Concurrency with Processes/Threads

ICS432

Concurrent and High-Performance Programming

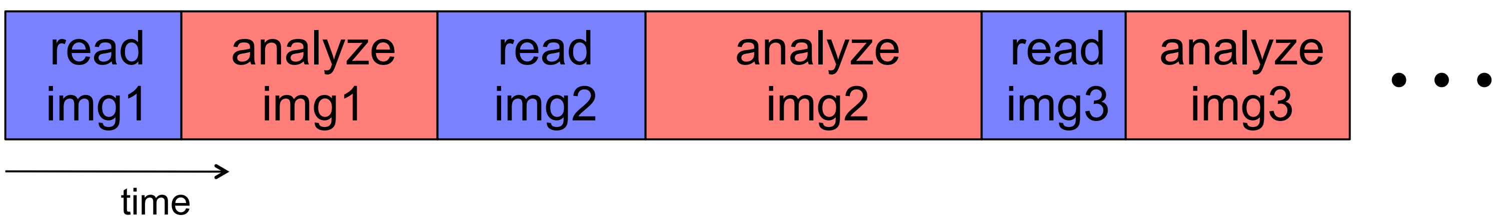
Henri Casanova (henric@hawaii.edu)

Concurrency with Tasks

- When developing a concurrent application one thinks of the application as a set of tasks
- Different tasks can do different things, or can do the same things on different data
  - One talks of “task parallelism” and “data parallelism”
- Some tasks may need to talk to each other
  - e.g., wait for each other, say “go head” to each other, wake up each other.
- Let’s take as an example a simple image analysis application…

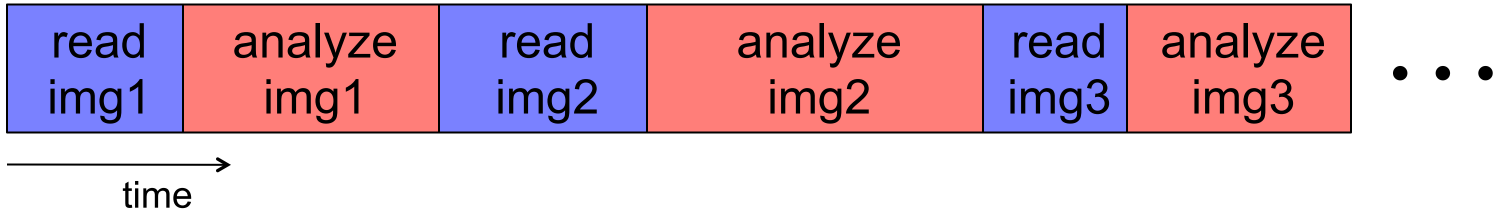
Example Image Analysis App

- Consider an application that reads image files and "analyzes" the images
  - e.g., applies an ML algorithm to detect license plates
- We have a SINGLE CORE and a SINGLE DISK
- A sequential execution would look like this:



- **Our objective:** use concurrency to improve performance
  - i.e., reduce overall execution time
- Why is the above picture "not good" performance-wise?

Improving Performance

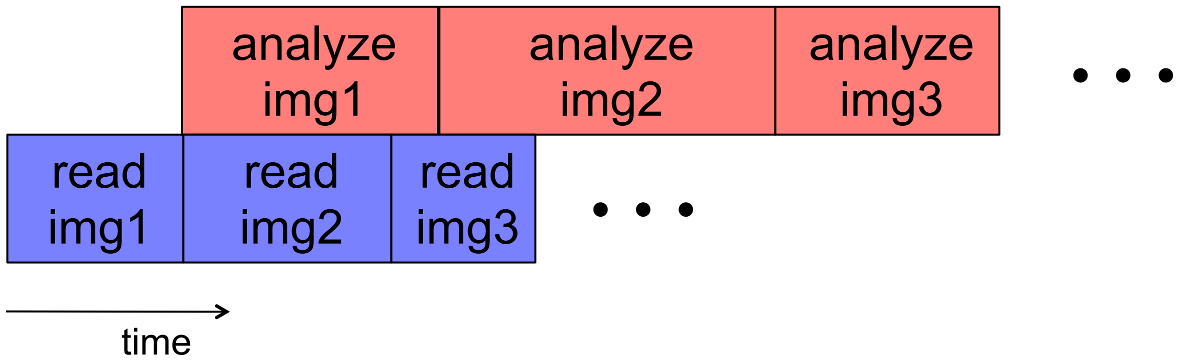


- While an image is being read, the CPU is (mostly) idle
- While an image is being processed, the disk is idle

- This is not the best use of the hardware!
- So let's now think of the application as two tasks:
  - Task #1: Image reader
  - Task #2: Image analyzer

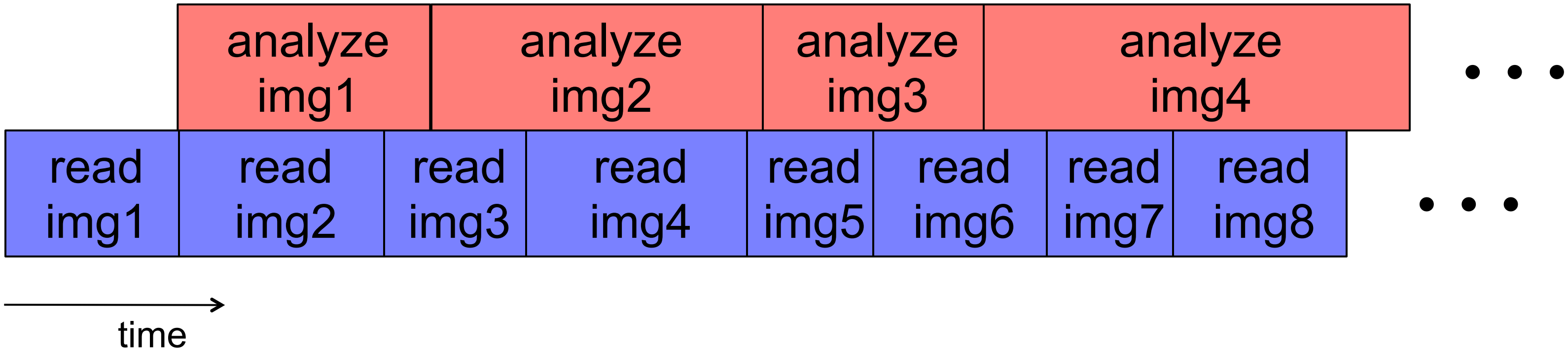
Concurrency with Two Tasks

- Now the executions (could) look like this:



- The cost of reading images is hidden after the first image has been read
- This is called **overlap of I/O and computation**
- **The tasks need to communicate:** The Reader task needs to tell the Analyzer task "I just read image #i, so you can go ahead an analyze it whenever"

