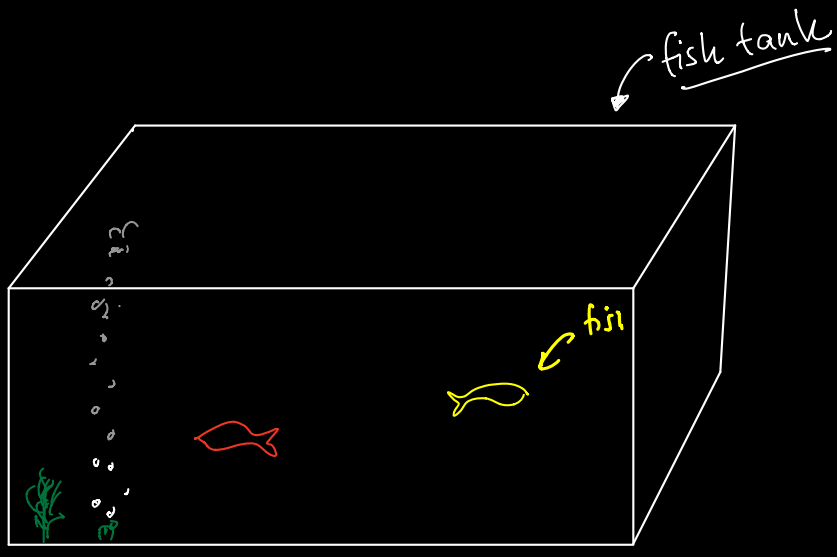
- Frequent synchronization with each other – order of execution is guaranteed

Parallel

- Processes run at the same time on separate processors

- What happens when things occur in parallel?

- Well, fish die!



acquire(fish_lock)

You

1. Ask mom if fish have been fed
- if not:
  - 2 Get the food
  - 3 Feed the fish
  - 5 Tell mom!

release(fish_lock)

You annoying brother

4 Ask mom if fish have been fed
- if not:
  - 6 Get the food
  - 7 Feed the fish
  - 8 Tell mom!

Fish dead

You

1 if light is red
- 4 feed fish
- turn light green

AB

2 if light is red
- 3 feed fish
- turn light green

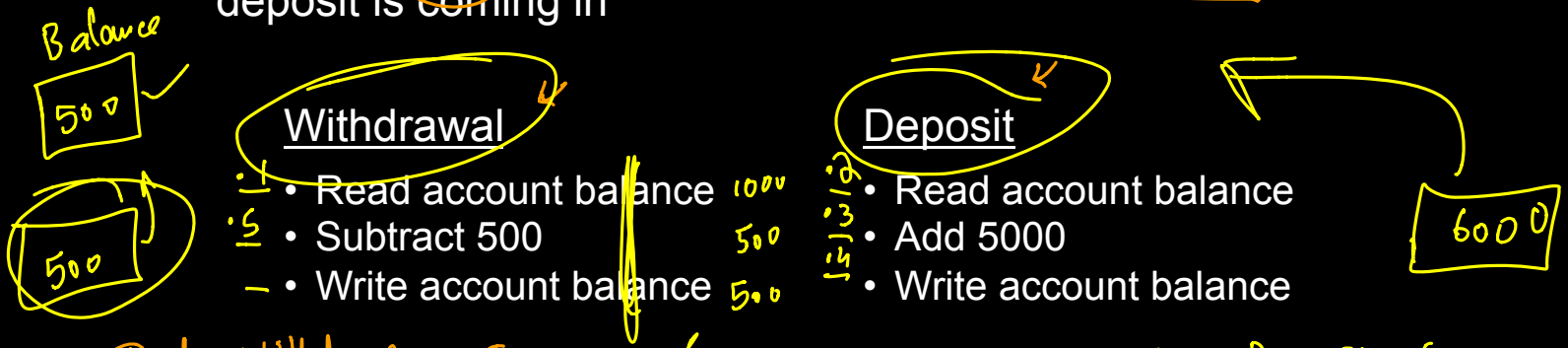
Race Conditions

A race condition is a bug:

- The outcome of concurrent threads are unexpectedly dependent on a specific sequence of events.

Example

- Your current bank balance is $1,000
- Withdraw $500 from an ATM machine while a $5,000 direct deposit is coming in

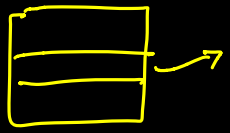


Possible outcomes:

Total balance = $5500, $500, $6000

Synchronization

Synchronization deals with developing techniques to avoid race conditions



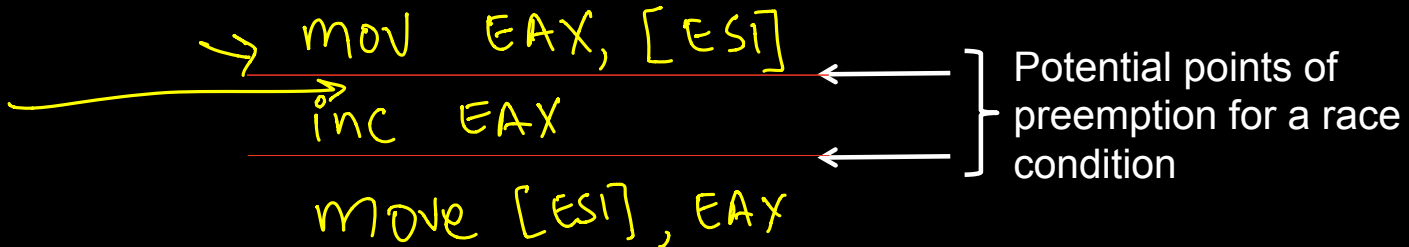
"This function is not thread-safe!

