MULTITHREADING

CSE13S
Prof. Darrell Long

PROCESSES & THREADS

What's a process?

- Code, data, and stack
  - Usually has its own address space
- Program state
  - CPU registers
  - Program counter (current location in the code)
  - Stack pointer

Only one process can run on a single CPU core at any given time.

- Multi-core CPUs can support multiple processes and threads.

A process contains one or more *threads*.

- Threads within a process run concurrently.
- Threads share memory with each other.
  - They live in the same address space!

Process Items	**Thread Items**
• Address space • Open files • Child processes • Signals & handlers • Accounting info • Global variables	• Program counter • Register values • Stack • Stack pointer • Local variables

ADDRESS SPACE

Programs execute code.

- Each instruction has an address.

Programs access data.

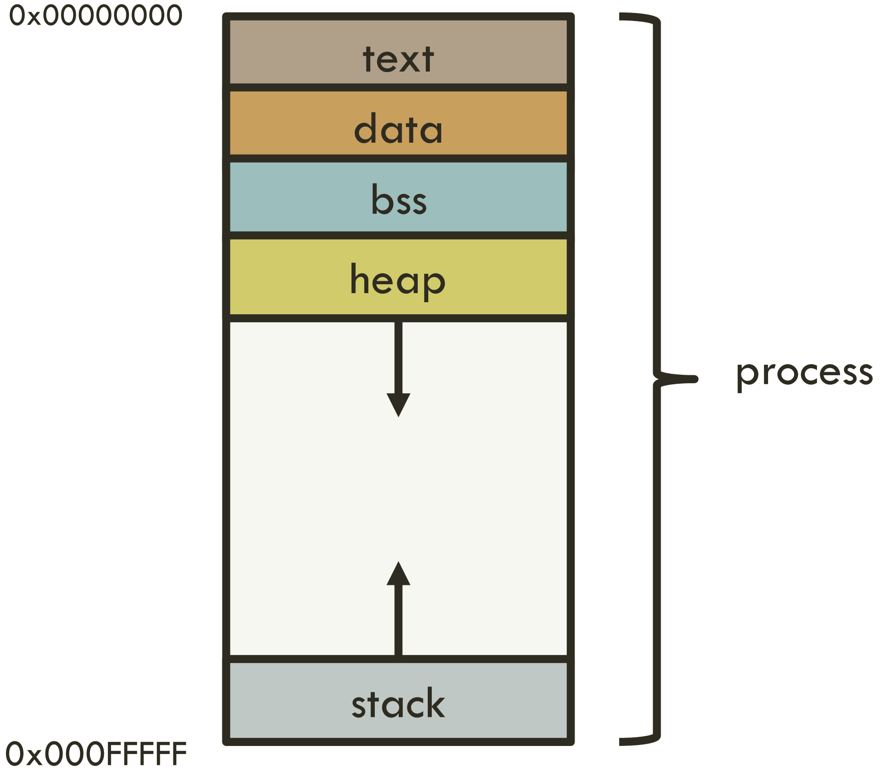
- Each byte of data also has an address.

We would like to think that our program is the only program executing on the computer.

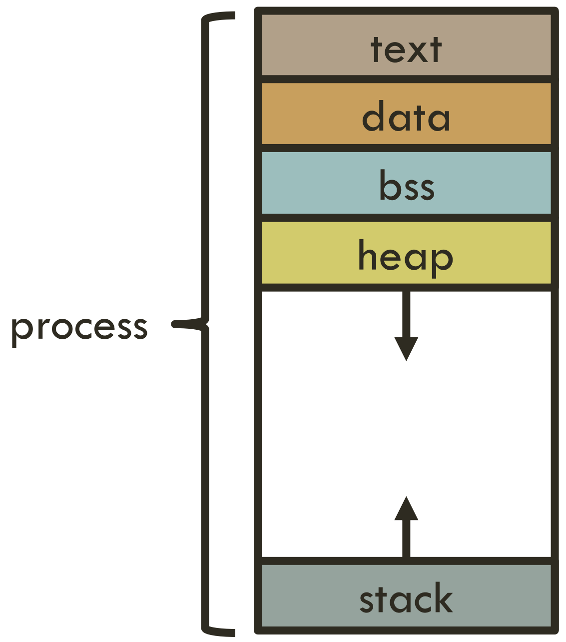
- Which prompts the need for a level of abstraction by the OS.

An address space is the region of a computer's memory where a program executes.

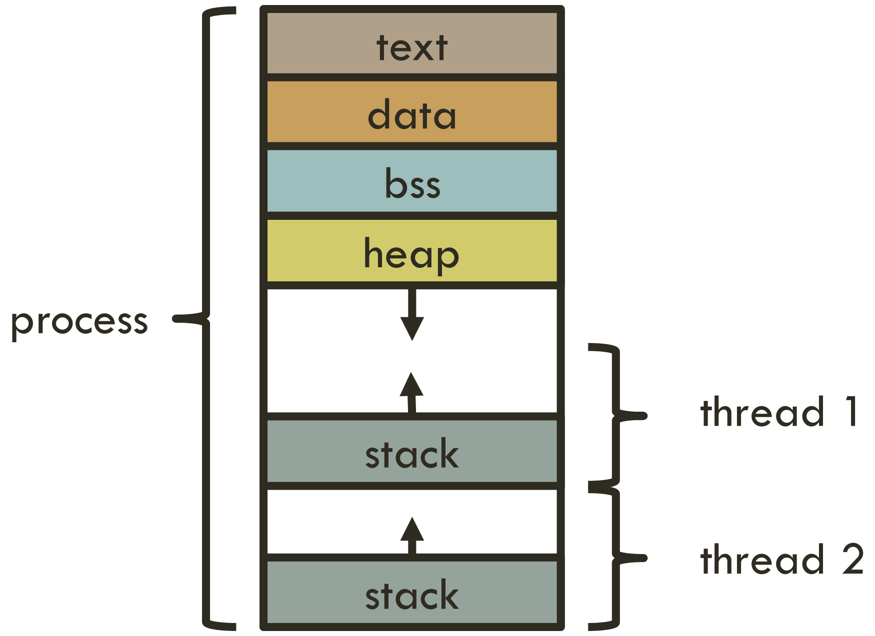
- Ideally, it is protected from other programs accessing it.