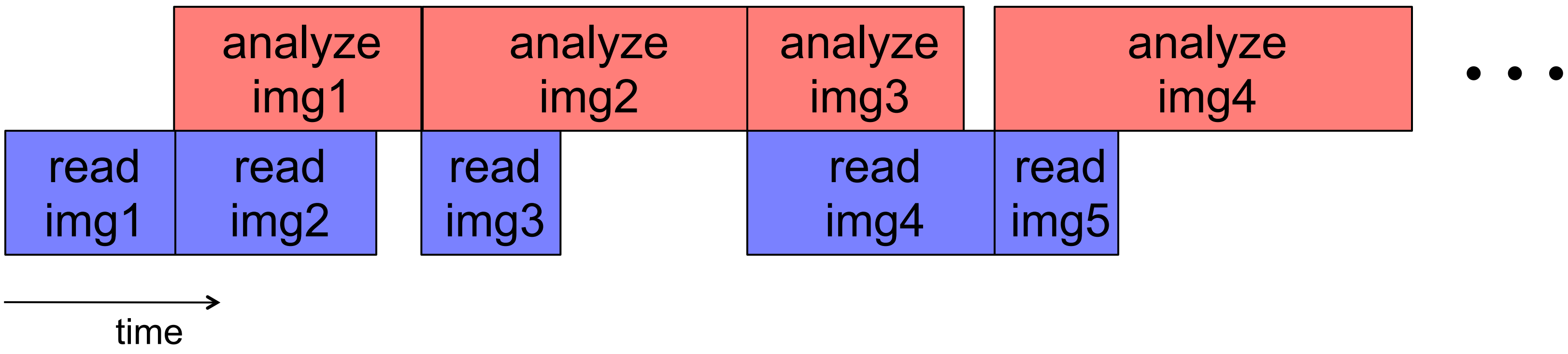
Memory Explosion?



- In this example, image reading takes less time than image analyzing
- This can lead to a memory problem: only a limited number of images can be held in memory
  - If one tries to keep too many in memory, then the application will start swapping pages to disk!
    - See your virtual memory lectures (ICS332)

Only One Image at a Time

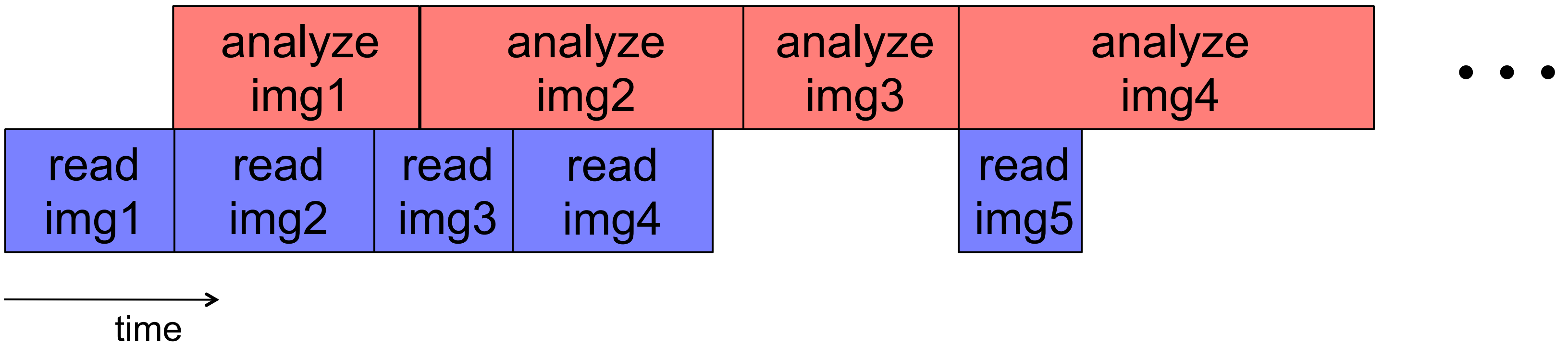
- Solution #1: Read only one image ahead of time
- Requires some synchronization between the two tasks (they need to “talk”, see later…)



- Problem: If we have images of different sizes, then reading image #i+1 may take longer than analyzing image #i
  - i=3 above leads to idle time

Only N Images at a Time

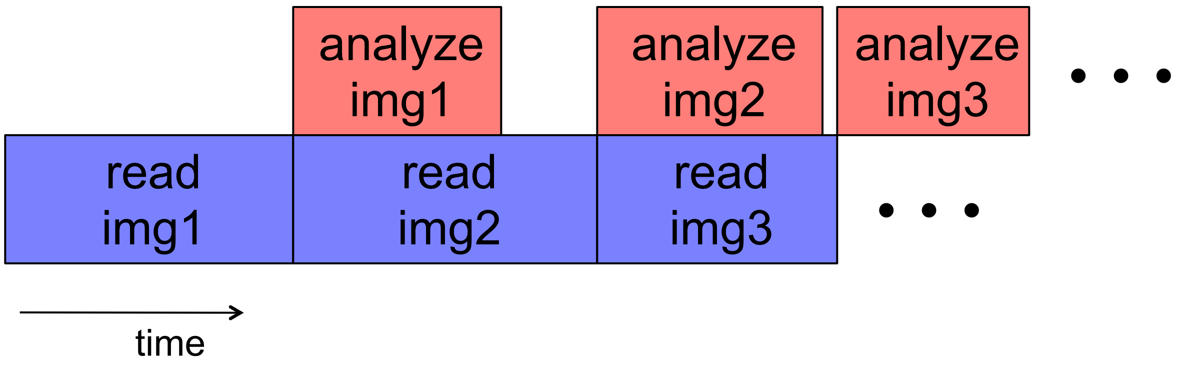
- Solution #2: Read only N images ahead of time
  - Making sure that N images always fit in memory



- In the above example, N = 3 (let's check)
- If images are very different, it could be difficult to determine the smallest N
  - Best bet: just keep at most X MBytes of image data in memory

I/O-intensive?

- What if analyzing takes less time than reading?



- The cost of analyzing images is hidden after the first one
- Good news: One doesn't have to know which operation takes less time ahead of time
  - Difficult to know: depends on the computer, to the analysis program, perhaps even on the image
- **Lesson**: Just create your tasks and make sure memory doesn't become a problem

This is a "Pipeline"

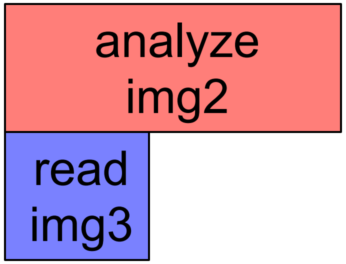
- The previous application is called a 2-stage pipeline
  - You have a sequence of operations to do
  - Each operation can be done in two stages
  - While operation #1 is in stage #2, operation #2 is in stage #1
- Typical real-life example: washer and dryer
  - While load #2 is in the dryer, load #3 is in the washer
- Similar concept here, but in software
  - Things are great if both stages take the exact same time
    - Not the case for washer/dryer (typically drying takes longer)
  - When stages don't take the same time, we can do things like hold up to N images in memory
  - Same thing with your laundry room, which has hopefully some capacity to hold some "waiting to be dried" loads
    - But If you have 1000 loads to do, you can't just keep using the washer otherwise your laundry room will overflow with wet clothes
    - Just like our RAM with images

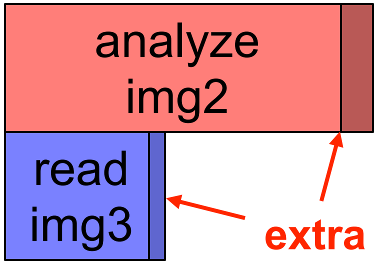
Pipeline Bottleneck

- Note that in our example, we go only as fast as the slower stage (reading or analyzing)
- If your disk can deliver 10 images per second, it doesn't matter that your core can analyze 100 images per seconds: you can only feed them 10 images per second in memory anyway
- In this case we say that the disk is the bottleneck
  - If I were to give you a faster core, that wouldn't do you any good, so cores are not the bottleneck
  - If I were to give you a faster disk, that would do you some good, so the disk is the bottleneck
- In my laundry room, the dryer is the bottleneck

No Extra Cost??