Something as simple as

```
x = x + 1;
```

x++;

May have have a race condition:



Mutual Exclusion

Critical section:

- Region in a program where race conditions can arise

Mutual exclusion:

- Allow only one thread to access a critical section at a time

Deadlock:

- A thread is perpetually blocked (circular dependency on resources)

Starvation:

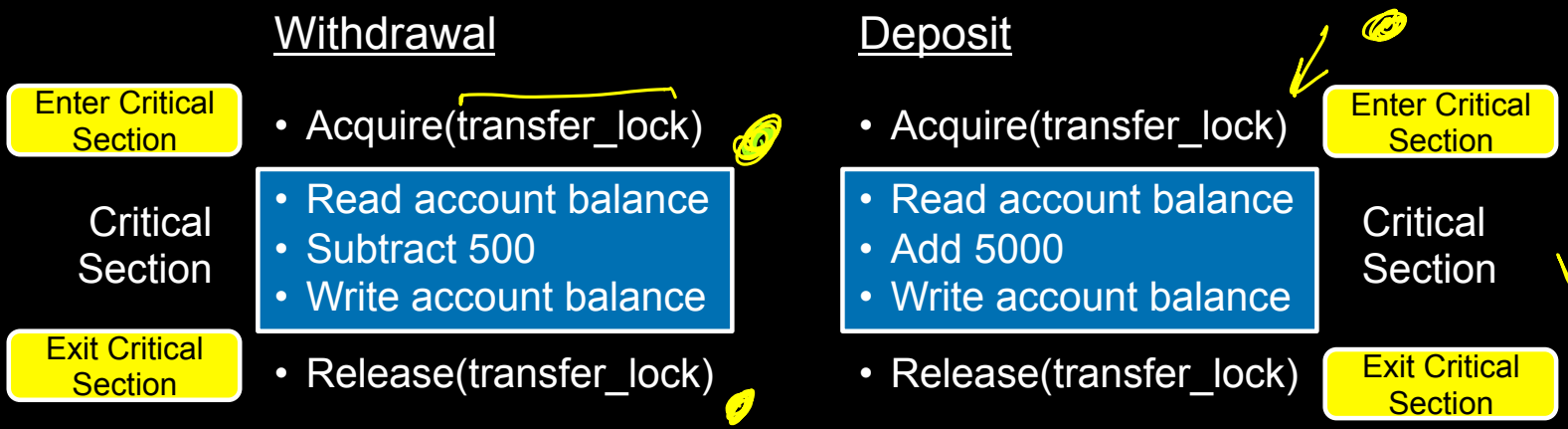
- A thread is perpetually denied resources

Livelock:

- Threads run but no progress in execution

Controlling critical section access: locks

- Grab and release locks around critical sections
- Wait if you cannot get a lock



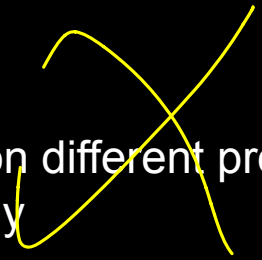
The Critical Section Problem

Design a protocol to allow threads to enter a critical section

Conditions for a solution

- Mutual exclusion: No threads may be inside the same critical sections simultaneously
- Progress: If no thread is executing in its critical section but one or more threads want to enter, the selection of a thread cannot be delayed indefinitely.
  - If one thread wants to enter, it should be permitted to enter.
  - If multiple threads want to enter, exactly one should be selected.
- Bounded waiting: No thread should wait forever to enter a critical section
- No thread running outside its critical section may block others
- A good solution will make no assumptions on:
  - No assumptions on # processors
  - No assumption on # threads/processes
  - Relative speed of each thread

Critical sections & the kernel



- Multiprocessors
  - Multiple processes on different processors may access the kernel simultaneously
  - Interrupts may occur on multiple processors simultaneously
- Preemptive kernels
  - Preemptive kernel: process can be preempted while running in kernel mode
  - Nonpreemptive kernel: processes running in kernel mode cannot be preempted (but interrupts can still occur!)
- Single processor, nonpreemptive kernel: free from race conditions

Solution #1: Disable Interrupts

Disable all system interrupts before entering a critical section and re-enable them when leaving

Bad!

- Gives the thread too much control over the system
- Stops time updates and scheduling
- What if the logic in the critical section goes wrong?
- What if the critical section has a dependency on some other interrupt, thread, or system call?
- What about multiple processors? Disabling interrupts affects just one processor