SINGLE VS MULTI-THREADED ADDRESS SPACES



Single-threaded



Multi-threaded

Threads do not replace processes, they *enhance* them.

WHY USE THREADS?

Faster to create or destroy than processes.
- No separate address space.

Allow a single application to do many things at once (like a web server).
- Can keep working during IO wait.
  - Each threads gets to issue its own IOs.
  - More IOs can be outstanding/pending.

Context switching:
- Expensive between *processes*.
- Less expensive between *threads*.

Sharing memory:
- Processes don't inherently share memory (each process has its own address space).
  - Need to use inter-process communication (IPC) to manage shared data.
- Threads share memory, making it easier to share data.

BASIC POSIX THREAD API

The POSIX thread library for C is a standard API for multithreading.

- Included under <pthread.h>.

For basic multithreading, we will only really need two of the functions:

1. `pthread_create()`
2. `pthread_join()`

More into the details and ideas of these functions in a bit.

Thread call	Description
`pthread_create()`	Create a new thread.
`pthread_join()`	Wait for a thread to complete.

CREATING A THREAD

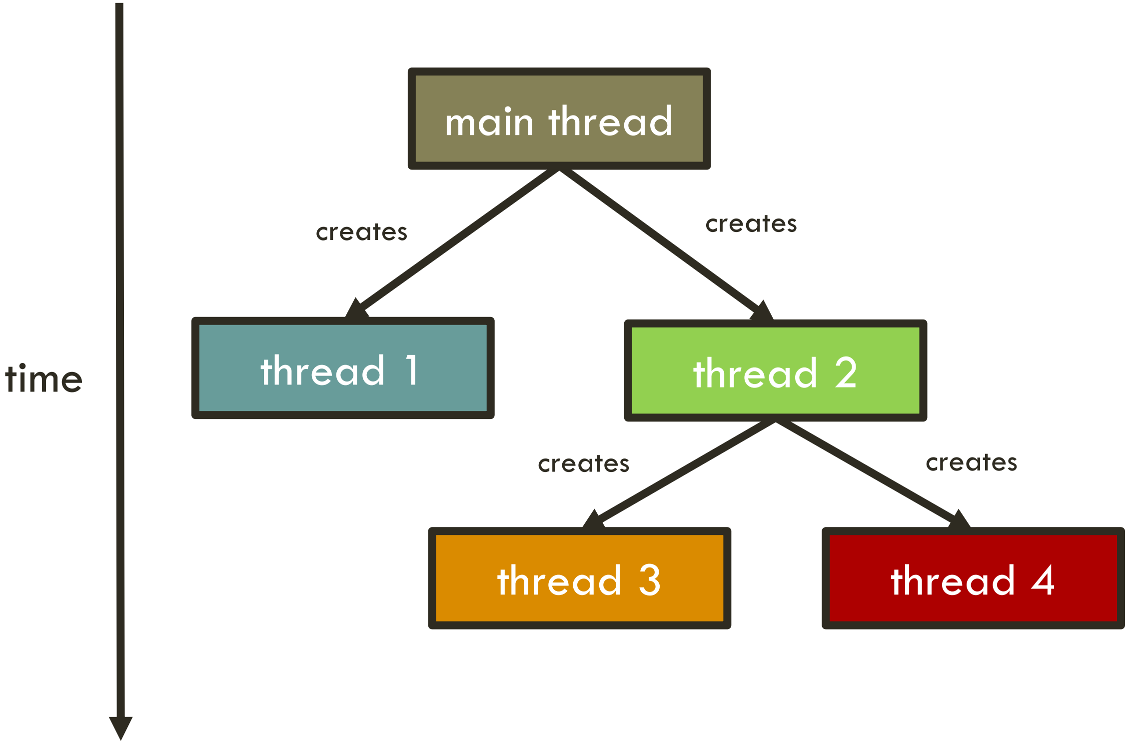
A process is started with one thread of execution: the main thread.

The main thread can create threads, and those threads may create other threads as well.

- Threads are siblings, no sense of hierarchy between them.
- The main thread, though, should end last.
  - If it ends, the process ends.

When a thread is created, it gets:

- A thread ID
- A program counter
- A stack and a stack pointer
- Register values
- Some priority
  - Affects the thread execution order.
  - The higher the priority, the earlier it gets scheduled.



CONTEXT SWITCHING

Halting (or pausing) the execution of a certain thread by the OS for some reason in order to execute some other thread.

Involves saving the state of the process or thread into a private memory region, so that the state can be reloaded and execution resumed later on.

Analogy:

- You are reading a book, and a friend comes along and requests to read the book as well.
- You save your position in the book, and hand off the book to your friend.
- You can also request the book back from your friend.
- The friend will save their position in the book, and hand it back to you.
- You then continue reading from the position that you saved from before.

The more that needs to be saved, the more expensive the switch.

Excessive context switching should be avoided.

GREETINGS, I'M A THREAD

Here we'll define a simple program that spawns threads and makes each one print out a message.

- We create 3 threads.
- Each will print out their thread ID, the language assigned to them, and a message in that language.

Upon creation, each thread is given a function, or a start routine, to execute.

- This function is allowed at most one argument.

If you need to pass multiple arguments to the start routine, you must first define a struct that contains those arguments.

- In this case, we bundle the language and message into one argument struct.