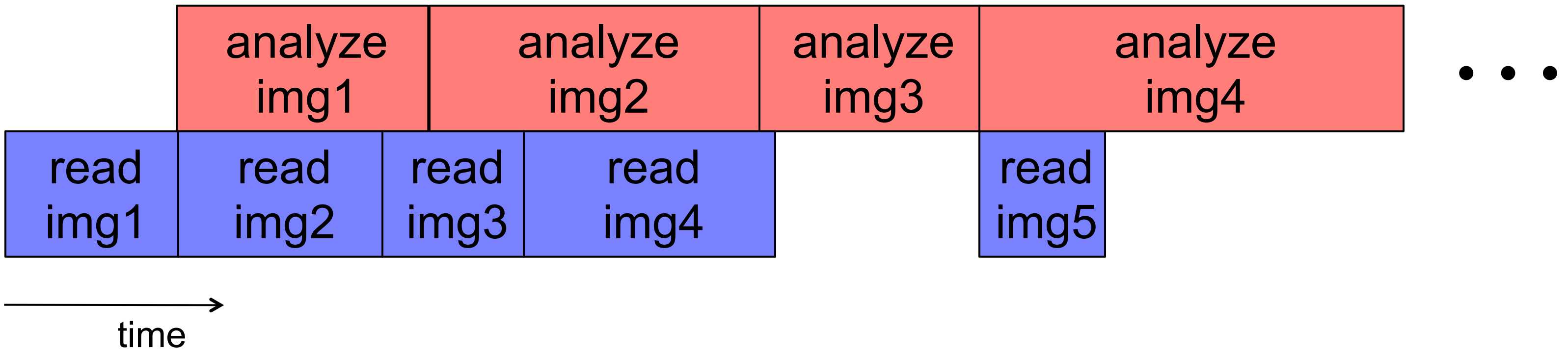
- In the laundry room, your washer and dryer can both run at full speed simultaneously
- In software it's not 100% true
- A task that reads data from disk still needs to execute some instructions on the CPU
  - But they are not very frequent because the task spends most of its time waiting for the disk (small CPU bursts, large I/O bursts)
- Furthermore, running more than one task at a time may have overhead
  - The "interleaving" of instruction requires some extra work by the CPU, O/S: context switching (we assume a single core)
- So we always lose a little bit



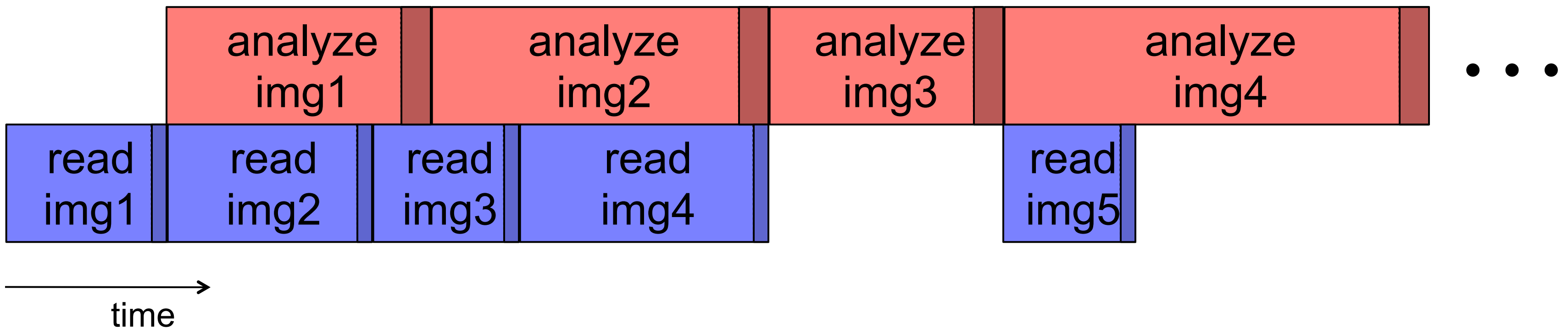


Example Concurrent App

- The ideal picture looks like this:



- The real execution time may be longer (still way better than the non-concurrent execution):



Where are we?

- We now have a pretty good idea of how one could design our image analysis application as two tasks
  - While achieving nice overlap of I/O and computation
  - And while avoiding memory explosion (even though that may cause us to have some CPU idle time depending of the images)

- Let's try to design an implementation using processes
- First, let's review what processes are…

Processes

- A process is a running program
- The OS keeps track of running programs in a data structure called *the process table*
- Each process is described as
  - A pid (process id: an integer)
  - A username (who started the process)
  - A state (running, blocked, ready, ...)
  - A program counter (points to the next instruction)
  - A stack (bookkeeping for function calls)
  - A set of file descriptors (open files, network connections,...)
  - A page table (way to track where in RAM the process' address space is located)
  - The pid of the parent process
  - ...

Processes

- All modern OSes support multiple active processes at the same time
- Each process goes through three main states
  - Ready: “I can run if the OS would let me”
  - Running: “I am running right now”
  - Blocked: “I can’t run right now because am waiting for the disk, the network, etc.”
- The OS decides which ready process runs when and for how long
  - This decision impacts the performance and the responsiveness of the computer, and OSes have been designed to do this well
  - The decision is called scheduling

Processes and Memory

- Each process has its own address space: a set of memory locations that can be read from and written to
- Virtual memory: the illusion that there is a large memory (perhaps larger than the physical memory), and that a process is the only one using it
- This illusion is always maintained, but at the cost of degraded performance at times (swapping)
- This is what makes it possible for developers to write programs and not care about the state of the computer when the program will be run
    - I write a program assuming a large address space and I don't care what other programs will be running when my program is running

(UNIX) Process Creation?

