ALLEGHENY COLLEGE

Operating Systems:
Chapter 2
Semaphores
CS400

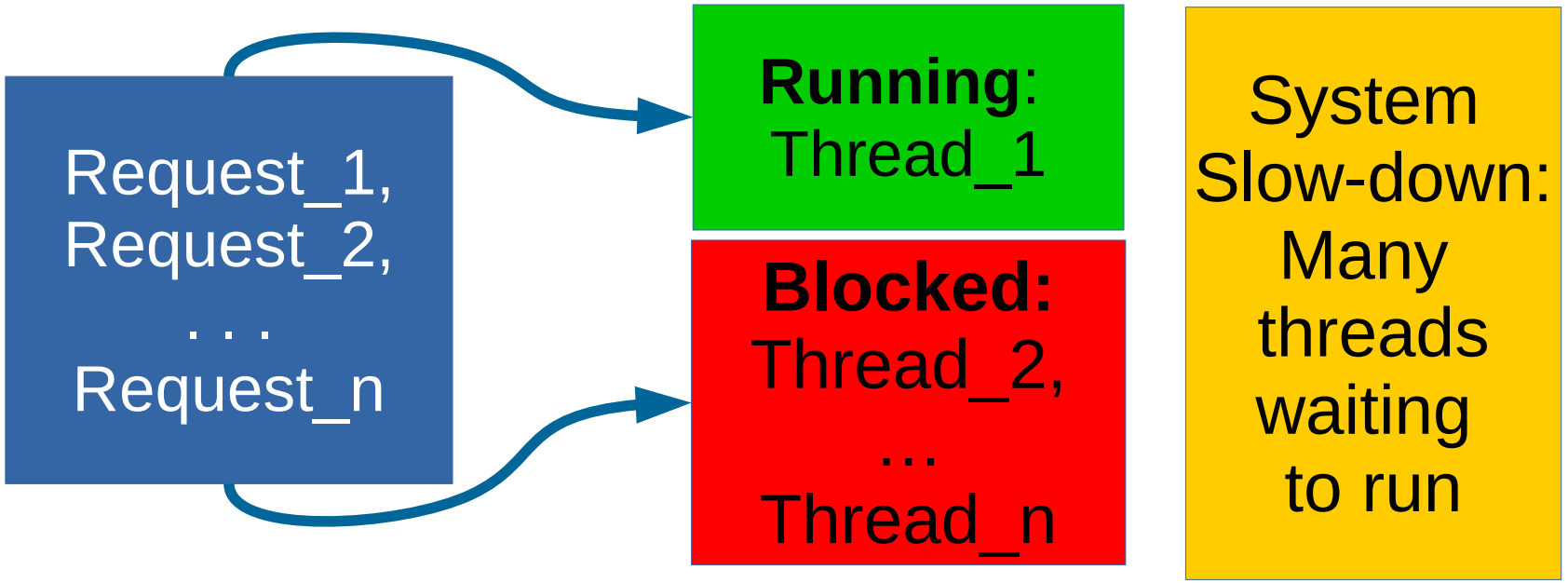
Week 4: 6th Feb

Spring 2020

Oliver BONHAM-CARTER

Over-Threading in Software

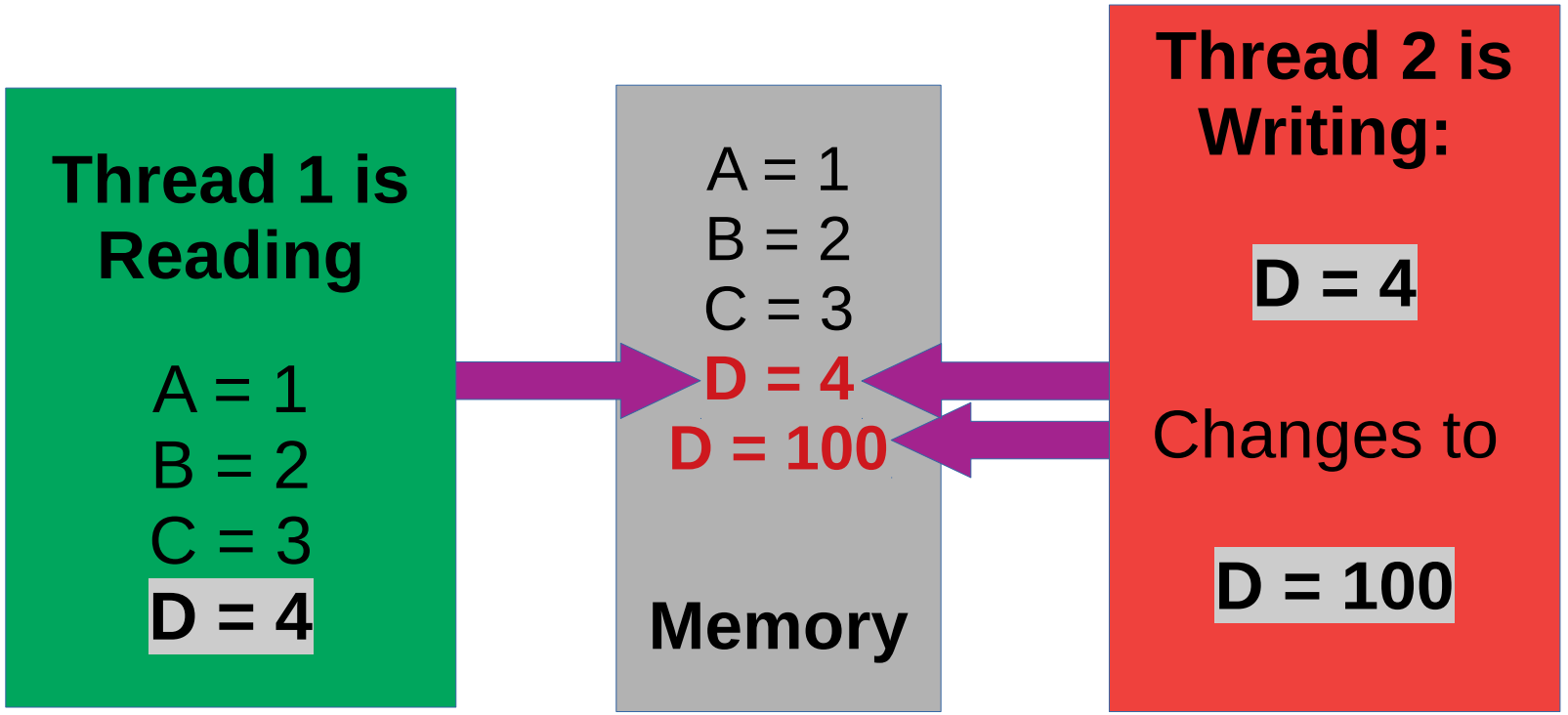
- Threads are helpful in software for “background work”
- Some developers use too many threads that are prone to blocking in code and make system calls to complete.
- Ex: A multi-threaded web server where requests are queued to threads to complete when resources become available.



ALLEGHENY COLLEGE

Controlling Who Reads and Writes

- Problems with threads include when one is reading and the other is writing at the same time



A problem has been detect and Windows has been shut down to prevent damage
to your computer.

DRIVER_IRQL_NOT_LESS_THAN_OR_EQUAL_TO

If this is the first time you've seen this Stop error screen,
restart your computer. If this screen appears again, follow these steps:

Check to make sure any new hardware or software is properly installed.
If this is a new installation, ask your hardware or software manufacturer
for any Windows updates you might need.