But what does the loop at the end do?

- What might happen if the main thread finishes execution but the other threads haven't?
- We solve this by *joining threads*.

```c
#include <inttypes.h>
#include <pthread.h>
#include <stdio.h>

#define NUM_THREADS 3

typedef struct Args {
    char *language;
    char *message;
} Args;

void *speak(Args *a) {
    uint64_t tid = 0;
    pthread_threadid_np(pthread_self(), &tid);
    printf("I'm thread %" PRIu64 ". I speak %s. %s.\n", tid, a->language, a->message);
    return NULL;
}

int main(void) {
    pthread_t threads[NUM_THREADS];

    Args a = { "English", "Hello" };
    pthread_create(&threads[0], NULL, (void *)speak, &a);

    Args b = { "French", "Bonjour" };
    pthread_create(&threads[1], NULL, (void *)speak, &b);

    Args c = { "Italian", "Ciao" };
    pthread_create(&threads[2], NULL, (void *)speak, &c);

    for (int i = 0; i < NUM_THREADS; i += 1) {
        pthread_join(threads[i], NULL);
    }

    return 0;
}
```

JOINING A THREAD

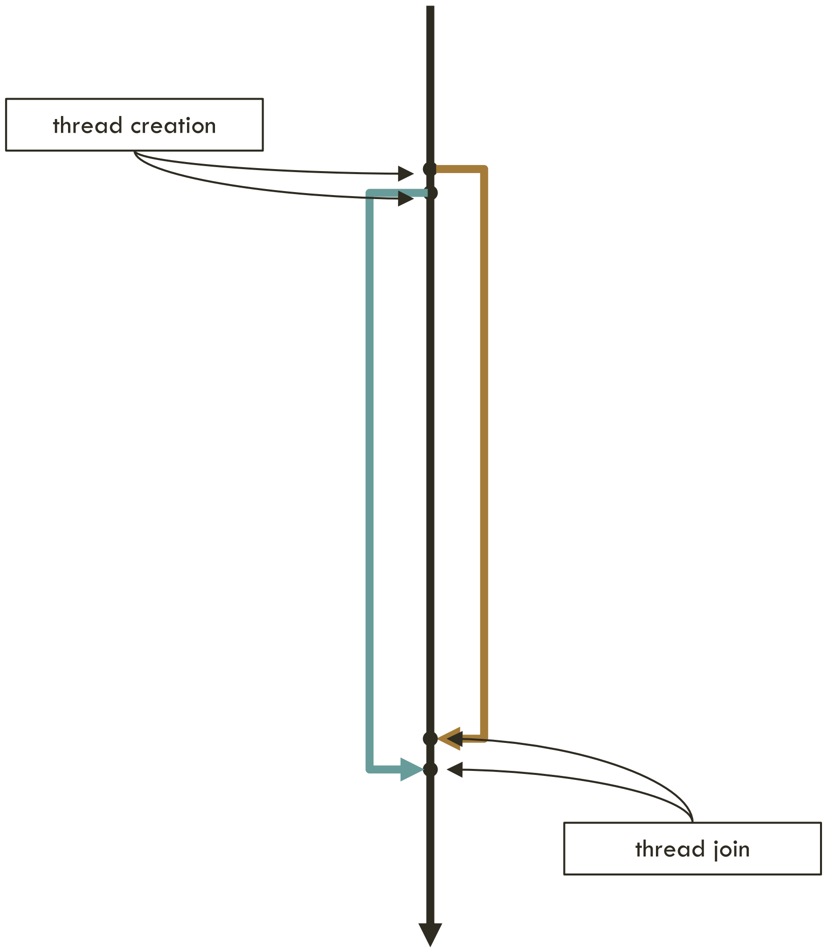
To join a thread means waiting for it to finish.

Basic idea:

- Let threads perform some job in parallel.
- Wait for all threads tasked with the job to finish.
- Once all threads finish execution, the job is completed.
  - This is where joining comes in!
  - Weird things can happen if you continue your program without verifying that all threads are done.

There are cases in which you do not want to join threads.

- You want the main thread to continue execution while the other threads do their assigned tasks.
- Commonly seen in web servers.



FIRST TRY AT MULTITHREADING

We'll write a simple multithreaded program.

- Each process starts out with a main thread.
- This main thread will then spawn 4 additional threads.
- Each of these threads is tasked with incrementing a global counter 10000 times.
- The threads are joined, and the final counter value is printed.

We expect the output of this program to be 40000.

- We run the program 10 times in succession, but the output is never correct.
  - Why? Because of *race conditions*.

```c
#include <stdio.h>
#include <pthread.h>

#define NUM_THREADS 4

int counter = 0;

void *increase() {
    for (int i = 0; i < 10000; i += 1) {
        counter += 1;
    }
    return NULL;
}

int main(void) {
    pthread_t threads[NUM_THREADS];
    for (int i = 0; i < NUM_THREADS; i += 1) {
        pthread_create(&threads[i], NULL, (void *)increase, NULL);
    }

    for (int i = 0; i < NUM_THREADS; i += 1) {
        pthread_join(threads[i], NULL);
    }

    printf("counter = %d\n", counter);
    return 0;
}
```

```
$ for i in {0..10}
for> do
for> ./thread_unsafe
for> done
counter = 29157
counter = 18830
counter = 20291
counter = 17766
counter = 16679
counter = 19092
counter = 22780
counter = 23898
counter = 25092
counter = 18460
counter = 19462
```

RACE CONDITIONS

Threads share memory.