- Each time you invoke a command in a Shell (which is itself a process), you create a new process
- Or more appropriately, the Shell creates a new process on your behalf
- So somewhere in the code of the Shell program, there is a place where processes are created
- Processes are created using the fork system call, which can be called from C/C++

The Fork() System Call

- The fork system call creates a copy of a the process that calls it
  - In fact, fork calls “clone”, which is the real syscall
  - In particular the memory is copied
- After the call, both processes are free to continue along following different execution paths in the program
- fork() returns an integer
  - It returns the PID of the new process to the “parent” process
  - It returns 0 to the “child” process
- Let’s see who remembers ICS332 stuff

The Fork() System Call

- What does this program print?

```
int count = 0;
if (fork() != 0) {
        while (count < 10) {
                count++;
                sleep(1);
        }
} else {
        sleep(5);
        printf("%d\n",count);
}
```

- Show of hands: 0, 4, 5, 6, 10, or something else?

The Fork() System Call

- What does this program print?

```
int count = 0;
if (fork() != 0) {
      while (count < 10) {
            count++;
            sleep(1);
      }
} else {
      sleep(5);
      printf("%d\n",count);
}
```

- Show of hands: 0, 4, 5, 6, 10, or something else?
- Answer: 0

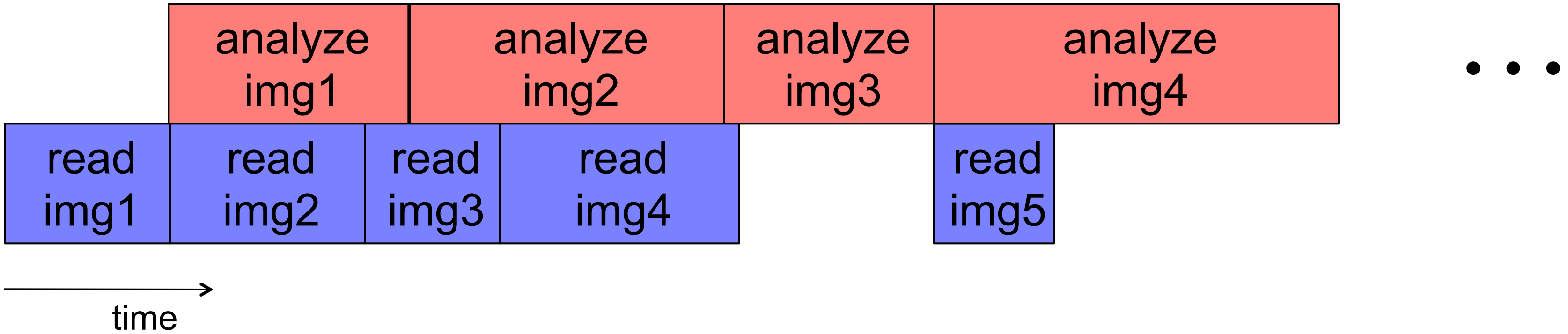
The Fork() System Call

- The two processes run on their own
- The OS is in charge of deciding when they run
  - Typically in some round-robin fashion
- The two processes have distinct address spaces
  - In our example, variables are **not** shared between the processes but each process has its own copy of each variable
  - It doesn’t matter than the parent updates its `count` variable, the child doesn’t have access to the parent’s memory space anyway
  - This is why the answer was “0”

Processes can Communicate

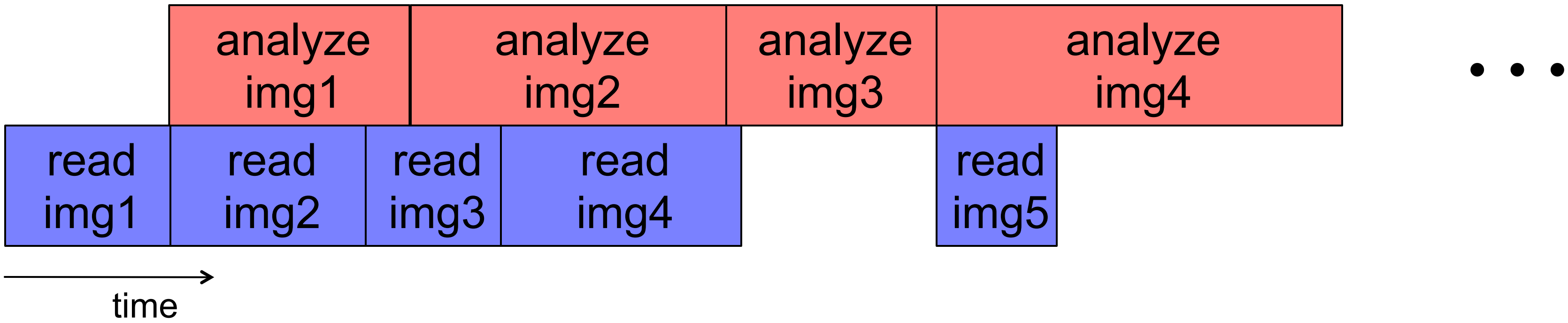
- This is called Inter Process Communication (IPC)
- IPC comes in several shapes or form
  - IPC via files
  - IPC via pipes (see ICS332)
  - IPC via sockets (as if on a network)
  - IPC via shared message queues
  - ...
- So we have a way for process A to send a message to process B
  - For our application: the Reader can tell the Analyzer “I have just read image #i”

Process-based Implementation



- Two processes:
  - P1: for image reading
  - P2: for image analysis

Processes: No Go :(



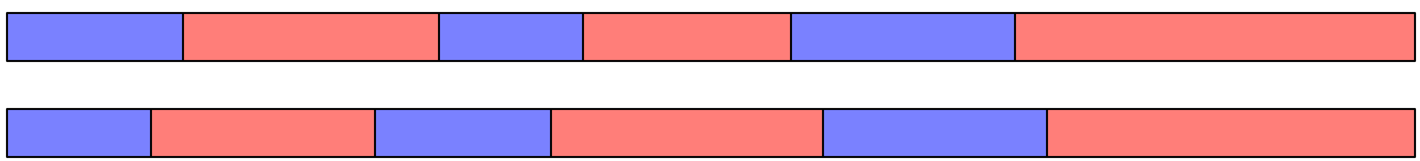
- **This doesn't work:**
  - P1 reads images into its address space
  - **P2 cannot access P1's address space!**
  - Processes are designed not to share memory space
  - Your "washer" and your "dryer" are each in its own parallel universe
- So we just cannot do a pipeline, end of story :(
- Can we do anything with processes? Any ideas??

Split the work in two…

- Without shared address spaces one could say:
  - I have N images to process
  - I am going to use 2 processes and each process will process N/2 images
- Execution could look like this

Process #1

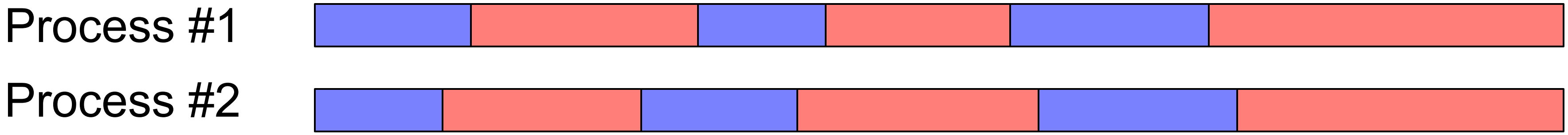
Process #2



- Why is this not so great?