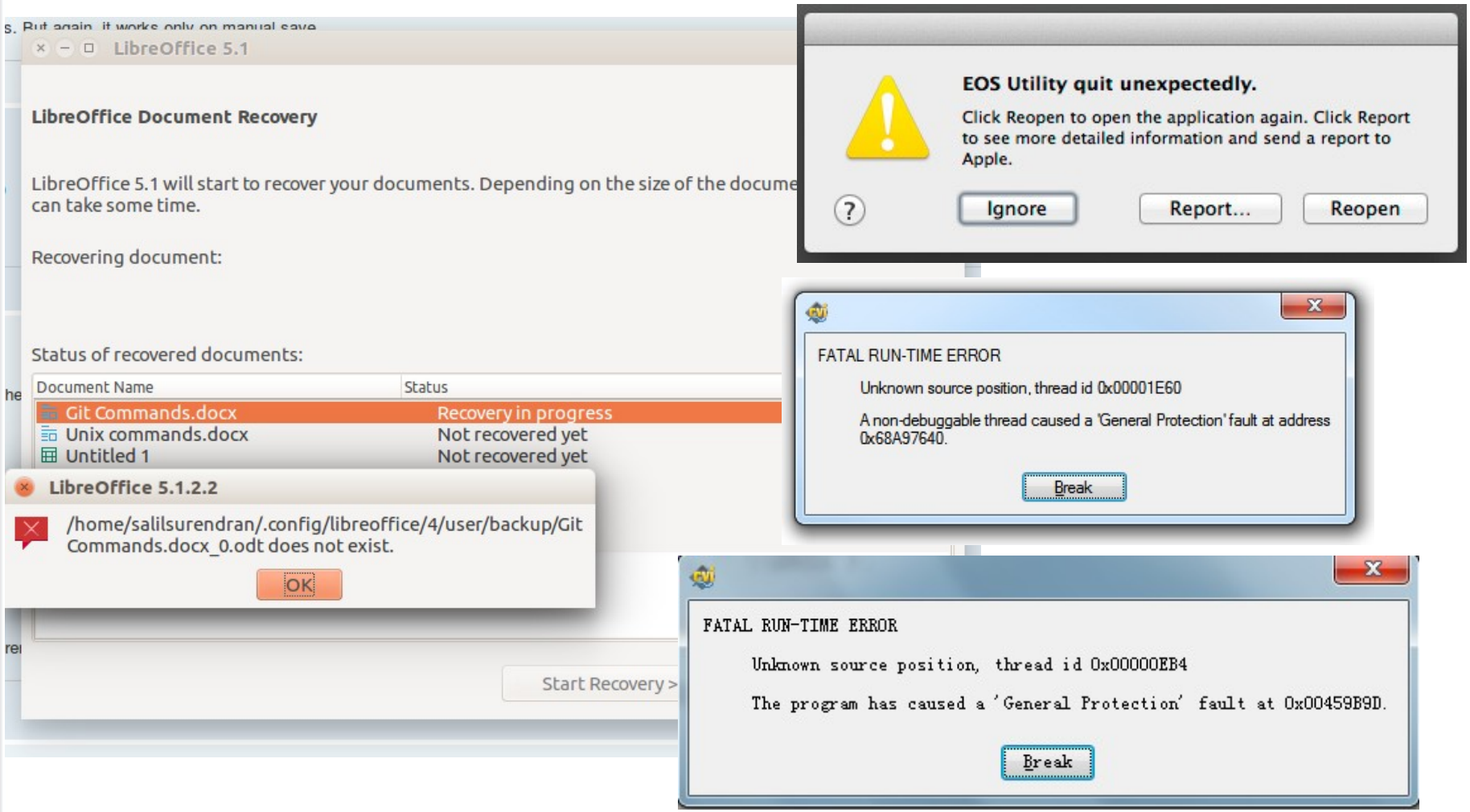
If problems continue, disable or remove any newly installed hardware
or software. Disable BIOS memory options such as caching or shadowing.
If you need to use Safe Mode to remove or disable components, restar
your computer, press F8 to select Advanced Startup options, and then
select Safe Mode.

Technical information:

*** STOP: 0x000000D1 (0x00000000, 0x00000000)

System Crashes

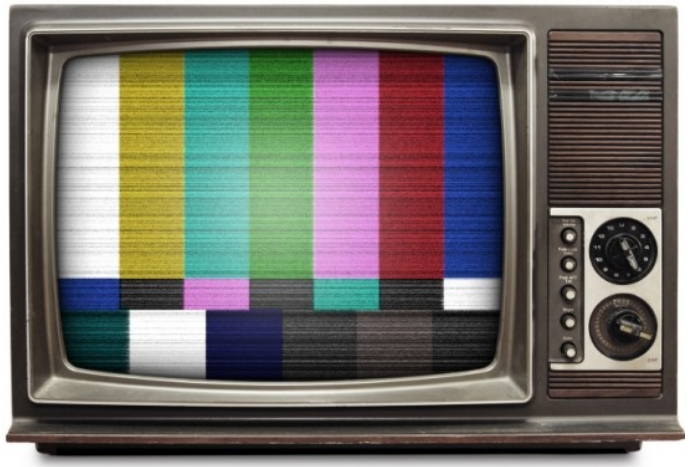
- Could badly programmed threads have caused your crash?