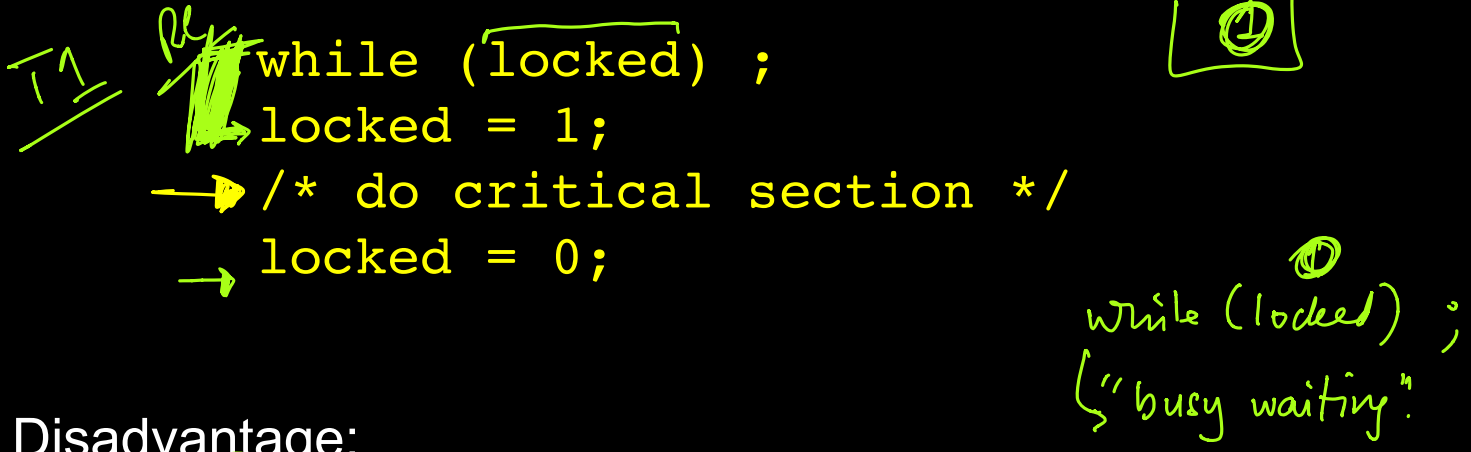
Advantage

- Simple, guaranteed to work
- Was often used in the uniprocessor kernels

Solution #2: Software Test & Set Locks

Keep a shared lock variable:

```
while (locked) ;
locked = 1;
/* do critical section */
locked = 0;
```

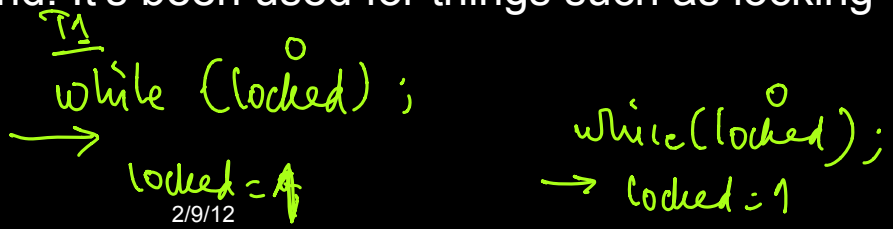


Disadvantage:

- Buggy! There's a race condition in setting the lock

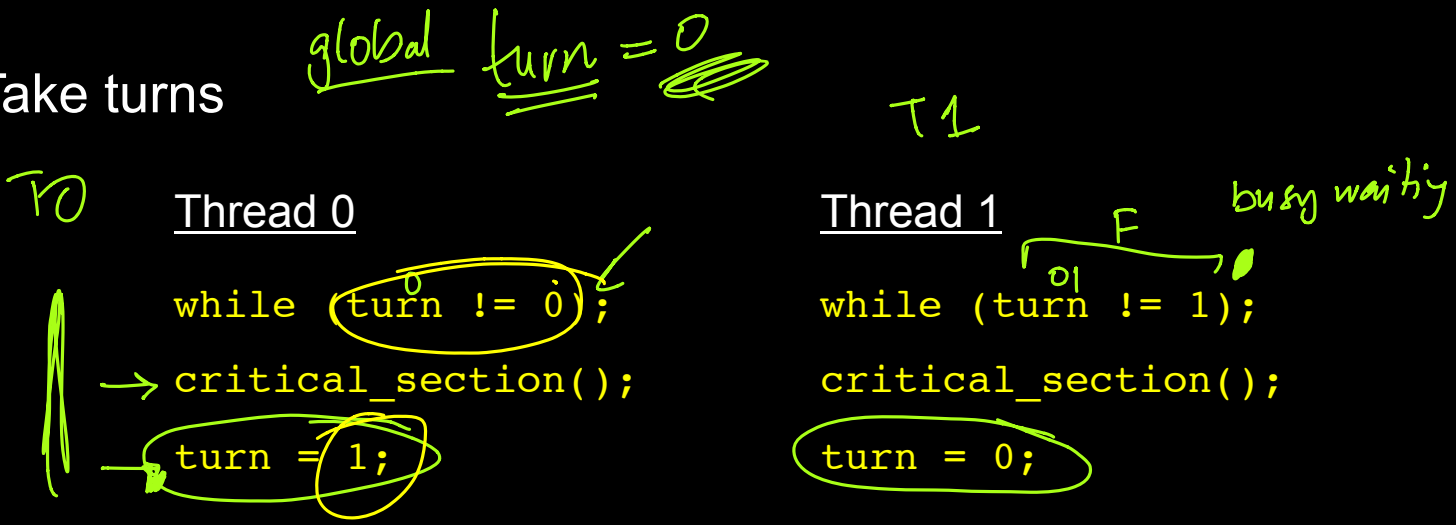
Advantage:

- Simple to understand. It's been used for things such as locking mailbox files



Solution #3: Lockstep Synchronization

Take turns



Disadvantages:

- Tight loop that spins waiting for a turn: **busy waiting** or **spin lock**
- Forces strict alternation; if thread 2 is really slow, thread 1 is slowed down with it

Software solutions for mutual exclusion

- Peterson's solution (page 221 of text)
- Others
- Disadvantages:
  - Difficult to implement correctly – have to rely on `volatile` data types to ensure that compilers don't make the wrong optimizations
  - Relies on busy waiting