Split the work in two…

- Without shared address spaces one could say:
  - I have N images to process
  - I am going to use 2 processes and each process will process N/2 images
- Execution could look like this



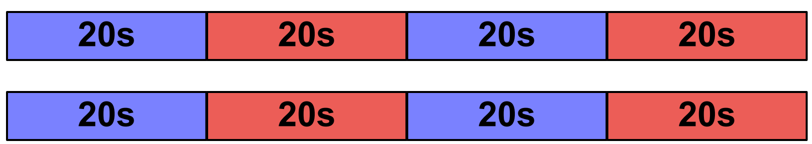
- Why is this not so great?
  - CPU idle time!
  - Competition for resources!

Competition for resources :(

- Say that
  - all images are identical, and we have 4 of them
  - it takes 10 seconds to read one image from disk
  - it takes 10 seconds to analyze the image on a core
  - we have one disk and one core

Process #1

Process #2



Total = 80s

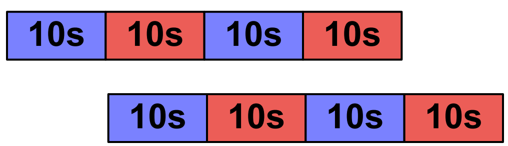
- This is a very inefficient use of the resources
  - We go as slowly as without concurrency!!!
- It would be better to organize the computation differently…

Avoiding competition

- Say that
  - all images are identical, and we have 4 of them
  - it takes 10 seconds to read one image from disk
  - it takes 10 seconds to analyze the image on a core
  - we have one disk and one core

Process #1

Process #2

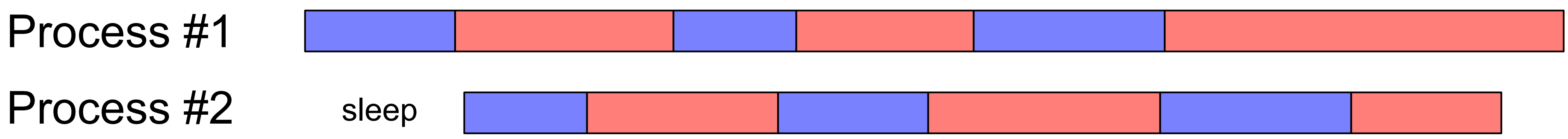


Total = 50s
(was 80s)

- Just have Process #2 start with a: sleep(10);
- No competition for resources at all
- Perfect overlap of I/O and computation!

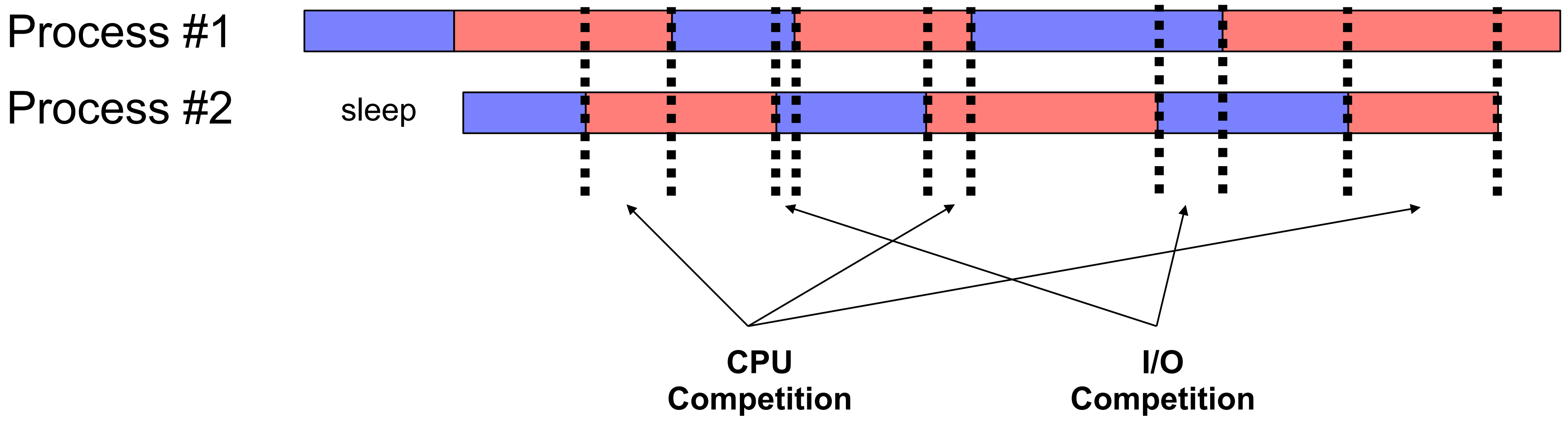
Not always so easy

- If every operation takes 10 seconds, we're good
- But if not, things are not so great



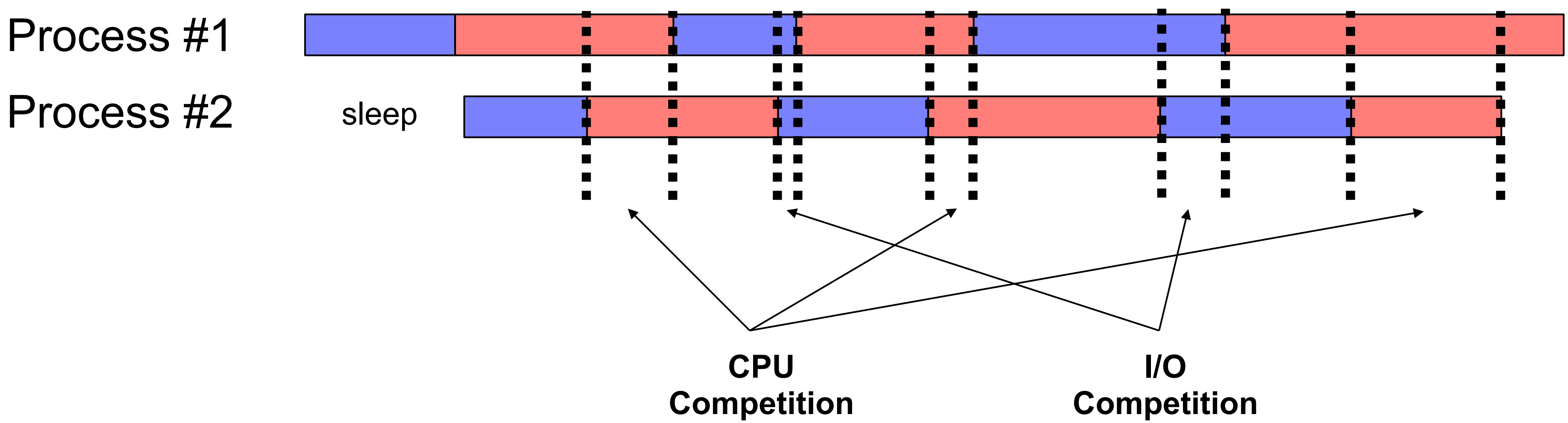
Not always so easy

- If every operation takes 10 seconds, we're good
- But if not, things are not so great