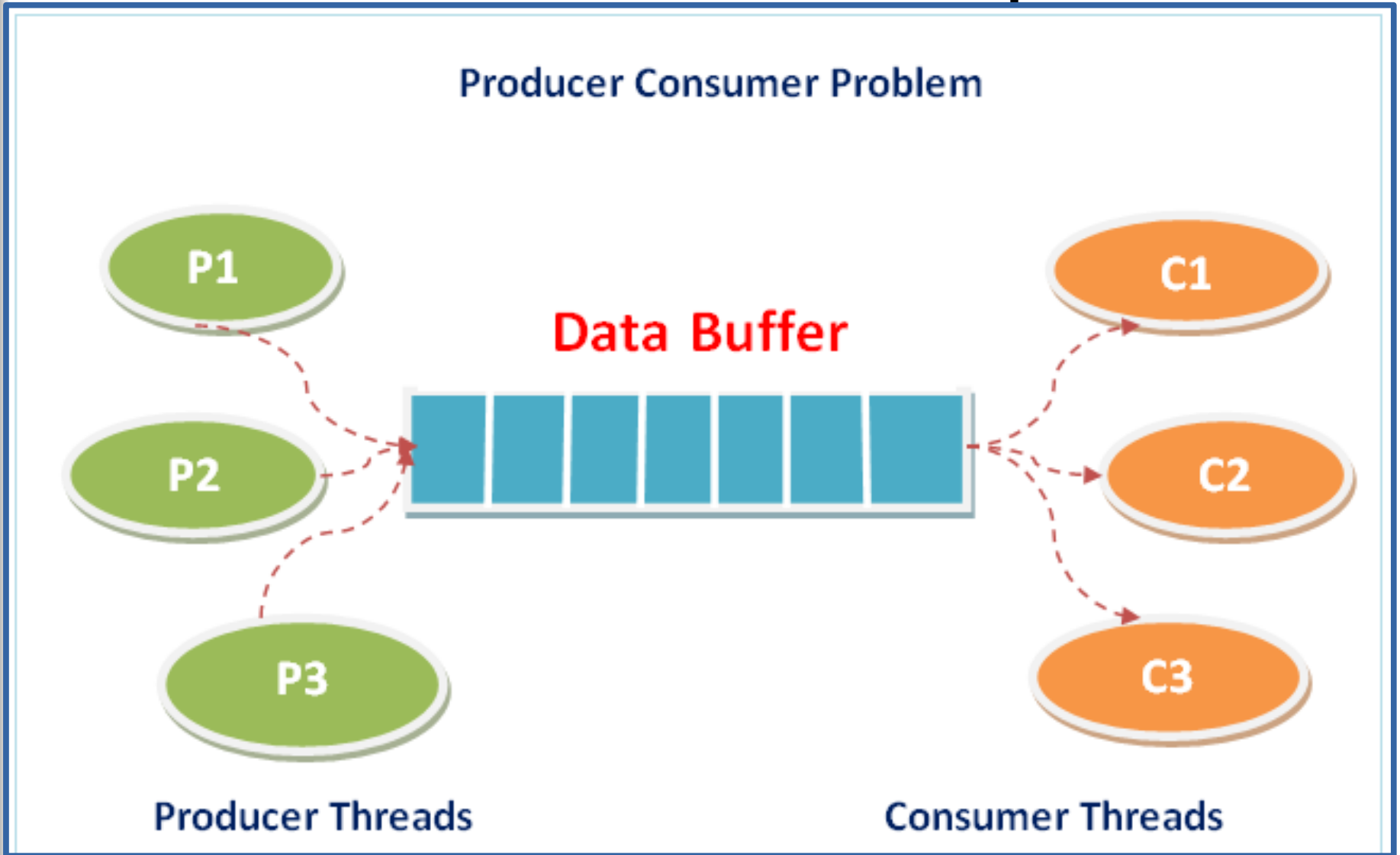
ALLEGHENY COLLEGE

Video: Processes and Threads

- How do processes and threads well work together?
- *Processes and threads - Gary explains*
- https://www.youtube.com/watch?v=h_HwkHobfs0&list=PLEYfCZIG4wG8-5QAV-xRxBHgzyl7T4Oi6