Let’s turn to hardware for help

Help from the processor

Atomic (indivisible) CPU instructions that help us get locks

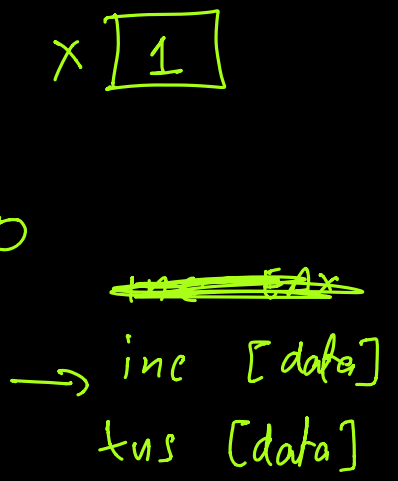
- Test-and-set
- Compare-and-swap
- Fetch-and-Increment

Test & Set

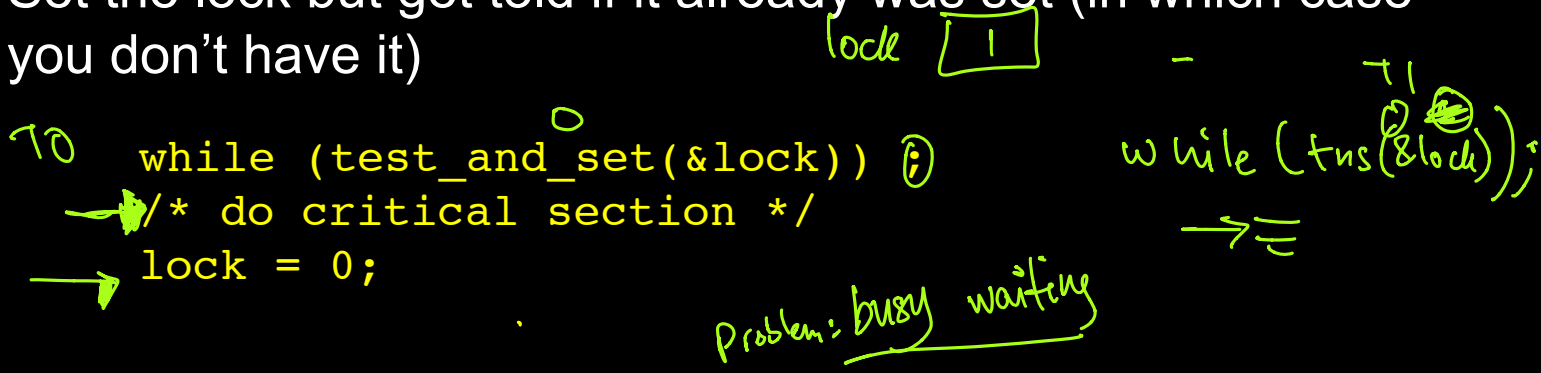
Test-and-set

```
int test_and_set(int *x) {
    last_value = *x;
    *x = 1;
    return last_value;
}
```

ATOMIC



Set the lock but get told if it already was set (in which case you don't have it)



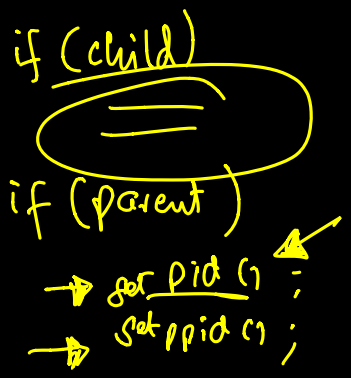
Fetch & Increment

Increment a memory location; return previous value

ATOMIC

```
int fetch_and_increment(int *x) {
    last_value = *x;
    *x = *x + 1;
    return last_value;
}
```

Compare and Swap



Fetch & Increment

Check that it's your turn for the critical section

```
ticket = 0; turn = 0;
...
myturn = fetch_and_increment(&ticket);
while (turn != myturn) ;
/* do critical section */
fetch_and_increment(&turn);
```

turn

ticket

Spin locks

- All these techniques rely on spin locks
- Wastes CPU cycles
- The process with the lock may not be allowed to run!
  - Lower priority process obtained a lock
  - Higher priority process is always ready to run but loops on trying to get the lock
  - Scheduler always schedules the higher-priority process
  - Priority inversion
    - If the low priority process would get to run & release its lock, it would then accelerate the time for the high priority process to get a chance to get the lock and do useful work
    - Try explaining that to a scheduler!

Spin locks aren't great

Can we block until we can get the lock?

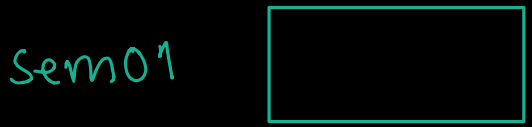
Semaphores

Count the number of threads that may enter a critical section at any given time.

- Each *down* decreases the number of future accesses
- When no more are allowed, processes have to wait *
- Each *up* lets a waiting process get in

– var
– 2 ops
– 0 – down ×

Atomic



* Not busy waiting

Semaphores

- Count # of wake-ups saved for future use
- Two atomic operations:

```
down(sem s) {
    if (s > 0)
        s = s – 1;
    else
        sleep on event s
}

up(sem s) {
    if (someone is waiting on s)
        wake up one of the threads
    else
        s = s + 1;
}
```

```
//initialize
mutex = 1;

down(&mutex)

// critical section

up(&mutex)
```

Binary semaphore (mutex)

T1

1. down (s)

CR

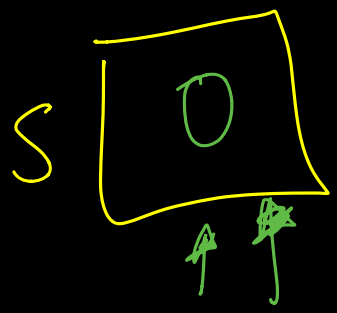
up (s)

T2

down (s)

