Operating Systems

Lecture 4 - Concurrency and Synchronization

Adrien Krähenbühl

Master of Computer Science
PUF - Hồ Chí Minh

2016/2017

Mutual exclusion

Hardware solutions

Semaphores

IPC: Message passing

Deadlocks

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

Race conditions

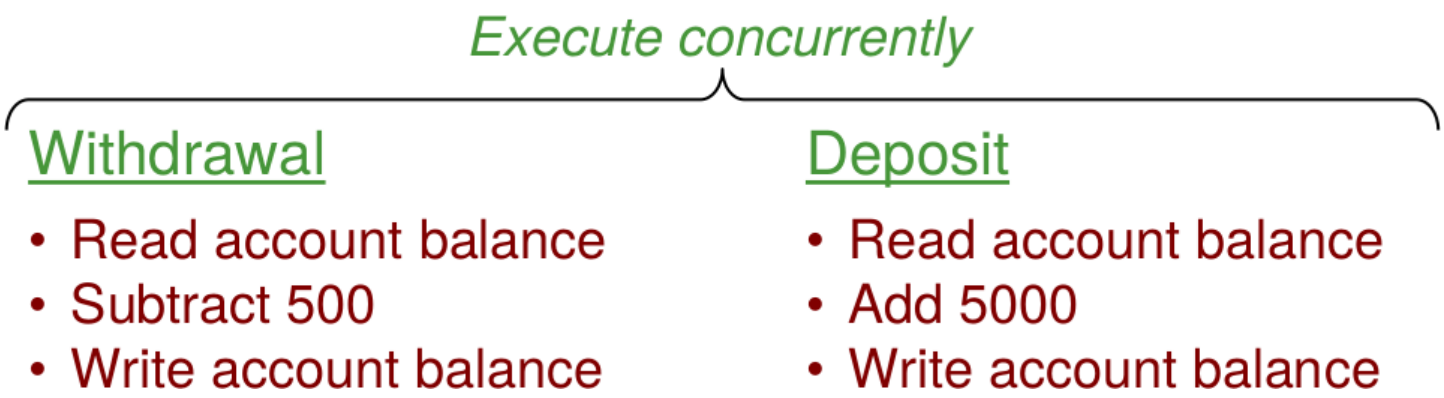
A race condition is a bug:

- ✓ 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

Execute concurrently:



Possible total balance:

Race conditions

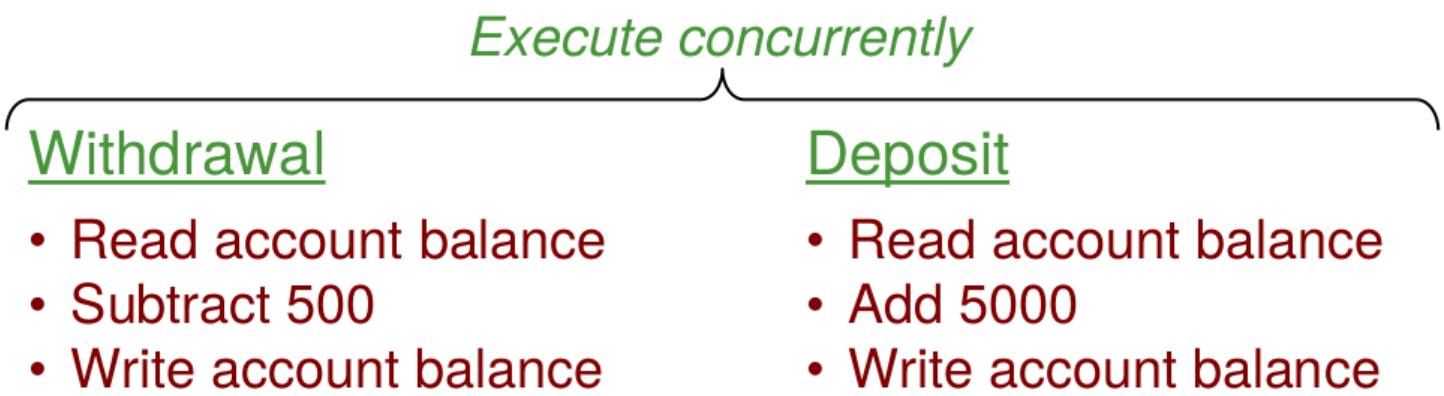
A race condition is a bug:

- ✓ 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

Execute concurrently:



Possible total balance: $5500 or $500 or $6000

Synchronization

- ✓ Synchronization deals with developing techniques to avoid race conditions

Something as simple as

$$x = x + 1;$$

Compiles to this and may cause a race condition:

```
movl _x (%rip), %eax
                <----------| Potential points of
addl $1, %eax              | preemption for a
                <----------| race condition
movl %eax, _x (%rip)
```

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

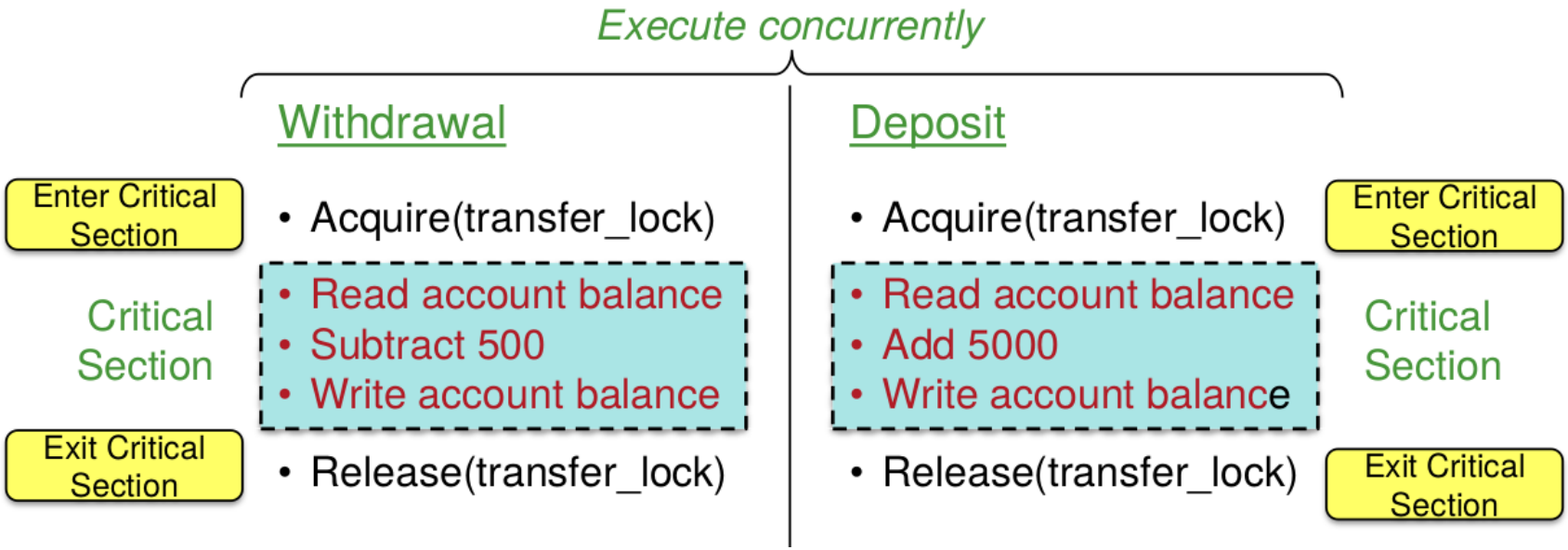
A thread is perpetually denied resources

Livelock

Threads run but with no progress in execution

Avoid race conditions with 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 (the scheduler may preempt a process even if it is running in the kernel)
  - Non preemptive kernel: processes running in kernel mode cannot be preempted (but interrupts can still occur!)
- ✓ Single processor, non preemptive 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

Thread 0

```
while (turn != 0);
critical_section();
turn = 1;
```

Thread 1

```
while (turn != 1);
critical_section();
turn = 0;
```

Disadvantage: Forces strict alternation.