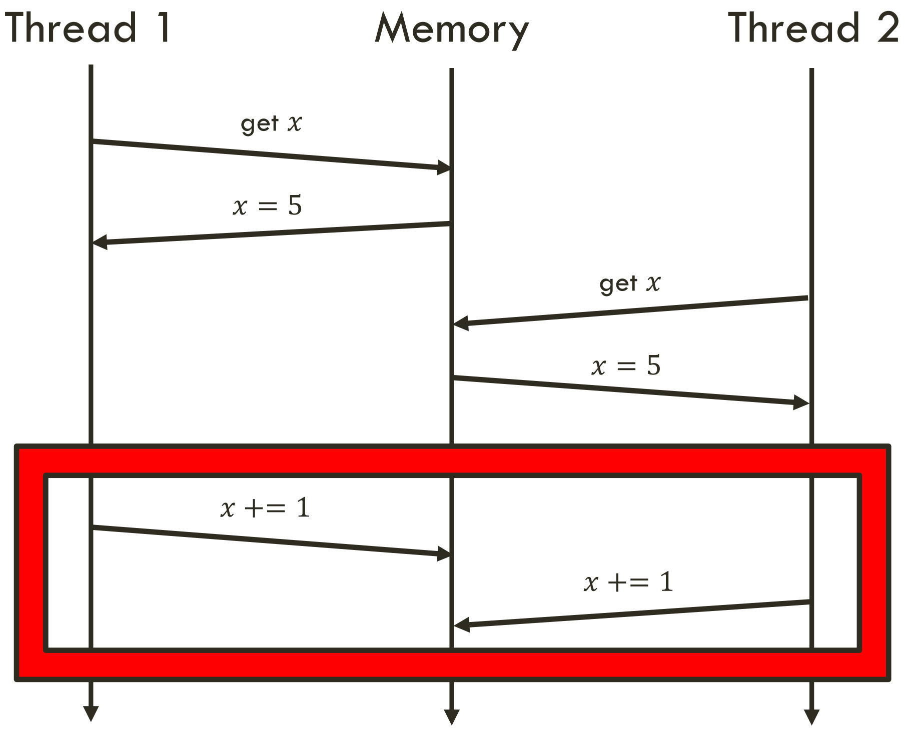
- Different threads may read/write the same memory.
- Thus, threads share resources.

Problem: no guarantee that read followed by write is atomic.

- Which means the ordering of events matters!
- Also results in erroneous results.

Example on right shows the problem with the first attempt at multithreading.

- Why some increments of the counter didn't seem to take effect.



$x = 6$
Incremented once, not twice!

CRITICAL REGIONS

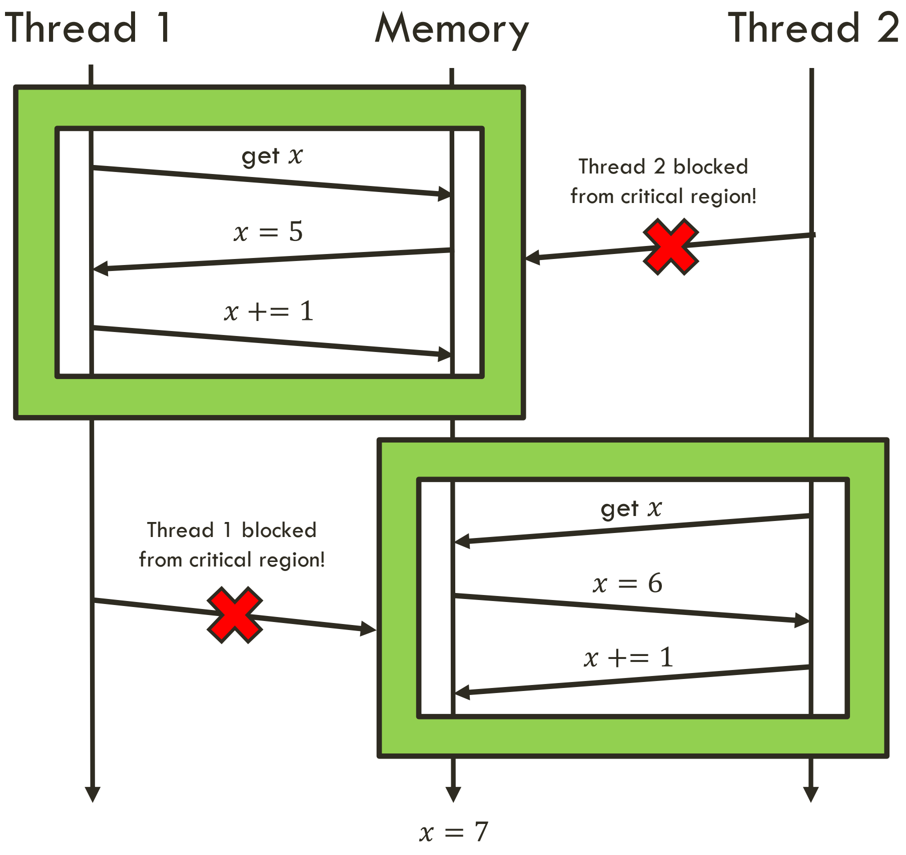
Use critical regions to provide *mutual exclusion* to help fix race conditions.

Four conditions must hold to provide mutual exclusion:

1. No two processes may simultaneously be in critical region.
2. No assumptions may be made about speed.
3. No process running outside its critical region may block another process.
4. A process may not wait forever to enter the critical region.

A process in this sense doesn't need to be specifically an OS process.

- Could be a thread, or a database connection.



LOCKING FOR MUTUAL EXCLUSION

The locking out of threads from critical regions is done using a *mutex*.

- With POSIX threads, this is type `pthread_mutex_t`.

Two main operations:

1. `pthread_mutex_lock()`
2. `pthread_mutex_unlock()`

Idea:

- Have a mutex for a critical region.
- Before entering the critical region, a thread must lock the mutex first.
  - If the mutex has already been locked, the thread is blocked until it has been unlocked.
- The thread unlocks the mutex after leaving the critical region.