CR

up (s)



Producer-Consumer example

- Producer
  - Generates items that go into a buffer
  - Maximum buffer capacity = N
  - If the producer fills the buffer, it must wait (sleep)
- Consumer
  - Consumes things from the buffer
  - If there's nothing in the buffer, it must wait (sleep)
- This is also known as the *Bounded-Buffer Problem*

NATIONAL UNIVERSITY
OF COMPUTER & EMERGING SCIENCES
PESHAWAR CAMPUS

Problem Set:	Assignment: A05	**Semester:**	Spring 2019
Points:	*10*		
Date Set:	*See SLATE*	**Due Date:**	*See SLATE*
Course:	CS217 - OOP	**Instructor:**	Dr. Nauman

1 Queue Writers and Queue Readers

We've already implemented a queue. Now we are going to put it to use. This is a very common case study in the real world that you have some entity that puts things to be processed in a queue and another entity takes stuff from the queue and processes it. The first entity is called a Writer (since it writes to the queue) and the second is called Reader (because it reads stuff from the queue).

1.1 Tasks 1: Make Queue Generic

The first thing you need to do is to take the implementation of the class Queue we did earlier and generalize it using class templates. After you're done, the Queue class should be able to handle either `int` or `string` type values. The rest of the FIFO logic should, of course, remain the same.

1.2 Task 2: The Writer

In this task, you need to create a separate class called `Writer`. This class should have one public function with the following signature:

```
void process_file(string filename, Queue<string> *q);
```

This function reads the contents of the file (given as `filename`) one line at a time. The line that is read should be *enqueued* in `q`. That's all this function does!

1.3 Task 3: The Reader

For this task, create yet another class called `Reader` that has one public function as follows:

```
void process_queue(Queue<string> *q);
```

The logic for this function is that it should continue to *dequeue* from `q` until there is nothing to dequeue any more. Whenever it dequeues a string, it should simply output its length. For example:

```
String:  [Abstraction] has length:  11
```

q

15	21	39			
0	1	2	3	4	5
out		in		in	

T1:

enqueue:
1. place value at in
4. increment in

T2:

enqueue:
2. place value at in
3. increment in

acquire (lock)
enqueue();
release (lock)

down (&mutex)
enqueue
up (&mutex)

2nd problem: constant annoying polling

empty 0

full 5

producer:
down (empty)
up (full)

consumer
down (full)
up (empty)

Producer-Consumer example

N = 5

```
sem mutex=1, empty=N, full=0;
producer() {
    for (;;) {
        produce_item(&item);    /* produce something */
        down(&empty);           /* decrement empty count */
        down(&mutex);           /* start critical section */
        enter_item(item);       /* put item in buffer */
        up(&mutex);             /* end critical section */
        up(&full);              /* +1 full slot */
    }
}
consumer() {
    for (;;) {
        down(&full);            /* one less item */
        down(&mutex);           /* start critical section */
        remove_item(item);      /* get the item from the buffer */
        up(&mutex);             /* end critical section */
        up(&empty);             /* one more empty slot */
        consume_item(item);     /* consume it */
    }
}
```

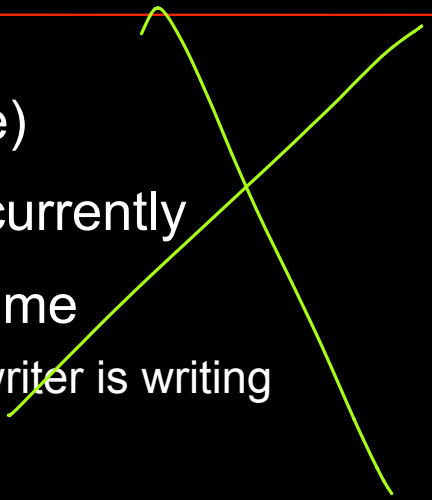
Counting semaphore

lock

lock

Readers-Writers example

- Shared data store (e.g., database)
- Multiple processes can read concurrently
- Only one process can write at a time
  - And no readers can read while the writer is writing



Condition Variables / Monitors

- Higher-level synchronization primitive → operation
- Implemented by the programming language / APIs
- Two operations:
  - *wait* (*condition_variable*)
    - Block until *condition_variable* is "signaled"
  - *signal*(*condition_variable*)
    - Wake up **one** process that is waiting on the condition variable
    - Also called *notify*

Producer:
wait(empty)
signal (full)

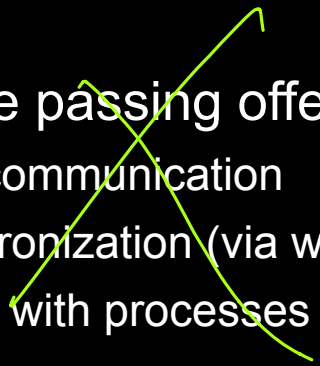
Consumer:
wait(full)
signal (empty)

Synchronization
Part II: Inter-Process Communication

Communicating processes

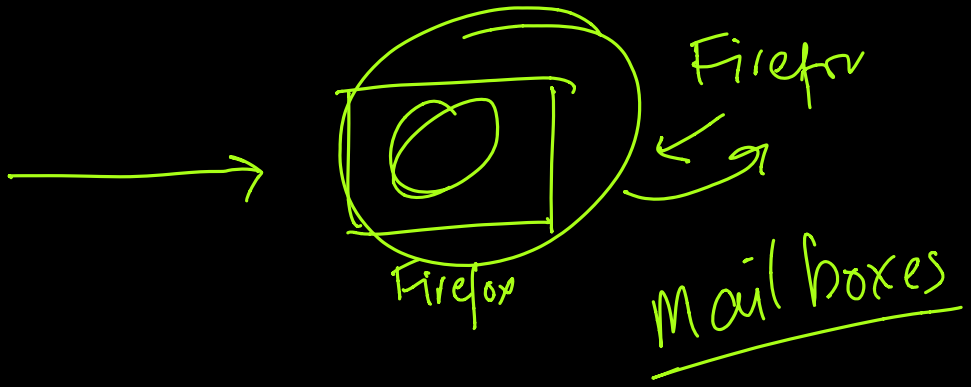
- Must:
  - Synchronize
  - Exchange data