- ✓ if thread 2 is really slow, thread 1 is slowed down with it.
- ✓ Turns **asynchronous threads** into **synchronous threads**

Software solutions for mutual exclusion

Peterson's solution, Dekker's, & others

Disadvantages:

- ✓ Difficult to implement correctly
  - Have to rely on volatile data types to ensure that compilers don't make the wrong optimizations
- ✓ Difficult to implement for an arbitrary number of threads

Mutual exclusion

Hardware solutions

Semaphores

IPC: Message passing

Deadlocks

Help from the processor

Atomic (indivisible) CPU instructions that help us get locks

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

These instructions execute in their entirety: they cannot be interrupted or preempted partway through their execution

Test & Set

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

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

How you use it to lock a critical section (i.e., enforce mutual exclusion):

```
// 1) spin
while (test_and_set(&lock) == 1) ;
/* do critical section */
// 2) release the lock
lock = 0;
```

Compare & swap (CAS)

Compare the value of a memory location with an old value. If they match then replace with a new value.

```
int CaS(int *x, int old, int new) {
    int save = *x;
    if (save == old) *x = new;
    return save;
}
```

Atomic

How you use it to grab a critical section?

- ✓ It avoid the race condition → set locked to 1 only if locked was still set to 0.

```
// 1) spin until locked == 0
while( CaS( &locked, 0, 1 ) != 0 );
// If we got here, locked == 1 and we have it
/* do critical section */
// 2) release the lock
locked = 0;
```

Fetch & increment

✓ Increment a memory location and return previous value

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

Ticket lock check that it's your turn for the critical section.

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

The problem with spin locks

- ✓ All these solutions require busy waiting
  - Tight loop that spins waiting for a turn: **busy waiting** or **spin lock**
- ✓ Nothing useful gets done!
  - Wastes CPU cycles

Priority inversion

- ✓ Spin locks may lead to **priority inversion**
- ✓ The process with the lock may not be allowed to run!
  - Suppose a 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 and 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!

Priority Inheritance

- ✓ Technique to avoid priority inversion
- ✓ Increase the priority of any process in a critical section to the maximum of any process waiting on any resource for which the process has a lock
- ✓ When the lock is released, the priority goes to its normal level

Mutual exclusion

Hardware solutions

Semaphores

IPC: Message passing

Deadlocks

Spin locks aren't great

How about this?

```
public class Lock
{
    private int val = UNLOCKED;
    private ThreadQueue waitQueue = new ThreadQueue();

    public void acquire() {
        Thread me = Thread.currentThread();
        while (TestAndSet(val) == LOCKED) {
            waitQueue.waitForAccess(me); // In queue
            me.sleep(); // Put self to sleep
        } // Got the lock
    }
    public void release() {
        Thread next = waitQueue.nextThread();
        val = UNLOCKED;
        if (next != null)
            next.ready(); // Wake up a waiting thread
    }
}
```

Sorry...

- ✓ Accessing the wait queue is a critical section
  - Need to add mutual exclusion
- ✓ Need extra lock check in acquire
  - Thread may find the lock busy
  - Another thread may release the lock but before the first thread enqueues itself
- ✓ This can get ugly!

The semaphores

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 thread
    else
        s = s + 1;
}
```

```
//initialize
mutex = 1;

down(&mutex)

// critical section
up(&mutex)
```

Binary semaphore

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

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 known as the Bounded-Buffer Problem.

Producer-consumer example

```
int 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 item from buffer
        up(&mutex);          // end critical section
        up(&empty);          // one more empty slot
        consume_item(item); // consume it
}}
```

Readers-writers example

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

Readers-writers example

```
sem mutex=1;          // CS used only by the reader
sem canwrite=1;       // CS for N readers vs. 1 writer
int readcount = 0; // number of concurrent readers

writer() {
    for (;;) {
        down(&canwrite); // block if we cannot write
        // write data
        up(&canwrite);   // end critical section
    }
}
```

Readers-writers example

```
sem mutex=1;          // CS used only by the reader
sem canwrite=1;       // CS for N readers vs. 1 writer
int readcount = 0; // number of concurrent readers
reader() {
    for (;;) {
        down(&mutex);
        readcount++;
        if (readcount == 1) // first reader
            down(canwrite); // sleep or disallow the
                writer from writing
        up(&mutex);
        // do the read
        down(&mutex);
        readcount--;
        if (readcount == 0)
            up(canwrite); // no more readers! Allow
                the writer access
        up(&mutex);
}}
```

Event counters

- ✓ Avoid race conditions without using mutual exclusion
- ✓ An event counter is an integer
- ✓ Three operations:

`read(E)`	return the current value of event counter `E`
`advance(E)`	increment `E` (atomically)
`await(E, v)`	wait until `E` $\geq$ `v`

Producer-consumer example