Not always so easy

- If every operation takes 10 seconds, we're good
- But if not, things are not so great

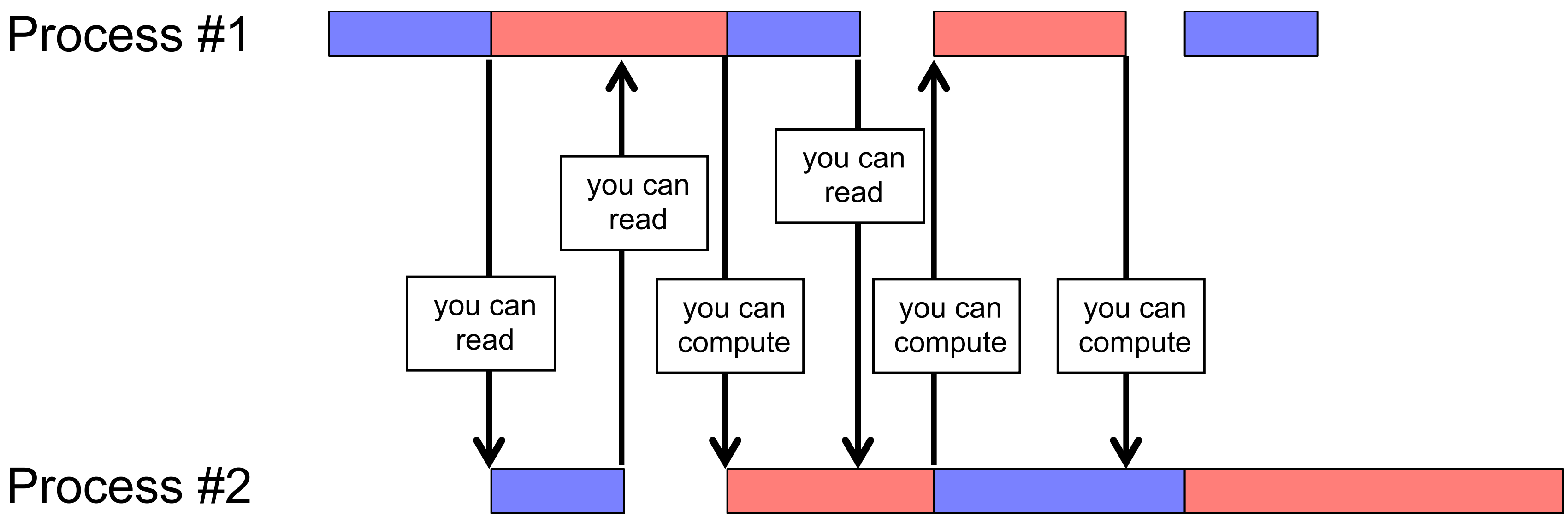


- And therefore the above picture is not to scale: All "competition areas" must be doubled in length
- Worst case: we go almost as slow as sequential!

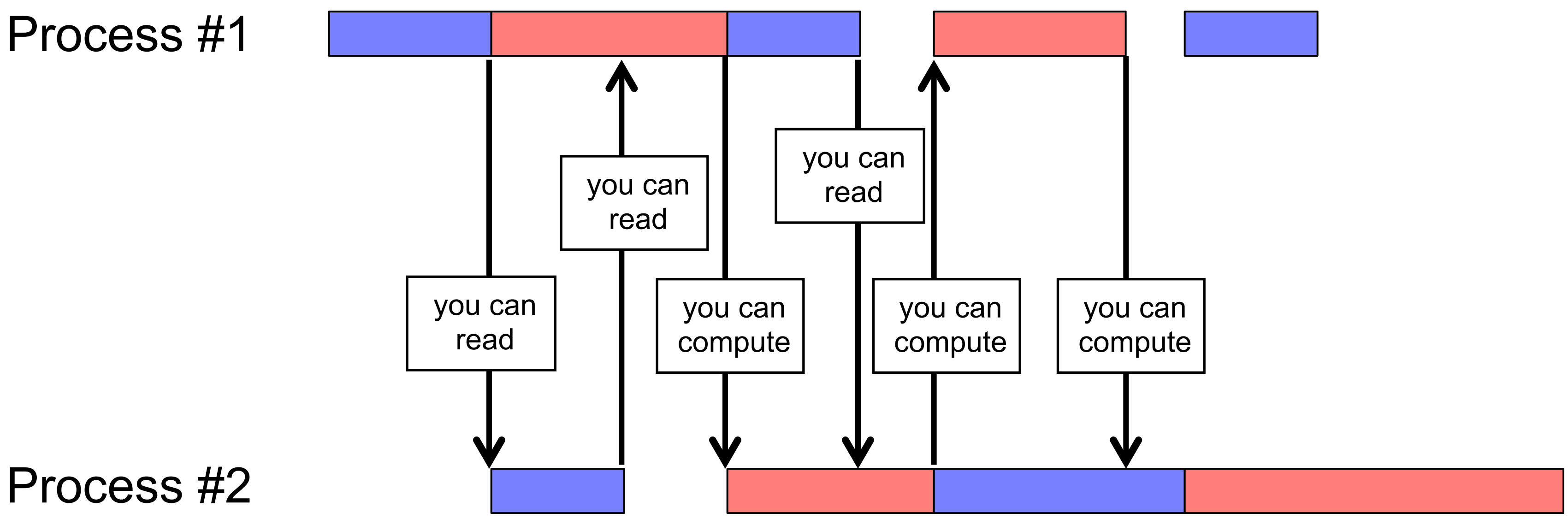
Avoid competition via communication

- The solution: have processes talk to each other
  - e.g., Process #1 says “I am done reading, go ahead and use the disk”
  - e.g., Process #2 says “I am starting computing, so please don’t compute right now”
- Easy to do with IPC
- Let’s see what our execution could look like if we have the processes communicate

Communicating Processes



Communicating Processes



- There is never any competition!
- But there are times during which the disk is idle or the CPU is idle (the "gaps" above)
- This cannot be avoided with the above I/O and computation times
- Anybody sees what a problem might be?