SECOND TRY AT MULTITHREADING

We go back and add locks around the incrementing loop.

- That's the critical region.
- Now, only one thread and increment the counter at a time.

Running this revised code yields the correct result.

- 4 threads incrementing a counter 10000 times each should yield 40000.

```c
#include <stdio.h>
#include <pthread.h>

#define NUM_THREADS 4

int counter = 0;

void *increase() {
    static pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

    pthread_mutex_lock(&lock);
    for (int i = 0; i < 10000; i += 1) {
        counter += 1;
    }
    pthread_mutex_unlock(&lock);

    return NULL;
}

int main(void) {
    pthread_t threads[NUM_THREADS];
    for (int i = 0; i < NUM_THREADS; i += 1) {
        pthread_create(&threads[i], NULL, (void *)increase, NULL);
    }

    for (int i = 0; i < NUM_THREADS; i += 1) {
        pthread_join(threads[i], NULL);
    }

    printf("counter = %d\n", counter);
    return 0;
}
```

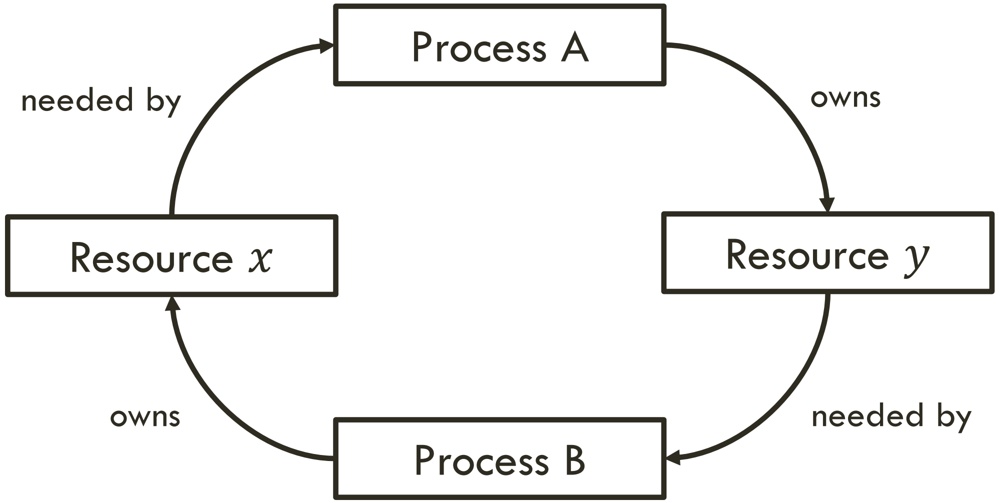
```
$ for i in {0..10}
for> do
for> ./thread_safe
for> done
counter = 40000
counter = 40000
counter = 40000
counter = 40000
counter = 40000
counter = 40000
counter = 40000
counter = 40000
counter = 40000
counter = 40000
counter = 40000
```

DEADLOCKS

Formal definition: “A set of processes is *deadlocked* if each process in the set is waiting for an event that only another process in the set can cause”.

Basically, a cyclic dependency on some resource.

- Process A needs x from process B in order to run.
- Process B needs y from process A in order to run.
- Neither process is willing to give up granted resources.



RESOURCES

Resource: something a process uses (usually limited).

Examples of computer resources:

- Printers
- Semaphores/locks
- Memory (with threads, this is usually a *shared* variable)
- Tables (in a database)

Processes need access to resources in reasonable order.

Two types of resources:

- Preemptable resources: can be taken away from a process with no ill effects.
- Non-preemptable resources: causes ill effects if taken away from a process.

USING RESOURCES

To use a resource, a process must:

1. Request the resource
2. Do something with the resource
3. Release the resource

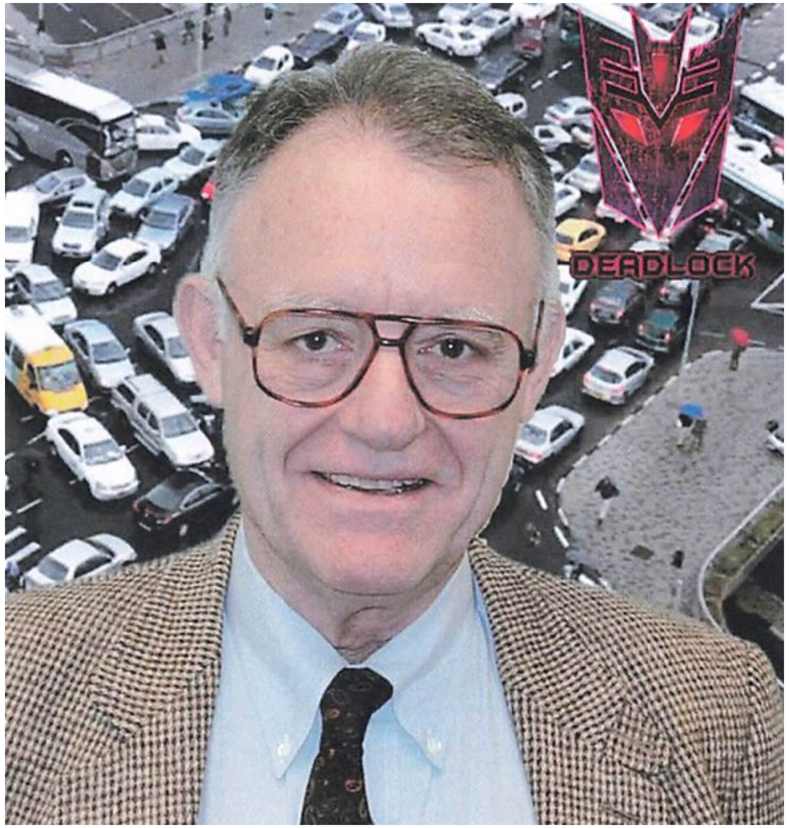
Can’t use the resource if the initial request is denied.