```
#define N 4 // four slots in the buffer
int in=0;   // number of items inserted into buffer
int out=0;  // number of items removed from buffer
producer() {
  int item, seq=0;
  for (;;) {
    produce_item(&item);// produce something
    seq++;              // item # of item produced
    await(out, seq-N);  // wait until there's room
    enter_item(item);   // put item in buffer
    advance(&in);        // to consumer: one more item
}}
consumer() {
  int item, seq=0;
  for (;;) {
    seq++;              // item # we want to consume
    await(in, seq);     // wait until item present
    remove_item(item);  // get the item from buffer
    advance(&out);      // to producer: item's gone
    consume_item(item); // consume it
}}
```

Producer-consumer example

```
#define N 4 // four slots in the buffer
int in=0, out=0;  // number of items inserted/removed
producer() {
  int item, seq=0;
  for (;;) {
    produce_item(&item);// produce something
    seq++;              // item # of item produced
    await(out, seq-N);  // wait until there's room
    enter_item(item);   // put item in buffer
    advance(&in);        // to consumer: one more item
}}
```

Suppose the producer runs for a while and the consumer does not:

It. 1: out=0, seq=1, await(0, 1-4): continue since $0 \geq -3 \rightarrow$ in=1

It. 2: out=0, seq=2, await(0, 2-4): continue since $0 \geq -2 \rightarrow$ in=2

It. 3: out=0, seq=3, await(0, 3-4): continue since $0 \geq -1 \rightarrow$ in=3

It. 4: out=0, seq=4, await(0, 4-4): continue since $0 \geq 0 \rightarrow$ in=4

It. 5: out=0, seq=5, await(0, 5-4): wait since $0 < 1$

Producer-consumer example