A Motivation for Semaphores

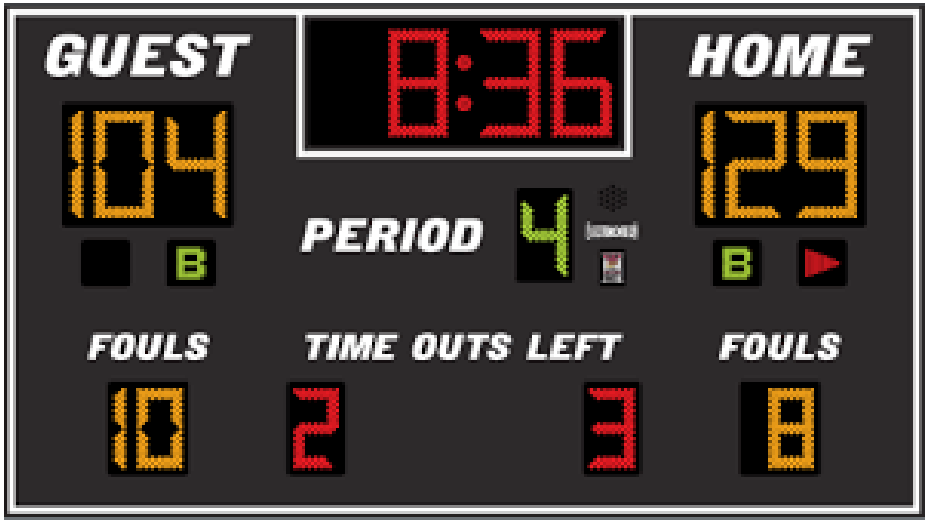


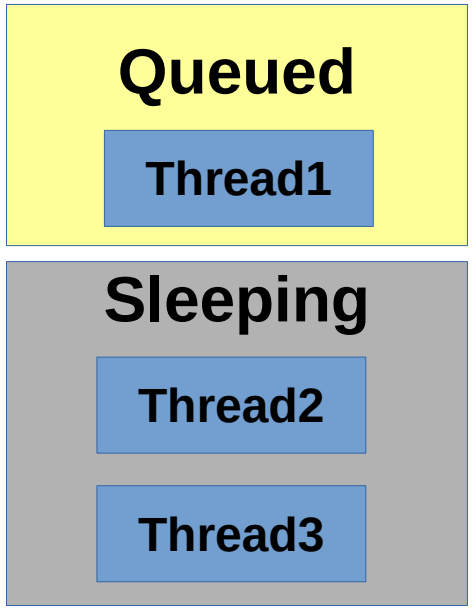
How can we be sure that the product is available for consumers?

How To Control Who Gets To Go?

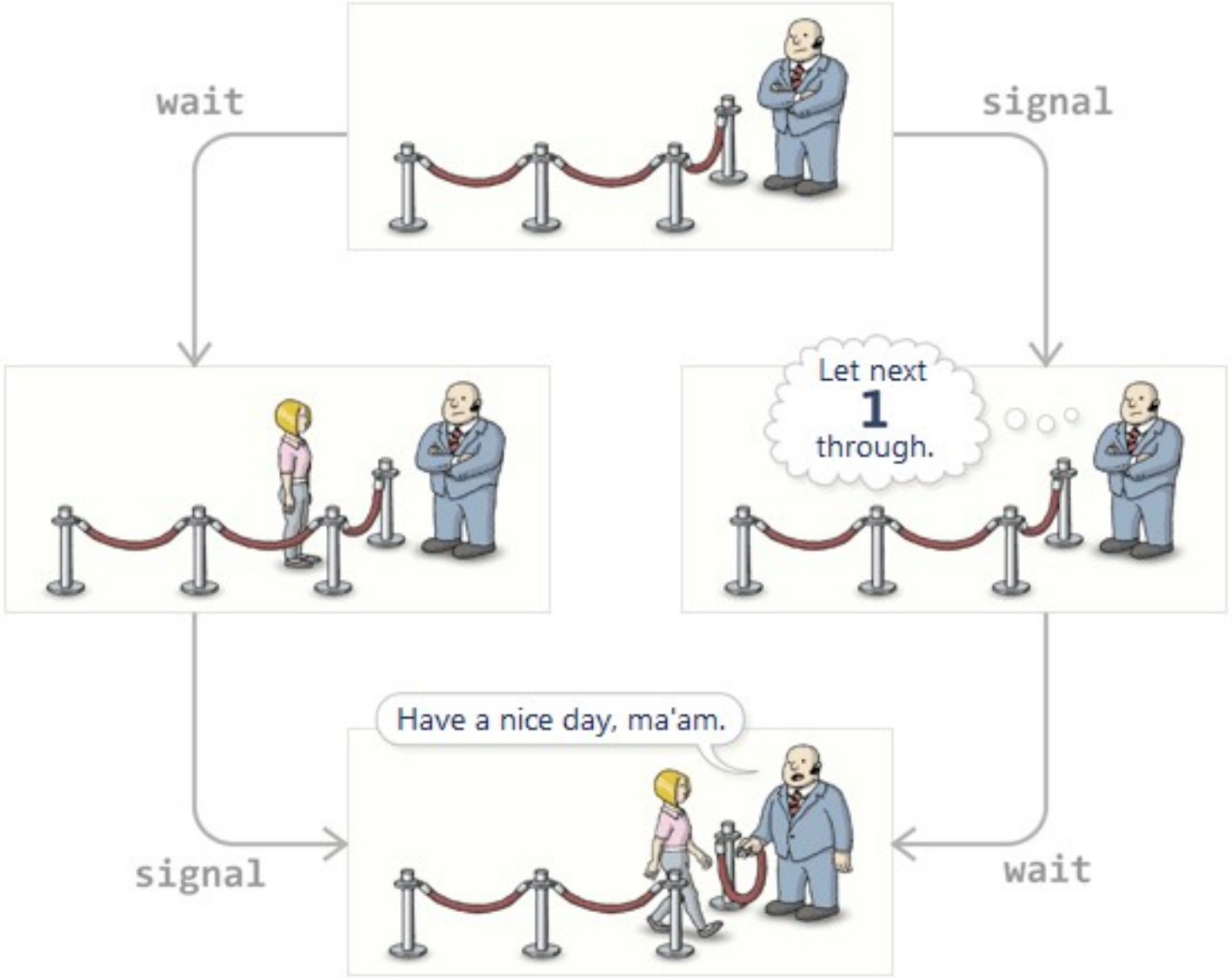
Semaphores!