- The requesting process has some options:
  - Block and wait for the resource.
  - Continue (if possible) without it, potentially using an alternate resource.
  - Fail with an error code.
- Some of these options may be able to prevent deadlocks from occurring.

CONDITIONS FOR DEADLOCK

There are 4 conditions necessary for deadlock:

1. **Mutual exclusion**
   - Each resource is assigned to at most one process.
2. **Hold & wait**
   - A process holding resources can request more resources.
3. **No preemption**
   - Resources granted previous cannot be forcibly taken away.
4. **Circular wait**
   - There must be a circular chain of 2 or more processes where each is waiting for a resource held by the next member of the chain.



RESOURCE ALLOCATION GRAPHS

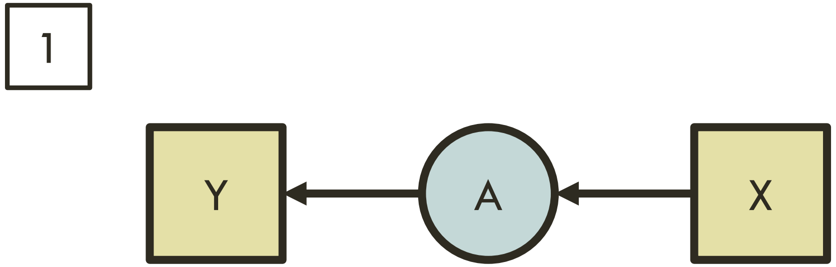
Resource allocation can be modeled using *directed* graphs.

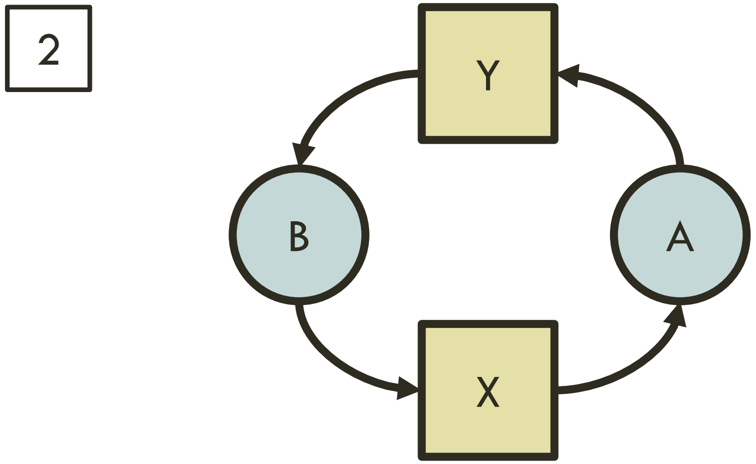
Example 1:

- Resource X is assigned to process A.
- Process A is requesting/waiting for resource Y.

Example 2:

- Process A holds X and is waiting for Y.
- Process B holds Y and is waiting for X.
- A and B are in deadlock!