- Message passing offers:
  - Data communication
  - Synchronization (via waiting for messages)
  - Works with processes on different machines



Message passing

- Two primitives:
  - *send*(destination, message)
  - *receive*(source, message)
- ~~Operations may or may not be blocking~~



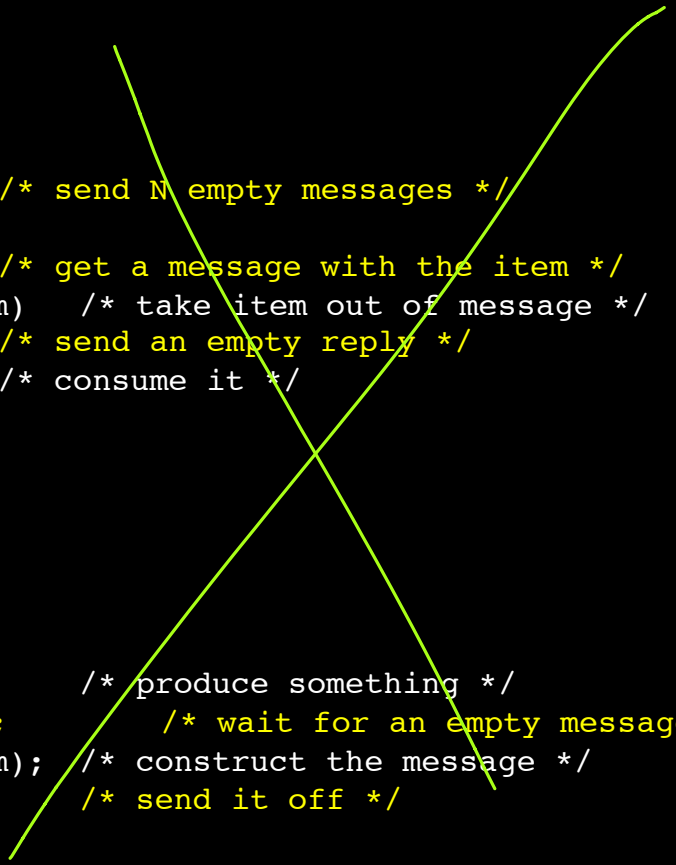
Producer-consumer example

```
#define N 4       /* number of slots in the buffer */

consumer() {
     int item, i;
     message m;

     for (i=0; i < N; ++i)
          send(producer, &m);   /* send N empty messages */
     for (;;) {
          receive(producer, &m) /* get a message with the item */
          extract_item(&m, &item)   /* take item out of message */
          send(producer, &m);   /* send an empty reply */
          consume_item(item);   /* consume it */
     }
}
producer() {
     int item;
     message m;

     for (;;) {
          produce_item(&item);      /* produce something */
          receive(consumer, &m);        /* wait for an empty message */
          build_message(&m, item);  /* construct the message */
          send(consumer, &m);       /* send it off */
     }
}
```



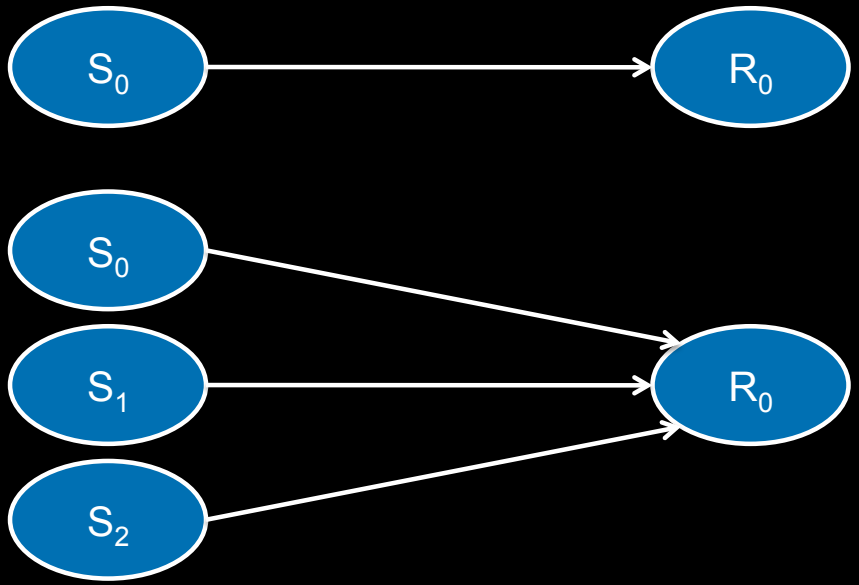
Messaging: Rendezvous

- Sending process blocked until receive occurs
- Receive blocks until a send occurs

- Advantages:
  - No need for message buffering if on same system
  - Easy & efficient to implement
  - Allows for tight synchronization
- Disadvantage:
  - Forces sender & receiver to run in lockstep