```
#define N 4 // four slots in the buffer
int in=0, out=0;  // number of items inserted/removed
producer() {
  int item, seq=0;
  for (;;) {
    produce_item(&item);// produce something
    seq++;              // item # of item produced
    await(out, seq-N);  // wait until there's room
    enter_item(item);   // put item in buffer
    advance(&in);        // to consumer: one more item
}}
```

Suppose the consumer runs first:

- ✓ Iteration 1: sequence $= 1$, await(0, 1) $\rightarrow$ sleep since $0 < 1$
- ✓ When the producer runs its first iteration, it will increment in
  - The consumer's await will wake up since it's now await(1,1) and $1 \geq 1$

Condition variables / Monitors

- ✓ Higher-level synchronization primitive
- ✓ 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

Mutual exclusion

Hardware solutions

Semaphores

IPC: Message passing

Deadlocks

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(prod, &m);          // send N empty messages
  for (;;) {
    receive(prod, &m);       // get a message with item
    take_item(&m, &item);// take item out of message
    send(prod, &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_mess(&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

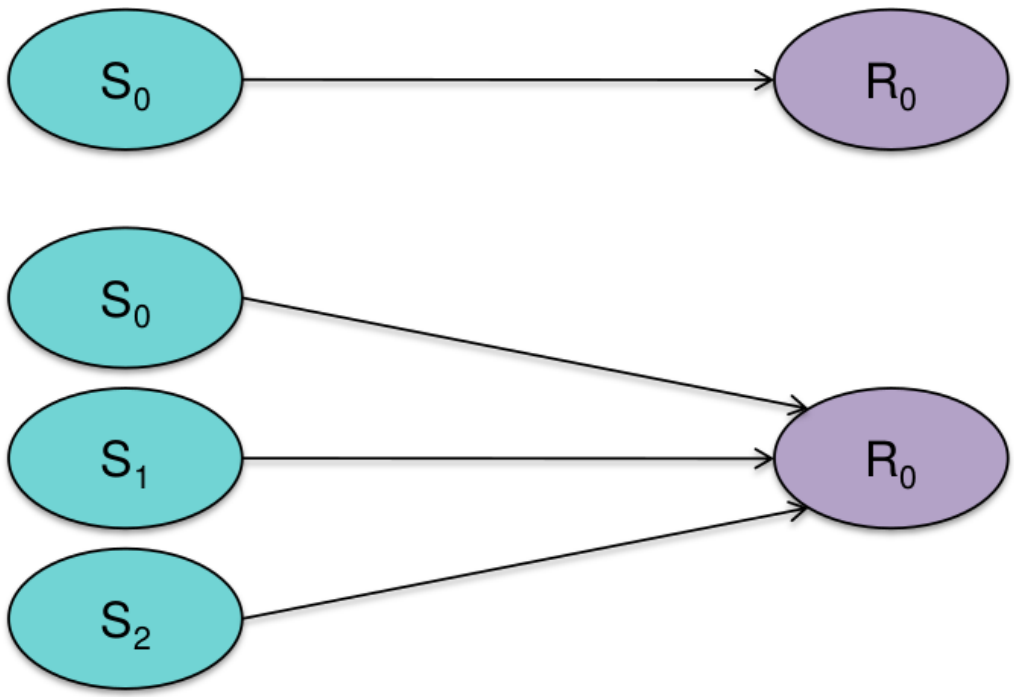
Advantages

- No need for message buffering if on same system
- Easy and efficient to implement
- Allows for tight synchronization

Disadvantage

- Forces sender and receiver to run in lockstep

Messaging: Direct Addressing



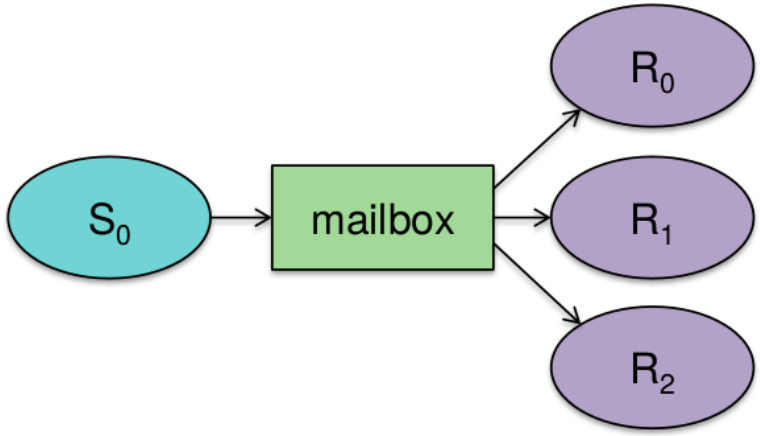
- ✓ Sending process identifies receiving process
- ✓ Receiving process can identify sending process
  - Or can receive it as a parameter

Messaging: Indirect Addressing

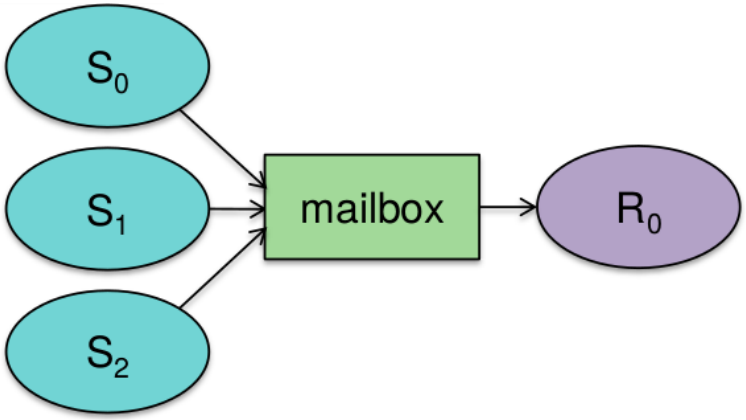
- ✓ Messages sent to an intermediary data structure of FIFO queues