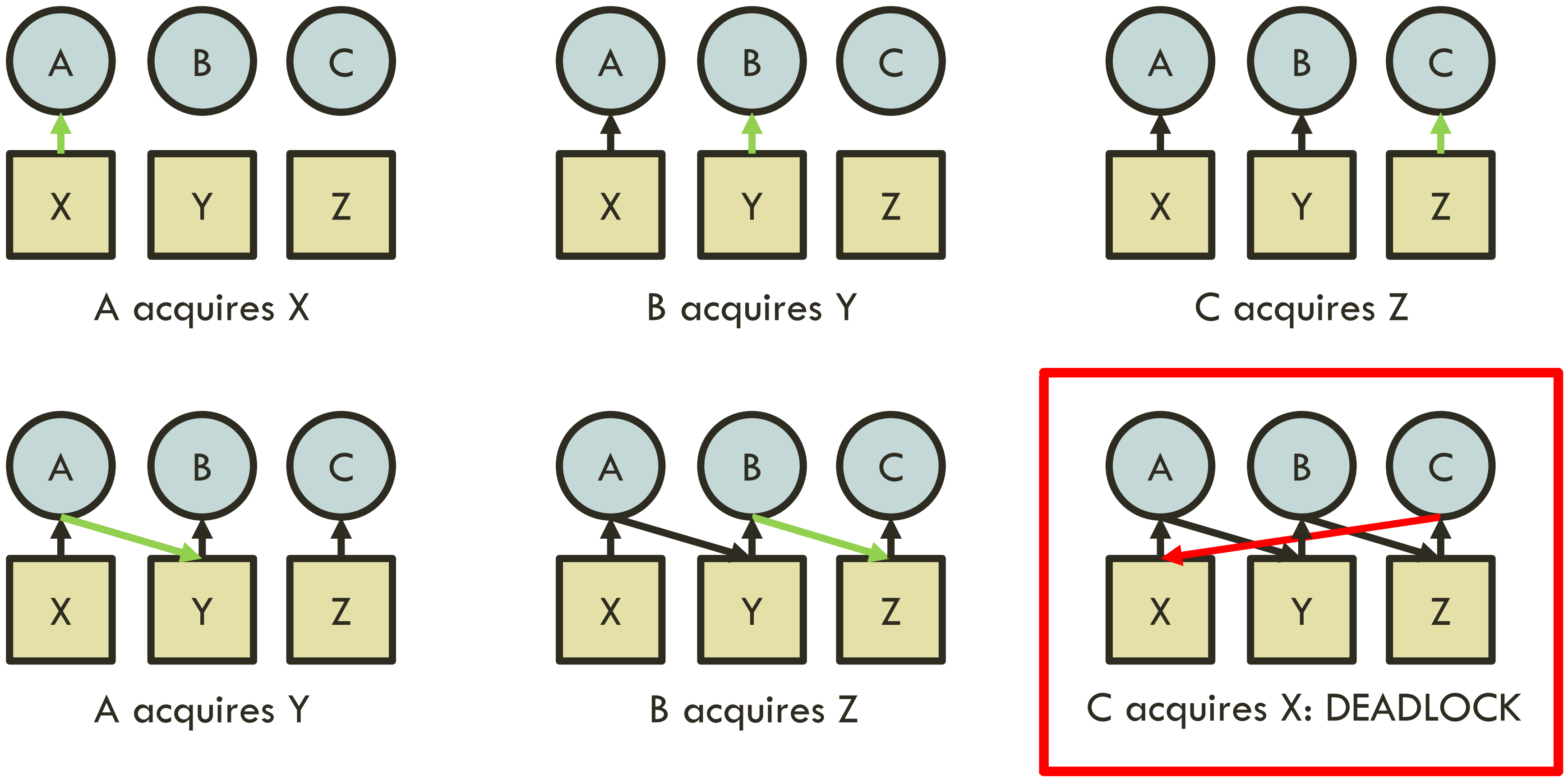


GETTING INTO DEADLOCK

Process A:
- Acquire X
- Acquire Y
- Release X
- Release Y

Process B:
- Acquire Y
- Acquire Z
- Release Y
- Release Z

Process C:
- Acquire Z
- Acquire X
- Release Z
- Release X



THE OSTRICH ALGORITHM

Simply pretend that there's no problem.

This is reasonable if:

- Deadlocks occur very rarely.
- Preventing deadlocks is costly.

Unix and Windows take this approach.

- Resources (memory, CPU, disk space) are plentiful.
- Deadlocks over such resources rarely occur.
- If a deadlock does occur, it is typically handled by rebooting.

This is a trade off between *convenience* and *correctness*.

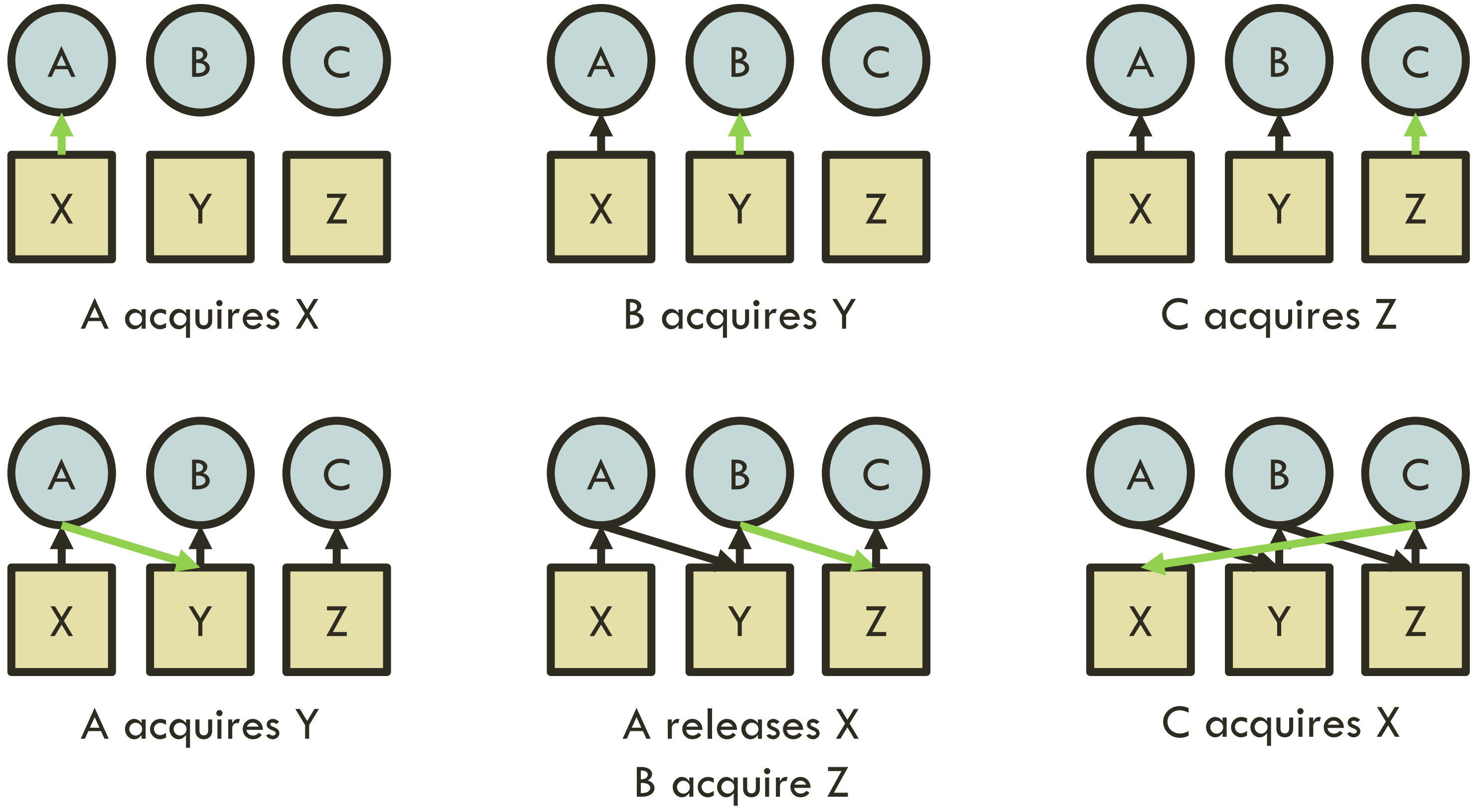
AVOIDING DEADLOCK

Many situations may result in deadlock, but they do not necessarily have to.