Sadly, not so easy…

- The previous picture assumes a nice “ping pong” effect
- But things can be much more complex
- For instance:
  - Images are all different, and some are quick to analyze
  - Therefore, one process can overtake the other and need to read twice in a row
  - Our simple “you go; you go; ..” synchronization doesn’t work
- So we need to come up with a more complex communication scheme:
  - “you go; but when I can go and you haven’t yet told me that you went, then never mind I’ll go…”

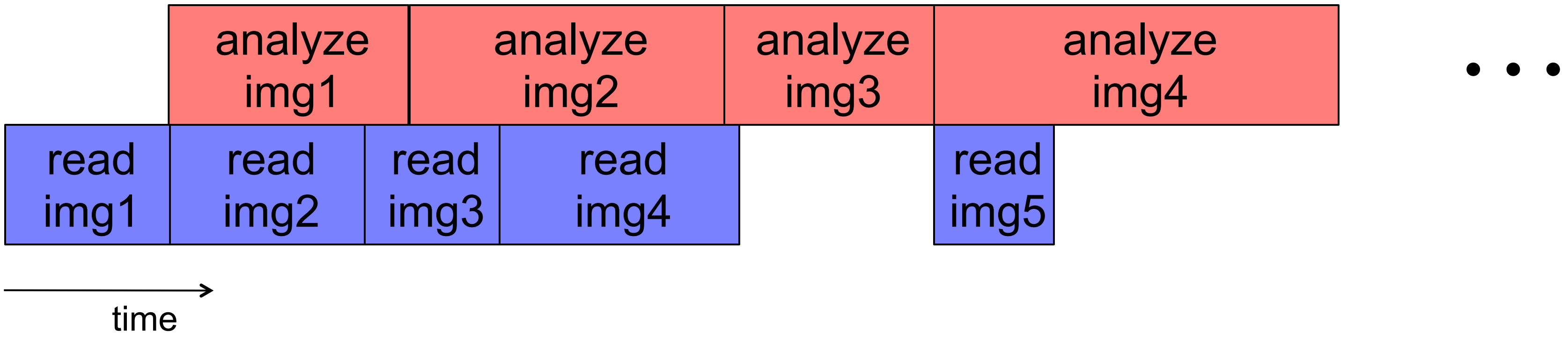
Even worse…

- Let’s say images are all different, with some easy to analyze and some hard to analyze
- And let’s say it doesn’t depend on the image size, meaning that it’s always a surprise whether an image is “easy” or “hard”
- We could be unlucky and give all the harder images to one process, and all the easier ones to the other!
  - One process will compute alone at the end, sequentially!
  - So our initially strategy “each process gets N/2 images to process” doesn’t work
- This is called load imbalance
- We would have to make synchronization more complicated, with a process “grabbing” the next image dynamically and telling the other process which image that was
- We could go down that route, but it’s getting really annoying
  - And yet, we’ll do things like that later in the semester

Where are we now?

- Using communicating processes has issues
  - 1) The communication patterns could be more complex that the basic ping-pong
  - 2) Load-balancing must be good
- Coming up with a good strategy is an interesting problem
  - And many people have investigated approach for many scenarios (including scenarios in which one must use processes, e.g., on different machines)
- But, if we abandon processes altogether, we may be much better off…

Back to Sharing Memory



- What we really, really want is the above picture, i.e., what we started with
- We want to **share memory** across processes
  - An image reader process
  - An image analyzer
  - The data is read in memory by the reader is used by the analyzer!

Share Memory between Processes?

- This idea of sharing memory among processes goes completely against the notion of clean, separated address spaces provided by the OS
  - Virtual memory is all about separation, not sharing!
- But, clearly it would be useful and would make programming concurrent applications much simpler
- As a result, there are mechanisms to share memory between processes
- Linux provides a “shared memory segment” abstraction
  - One process creates a zone of sharable memory
  - It then tells another process: here is a zone we can share

Shared Memory Segments

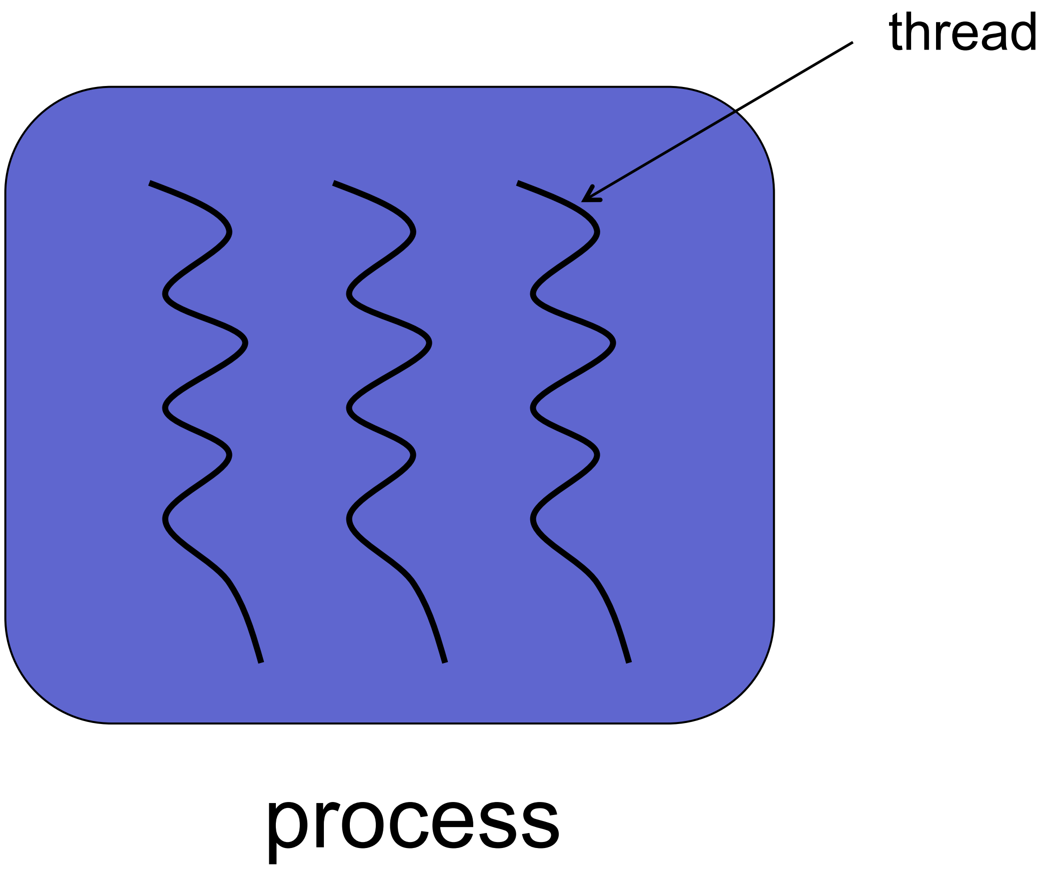
- The idea of shared memory segments is useful, but programming with them is very cumbersome
  - Many lines of code and bookkeeping
- We won't study them in this class, but ICS332 may have discussed them
  - The same principles about concurrency apply, so if you have to use shared memory segments for some reason, it shouldn't be very difficult after taking this class
  - Look at the man pages for shmget, shmat, shmdt, ...
- Nowadays, we typically don't use processes but instead use threads ("within" a process)