- ✓ Each queue is a **mailbox**
- ✓ Simplifies multiple readers

Mailboxes

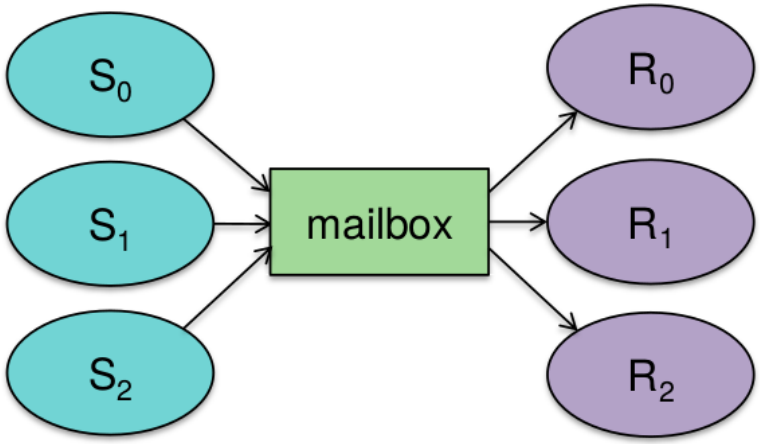
1 sender, 1 reader



M senders, 1 reader



1 sender, N readers



M senders, N readers

Mutual exclusion

Hardware solutions

Semaphores

IPC: Message passing

Deadlocks

Other common IPC mechanisms

- ✓ Shared files
  - File locking allows concurrent access control
  - Mandatory or advisory
- ✓ Signal
  - A simple poke
- ✓ Pipe
  - Two-way data stream using file descriptors (but not names)
  - Need a common parent or threads in the same process
- ✓ Named pipe (FIFO file)
  - Like a pipe but opened like a file
- ✓ Shared memory

Conditions for deadlock

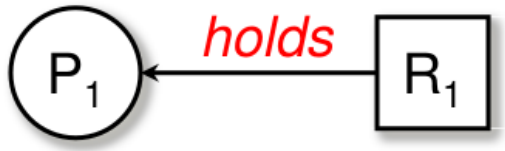
Four conditions must hold:

1. **Mutual exclusion**
   Only one thread can access a critical section (resource) at a time
2. **Hold and wait**
   A thread holds a resource but waits for another resource
3. **Non-preemption of resources**
   Resources can only be released voluntarily
4. **Circular wait**
   There is a cyclic dependency of threads waiting on resources

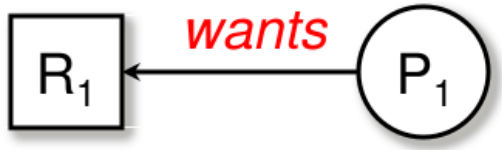
Deadlock

Resource allocation

- ✓ Resource R_1 is allocated to process P_1: **assignment edge**

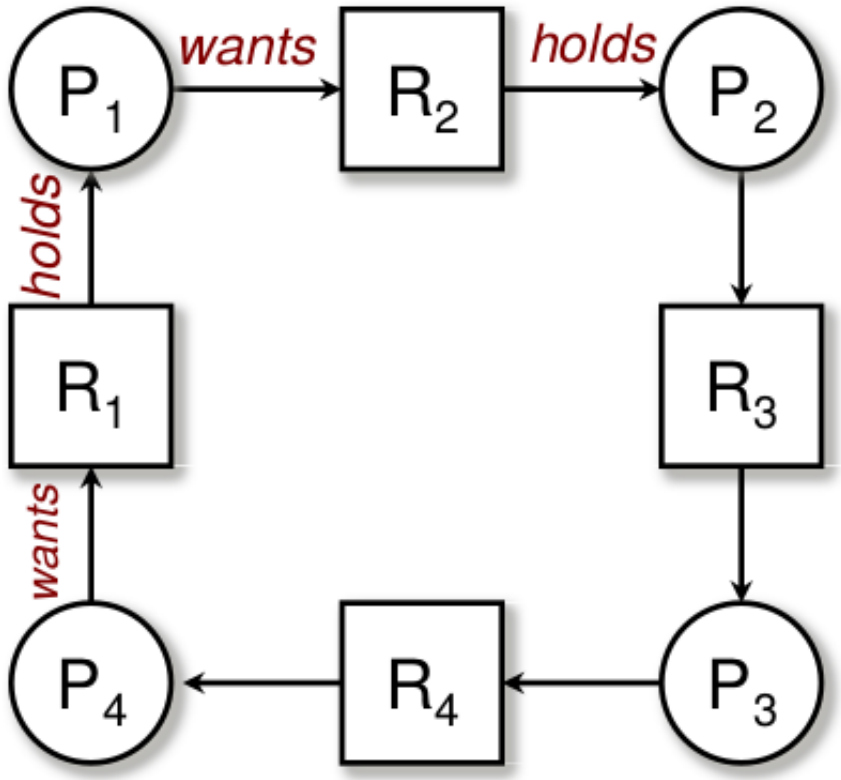


- ✓ Resource R_1 is requested by process P_1: **request edge**



`Deadlock` is present when the graph has `cycles`

Deadlock example



Circular dependency among four processes and four resources leads to deadlock.

Dealing with deadlock

Deadlock prevention

Ensure that at least one of 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
- ✓ But knowing which resources will be used (and when) is hard! (impossible, really)

Deadlock detection

Detect when a deadlock occurs and then deal with it

Ignore the problem

Let the user deal with it (most common approach)