- In previous example, A could have released X before C requested it, avoiding the deadlock.
- Can we always get out of it this way?

We want to either:

- Detect deadlocks and reverse them, or
- Stop deadlocks from occurring in the first place.



DEADLOCK DETECTION WITH GRAPHS

Process holdings and requests in resource allocation graph.

- If graph contains a cycle, then there exists a deadlock.

Algorithm:

- DFS at each node.
- Mark arcs as they're traversed.
- Build set of visited nodes.
- If a node to be visited is already in the set of visited nodes, then a cycle has been found.

```
// Deadlock detection pseudocode.

for node n in graph G {
  visited = empty set
  unmark all arcs in G
  traverse(n, visited, G)
}

func traverse(n, visited, G) {
  if n in visited {
    report deadlock
  }
  add n to visited
  for unmarked arc n to m in G {
    mark the arc
     traverse(m, visited, g)
  }
  remove n from visited
}
```

RECOVERING FROM DEADLOCK

Recovery through preemption

- Take a resource from some other process.
- This depends on the nature of the resource and process.

Recovery through rollback

- Checkpoint a process periodically.
- Use saved state to restart the process if it's in deadlock.
- This may be a problem if the process affects lots of "external" things.

Recovery through killing processes

- Crude but simple: kill one of the processes in the deadlock cycle.
- Allows other processes to get their resources.
- Try to choose a process that can be rerun from the start.
  - Pick one that hasn't run too far already.

PREVENTING DEADLOCK

It is sometimes possible to completely prevent deadlock.

- Simply ensure that at least one of the 4 necessary conditions for deadlock never occurs.

Try to attack:

1. Mutual exclusion
2. Hold & wait
3. No preemption
4. Circular wait

ATTACKING MUTUAL EXCLUSION

Some devices (such as printers) can be spooled.

- Only the printer daemon uses printer resource.
- This eliminates deadlock for the printer.

Spooling: queueing tasks together, in which a dedicated program, the spooler, dequeues as needed.

- The term spooling was probably used due to magnetic tape wound onto a spool.

Principle:

- Avoid assigning the resource when it isn't absolutely necessary.
- As few processes as possible can actually claim the resource.

Spooling cannot be used for all devices…

ATTACKING "HOLD & WAIT"

Require processes to request resources before starting.

- A process never has to wait for what it needs.

This can present issues:

- A problem may not know what resources it needs before it starts.
- This also ties up resources other processes could be using.
  - Processes will tend to be conservative and request resources they might need.

Variation: a process must give up all resources before making a new request.

- Process is then granted all prior resources as well as the new ones.
- Problem: what if the sources are claimed by another process in the meantime – how can the process save its state?

ATTACKING “NO PREEMPTION”

Attack ”no preemption” by allowing preemption.

Typically not a viable option.

Consider a process given a printer:

- Halfway through the job, take away the printer.

Could work for some resources.

- Forcibly take away memory pages, suspending the process.
- Might be possible for process to resume with no ill effects.

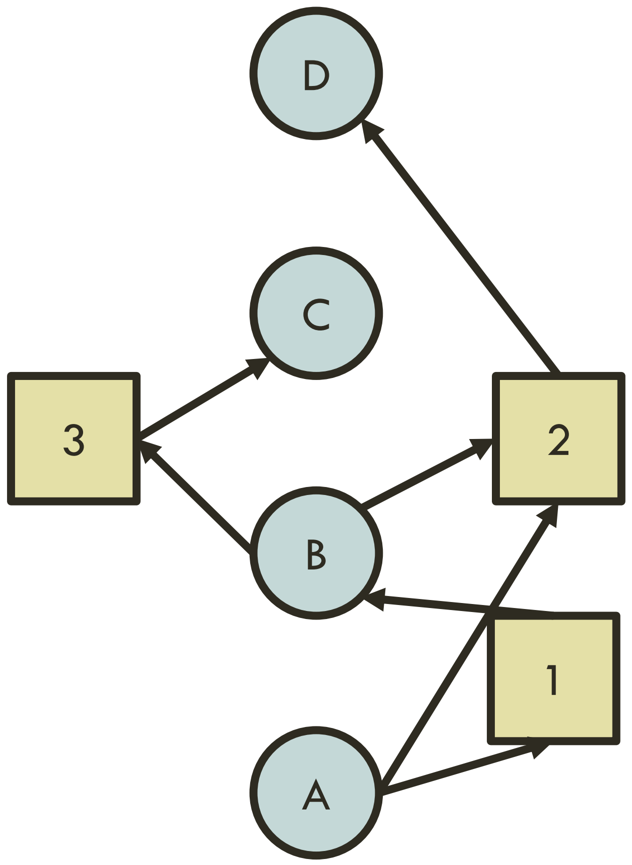
ATTACKING “CIRCULAR WAIT”

Assign an order to resources.

- Acquire the resources in numerical order.
- No need to acquire all of them at once.

Circular wait is prevented.

- A process holding resource y can’t wait for resource x if $x < y$.
- No way to complete a cycle.
  - Place processes above highest resource they hold and below any they’re requesting.
  - All arrows in the resource allocation graph will then point upwards.



DEADLOCK PREVENTION: SUMMARY

Condition	Prevented by
Mutual exclusion	Spool if possible.
Hold & wait	Request all resources before starting.
No preemption	Take resources away if there isn't a complete set.
Circular wait	Order resources numerically.

STARVATION

Assume the following algorithm to allocate a resource:

- Give the resource to the shortest job first.

Works great for multiple short jobs in a system.

- May cause long jobs to be postponed **indefinitely**.

Solution:

- Use a first-come, first-serve policy instead.

Starvation can lead to deadlock.

- Process starved for resources can already be holding resources.
- If those resource aren't used and released in a timely manner, the shortage could lead to deadlock.