- A *clocking* system to determine which thread gets to run at a particular time.





Semaphore Example



ALLEGHENY COLLEGE

A Motivation for Semaphores

- Consumers take *goods* which are prepared from producers. *In that order, ONLY*.
- Parts of the OS consume data which is created by operations in a certain order.
- Ex: A pipe-command:
  - **`cat file.txt | grep “hello” > outputFile.txt`**
- How do we ensure (synchronize) that each thread from each process completes in this logical order?

Semaphores: to handle the availability of resources.

Semaphores Pseudocode

```
struct semaphore {
        int value;
        queue L; // l is a list of processes
}

wait(S) {
        if (s.value > 0)
                s.value = s.value-1;
        else {
        add this process to S.L;
        block;
        }
}

signal(S) {
        if(s.L != EMPTY) {
                remove a process P from S.L;
                wakeUp(P);
        }
        else {
                {S.value = s.value + 1}
        }
}
```

The *wait()* and *signal()* are critical methods and must be atomically executed (all or nothing), with respect to each other.

ALLEGHENY COLLEGE

Threads Synchronized by Semaphores

- Each semaphore has an associated queue of processes /threads. Below w*ait()* and *signal()* called by threads.
- ***Wait()*** **Called by a thread. Decrements S by 1 and threads stall.**
  - If semaphore available; threads are stalled if S = 0
  - If semaphore unavailable: thread blocks, waits on queue.
- ***Signal()*** **Called by a thread. Increments the semaphore: first available thread can run.**
  - If thread(s) are waiting on queue, one thread is unblocked and run.
- If no threads on queue, signal is used for the next *wait()* call.

Semaphore Initialization

- **S = 1 (One *resource* is available)**
  - First call to *wait() by thread1*
    - **S = S – 1 = 1 – 1 = 0**. **(The resource is now taken by thread 1)**
  - Second call to *wait() by thread2*
    - S = 0, thread2 must wait to use the resource
  - If thread1 calls *signal(), thread2 takes resource*
    - **S = 0**: Thread1 left, so **s = 0 + 1 = 1**, and then thread2 took resource and **s = 1 – 1 = 0** again
    - Thread2. previously awaiting turn, is able to use resource
  - If second thread calls *signal()*
    - **S = S + 1 = 0 + 1 = 1**
    - Next caller to *wait()* will not have to wait.

ALLEGHENY COLLEGE

The Producer-Consumer Problem

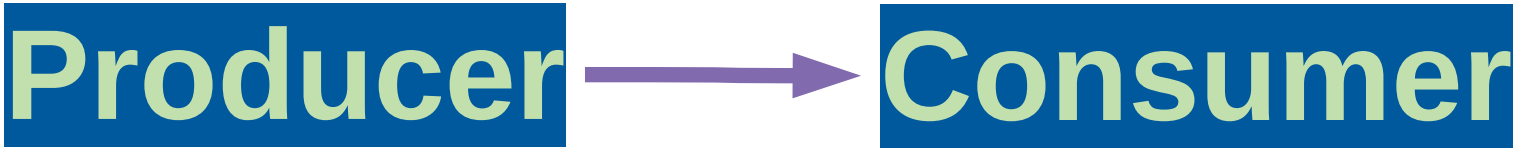
A classic problem this is used to describe the multi-process synchronization problem (i.e. synchronization between more than one processes).