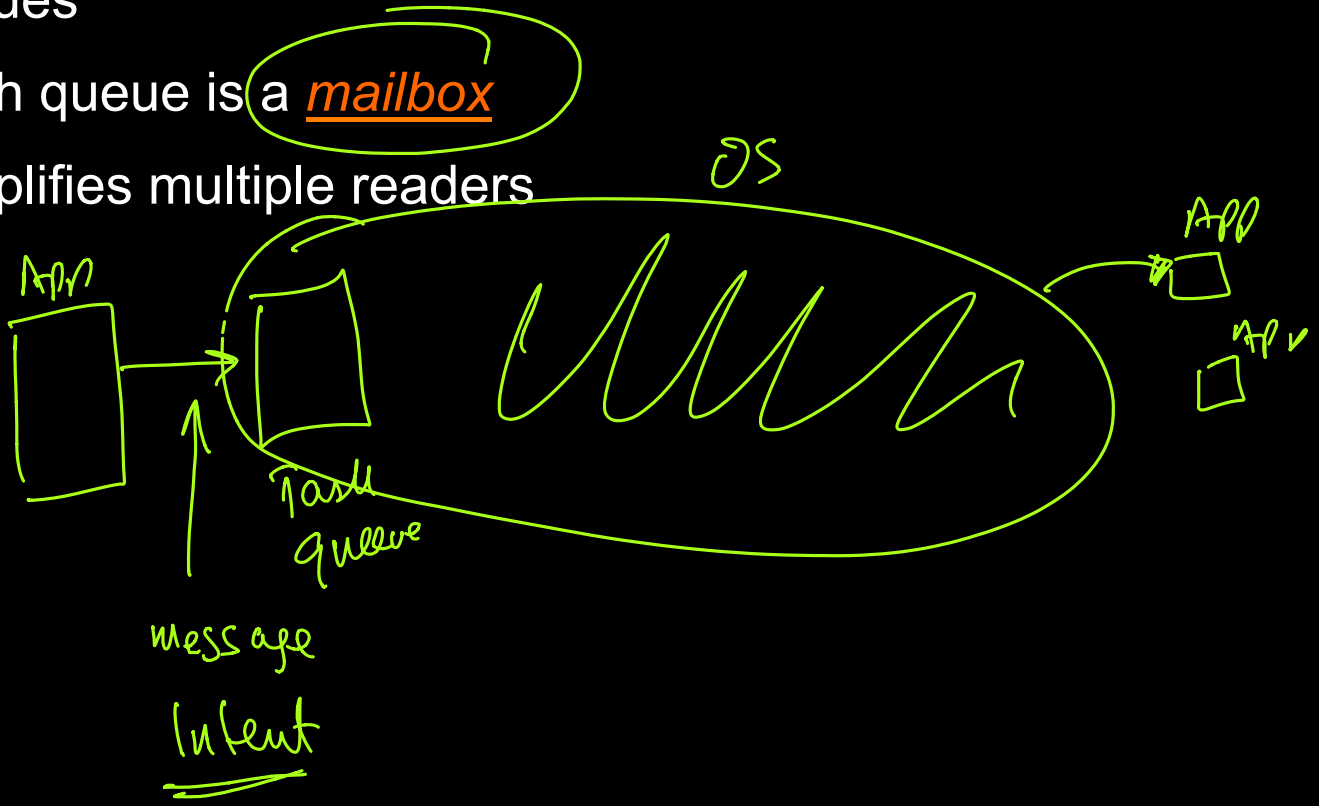
Messaging: Direct Addressing

- Sending process identifies receiving process
- Receiving process can identify sending process
  - Or can receive it as a parameter