Threads

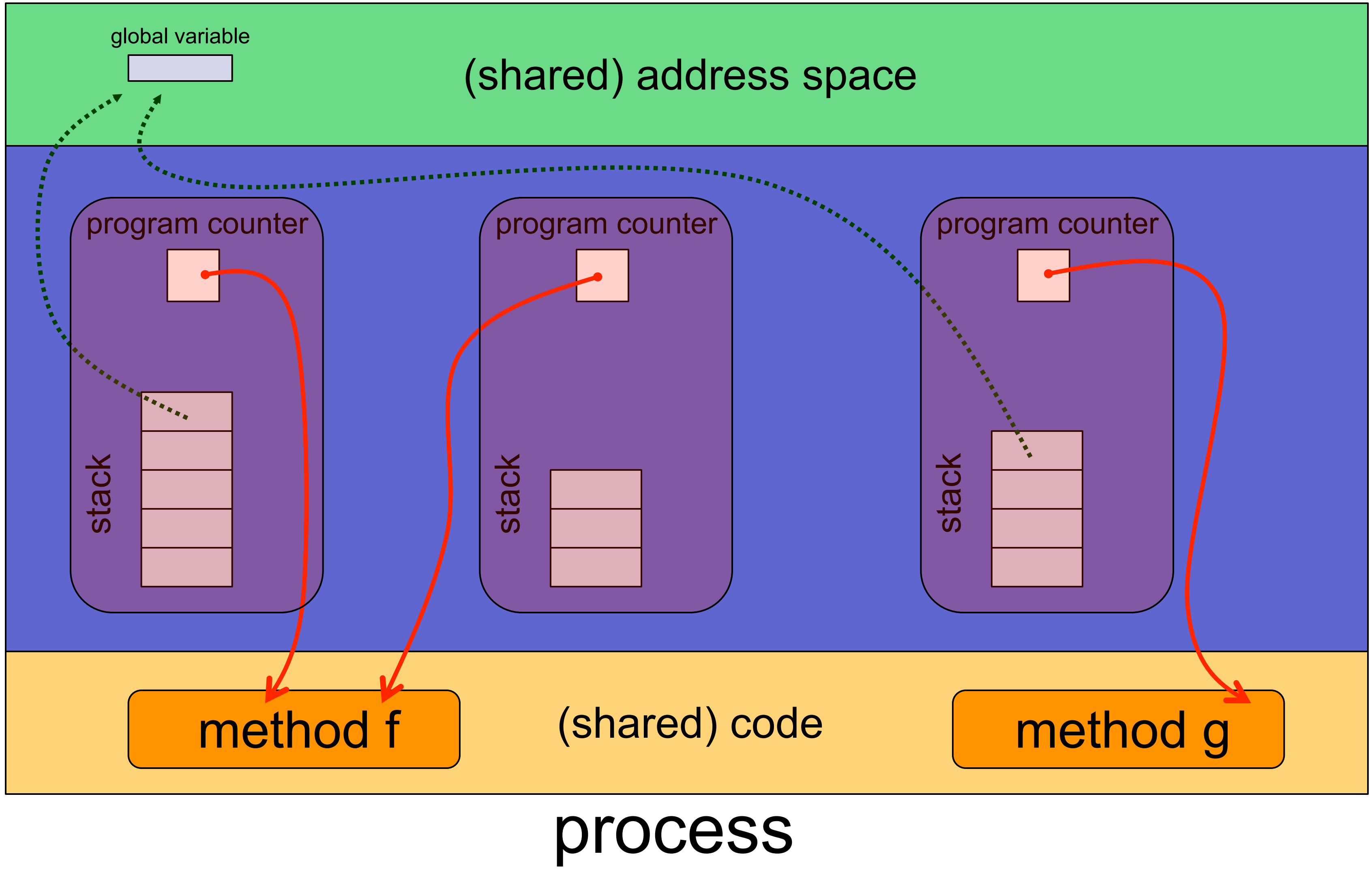
- Threads came about because of the need to write concurrent applications, that is the need for "tasks" that share memory
- Threads can be thought of as processes that share a single address space
- Threads are sometimes called "lightweight processes"
  - N processes have N page tables, N address spaces, N PIDs, ...
  - N threads together have 1 page table, 1 address space, 1 PID
- Things that threads do **not** share: program counter and stack
  - N threads have N program counters
  - N threads have N stacks
- Therefore, multiple threads can be executing different parts of the program "at the same time", and have followed completely different calling sequences

Threads in a Process

- Typical (but probably useless) representation



Threads in a Process

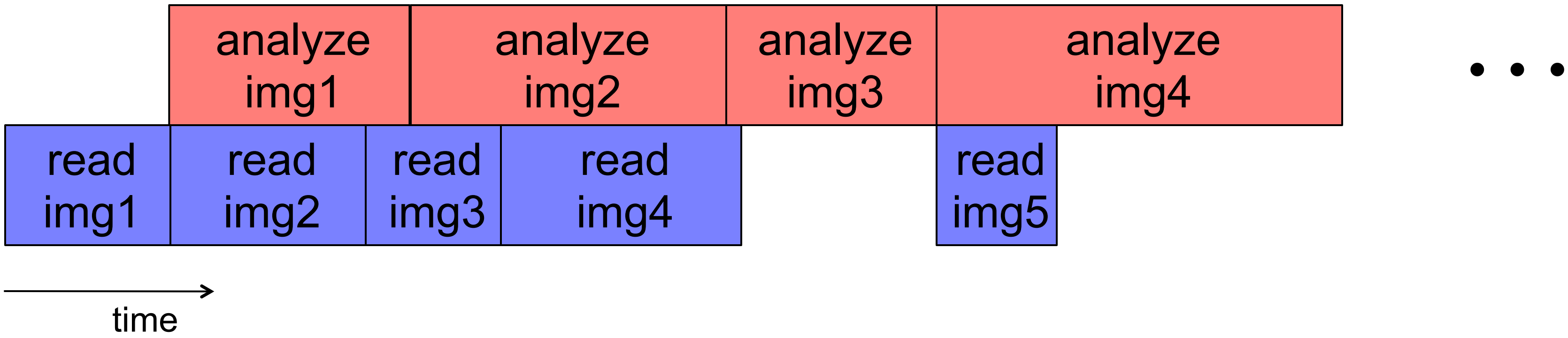


Threads vs. Processes

- Sharing memory with threads is straightforward
  - They were designed especially for this
- **But threads do not benefit from memory protection**
  - Can cause nasty bugs, which we will see at length

- Concurrent applications today are almost always written with threads
- What about Keynote?
  - Let’s find its PID
  - Let’s call ps with the “M” option

Threads as Tasks



- Each task is a thread:
  - An *image reader thread* that loads images into the process' address space
  - An *image analyzer thread* that analyzes images in the address space
- These threads need to communicate:
  - The analyzer has to wait for the reader to have read stuff in
  - The reader has to tell the analyzer that it has read something in
- But now we don't need IPC, we can just communicate in RAM (i.e., using variables!)
- We will implement this shortly, in our JavaFX application
  - after we learn more about multi-threaded programming!

Conclusion

- Most of the programs you use every day are multithreaded
- In the next module we'll review how to write multi-threaded programs, in Java
  - A screencast of ICS332 material
  - And then a lecture on more in-depth material
- Multi-threading is NOT new
  - Around for decades
  - Even part of ancient programming languages
    - IBM's PL/I F, Modula, Ada, etc.
- It's just became crucial due to multi-core (and GPUs), and now we cannot escape it (hence this course)
- Before the next lecture: Watch the "Java Threads" screencast