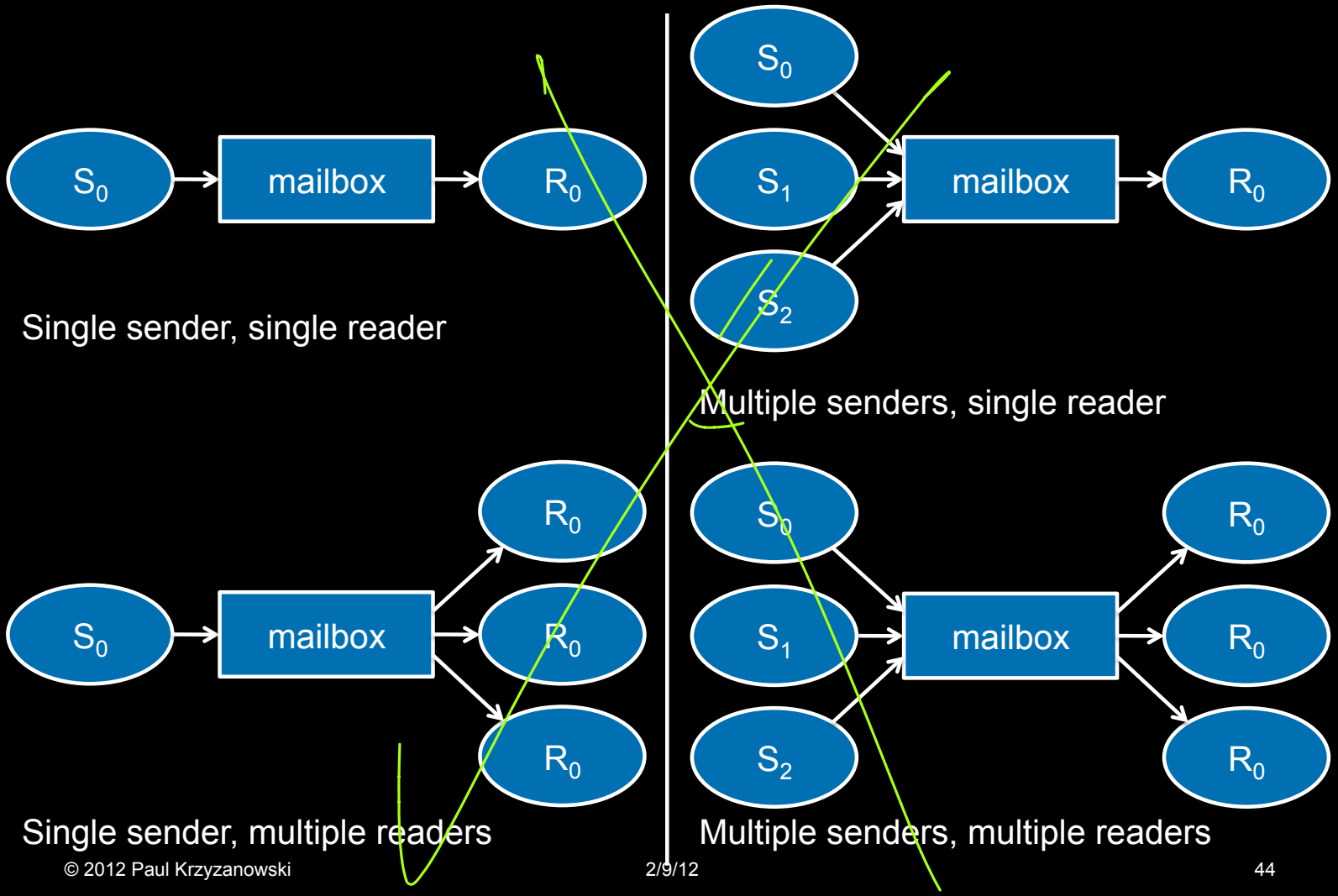


Messaging: Indirect Addressing

- Messages sent to an intermediary data structure of FIFO queues
- Each queue is a *mailbox*
- Simplifies multiple readers



Mailboxes


S_0
mailbox
R_0
Single sender, single reader
S_0
S_1
S_2
mailbox
R_0
Multiple senders, single reader
S_0
mailbox
R_0
R_0
R_0
Single sender, multiple readers
S_0
S_1
S_2
mailbox
R_0
R_0
R_0
Multiple senders, multiple readers

Example: What you find in the world

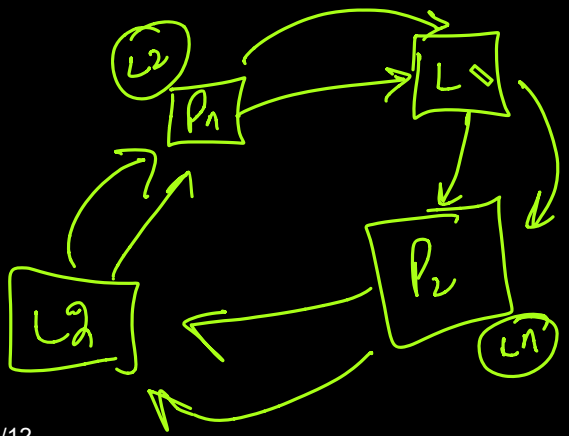
IPC mechanisms in Windows

- Clipboard: central repository for sharing data among apps
- Dynamic Data Exchange (DDE) – [older] allows apps to exchange data in various formats; extension of clipboard
- COM: automatically start another app to access data via interfaces
- Data Copy: Windows messaging – two cooperating processes can copy data
- File Mapping: Memory mapped files
- Mailslots: one-way communication of short messages (including network)
- Anonymous pipes: for I/O redirection
- Named pipes
- RPC (remote procedure call)
- Sockets
- Semaphore objects

Linux

IPC Mechanisms in Linux

- Signals
- Pipes
- Named pipes (FIFOs)
- Semaphores
- Message queues: one or more writers & one or more readers
- Shared memory
- Memory-mapped files
- RPC (remote procedure calls)
- Sockets

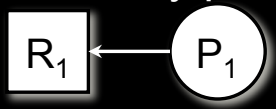


Deadlocks

- Resource allocation graph
  - Resource R_1 is allocated to process P1: assignment edge

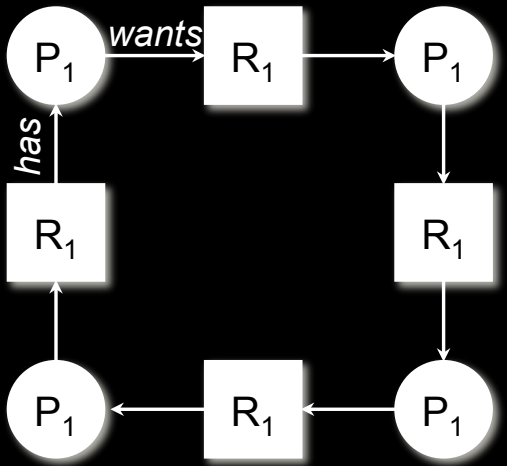
  - Resource R_1 is requested by process P_1: request edge

:



- Deadlock is present when the graph has cycles

Deadlock example



Circular dependency among four processes and four resources

Deadlocks

Four conditions must hold

1. Mutual exclusion
2. Hold and wait
3. Non-preemption of resources
   - Resources can only be released voluntarily
4. Circular wait

Dealing with deadlock

- Deadlock prevention
  - Ensure that at least one of the necessary conditions cannot hold
- Deadlock avoidance
  - Provide advance information to the OS on which resources a process will request.
  - OS can then decide if the process should wait
- Ignore the problem
  - Let the user deal with it (most common solution)

Banker's Algorithm

The End