Producer

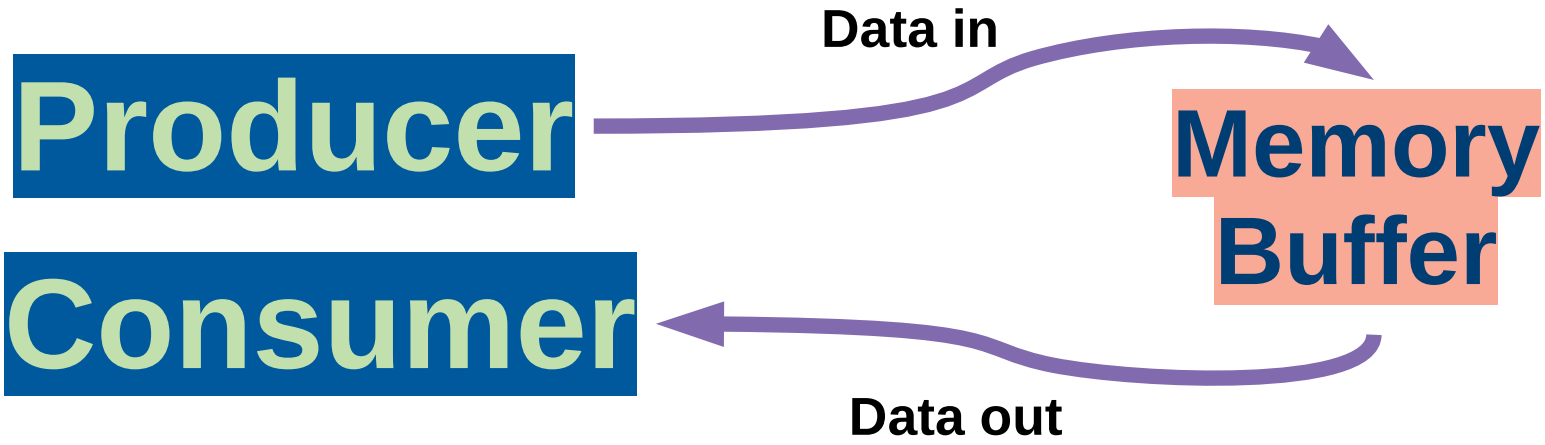
Consumer

The Producer-Consumer Problem

- A Producer produces something *(i.e. data)*
- A Consumer consumes something from the Producer.



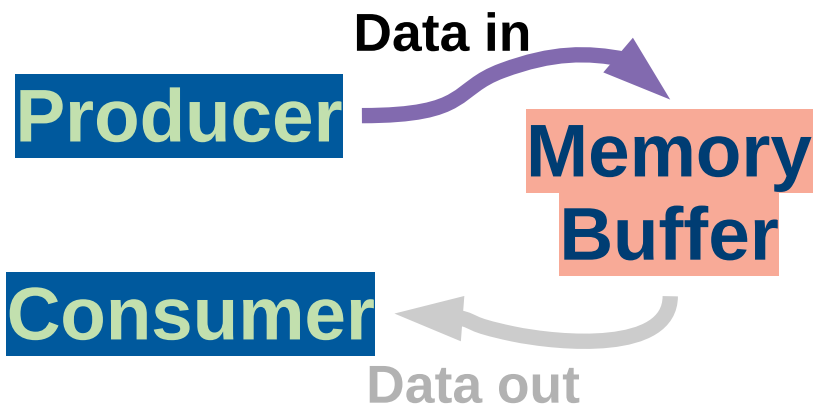
- The producer and consumer share the same memory buffer of a fixed-size

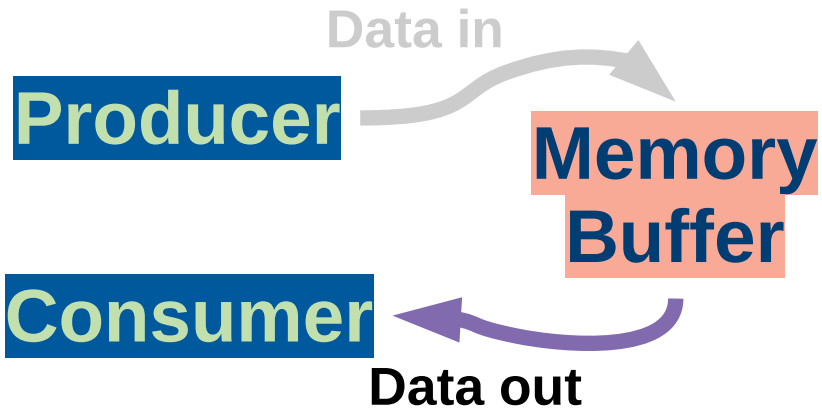


The Producer-Consumer Problem

Requirements:

- The producer can only produce data when the buffer is not full. (If the buffer is full, then no more data can go into the buffer)
- The consumer can only consume data when the buffer is not empty. (If the buffer is empty, then the consumer is not allowed to take any data from the buffer.)
- The producer and the consumer cannot access the buffer at the same time.





The Producer-Consumer : Semaphore for a Solution

In the producer-consumer problem, we use three semaphore variables:

- **First**
  - **Semaphore S**: This variable is for mutual exclusion between processes; either Producer or Consumer will be allowed to use or access the shared buffer at a particular time. This variable is set to 1 initially.

ALLEGHENY COLLEGE

The Producer-Consumer : Semaphore for a Solution

In the producer-consumer problem, we use three semaphore variables:

- **Second**
  - **Semaphore E**: This variable defines the empty space in the buffer. Initially, it is set to the whole space of the buffer (i.e. value is "n" because the buffer is initially empty.)

ALLEGHENY COLLEGE

The Producer-Consumer : Semaphore for a Solution

In the producer-consumer problem, we use three semaphore variables:

- **Third**
  - **Semaphore F**: This variable defines the space that is filled by the producer. Initially, it is set to "0" because there is no space filled by the producer initially.

ALLEGHENY COLLEGE

Semaphores In Action: Pseudocode

```
void producer() {
    while(T) {
        produce()
        wait(E)
        wait(S)
        append()
        signal(S)
        signal(F)
    }
}
```

- Each of the three semaphores are called using the **`wait()`** and **`signal()`** functions.
- **"Resource not Available"**: The **`wait()`** function decreases the semaphore variable by 1
- **"Resource Available"**: The **`signal()`** function increases the semaphore variable by 1

ALLEGHENY COLLEGE

Semaphore F: This variable defines the space that is filled by the producer.

```
void producer() {
    while(T) {
        produce()
        wait(E)
        wait(S)
        append()
        signal(S)
        signal(F)
    }
}
```

Semaphore variables are altered by the `wait()` and `signal()` functions.

The `wait()` function decreases the semaphore variable by 1 and the `signal()` function increases the semaphore variable by 1

Semaphore E: This variable defines the empty space in the buffer.

```
void producer() {
    while(T) {
        produce()
        wait(E)
        wait(S)
        append()
        signal(S)
        signal(F)
    }
}
```

producer() creates data repeatedly

wait(E): to reduce the value of the semaphore variable "E" by one since there is a decrease in empty space when data added to buffer. When semaphore value E is equal to 0, there is no more space and production must end.

ALLEGHENY COLLEGE

Semaphore S: This variable is for mutual exclusion between processes

```
void producer() {
    while(T) {
        produce()
        wait(E)
        wait(S)
        append()
        signal(S)
        signal(F)
    }
}
```

wait(S): used to set the semaphore variable "S" to "0" so that no other process can enter into the critical section.

Use in Code

```
void producer() {
    while(T) {
        produce()
        wait(E)
        wait(S)
        append()
        signal(S)
        signal(F)
    }
}
```

append( ) function:
used to append the newly produced data in the buffer (i.e., A type of SAVE function)

Semaphore S: This variable is for mutual exclusion between processes

```
void producer() {
    while(T) {
        produce()
        wait(E)
        wait(S)
        append()
        signal(S)
        signal(F)
    }
}
```

signal(s): used to set the semaphore variable "S" to "1". Production is complete and processes can begin harvesting (i.e., using and unloading) data from buffer

Semaphore F: This variable defines the space that is filled by the producer.

```
void producer() {
    while(T) {
        produce()
        wait(E)
        wait(S)
        append()
        signal(S)
        signal(F)
    }
}
```

signal(F) is used to increase the semaphore variable "F" by one because after adding the data into the buffer, one space is filled in the buffer. Variable "F" is to be updated

Bounded Buffer Problem With Semaphores

Shared Memory

```
char buf(N);    // the buffer
int in = 0, out = 0;
semaphore chars = 0,
      space = N
```

- Cannot consume char until it is produced.
- After production: signal available chars, wait for space
- After being consumed: signal available space, wait for chars.

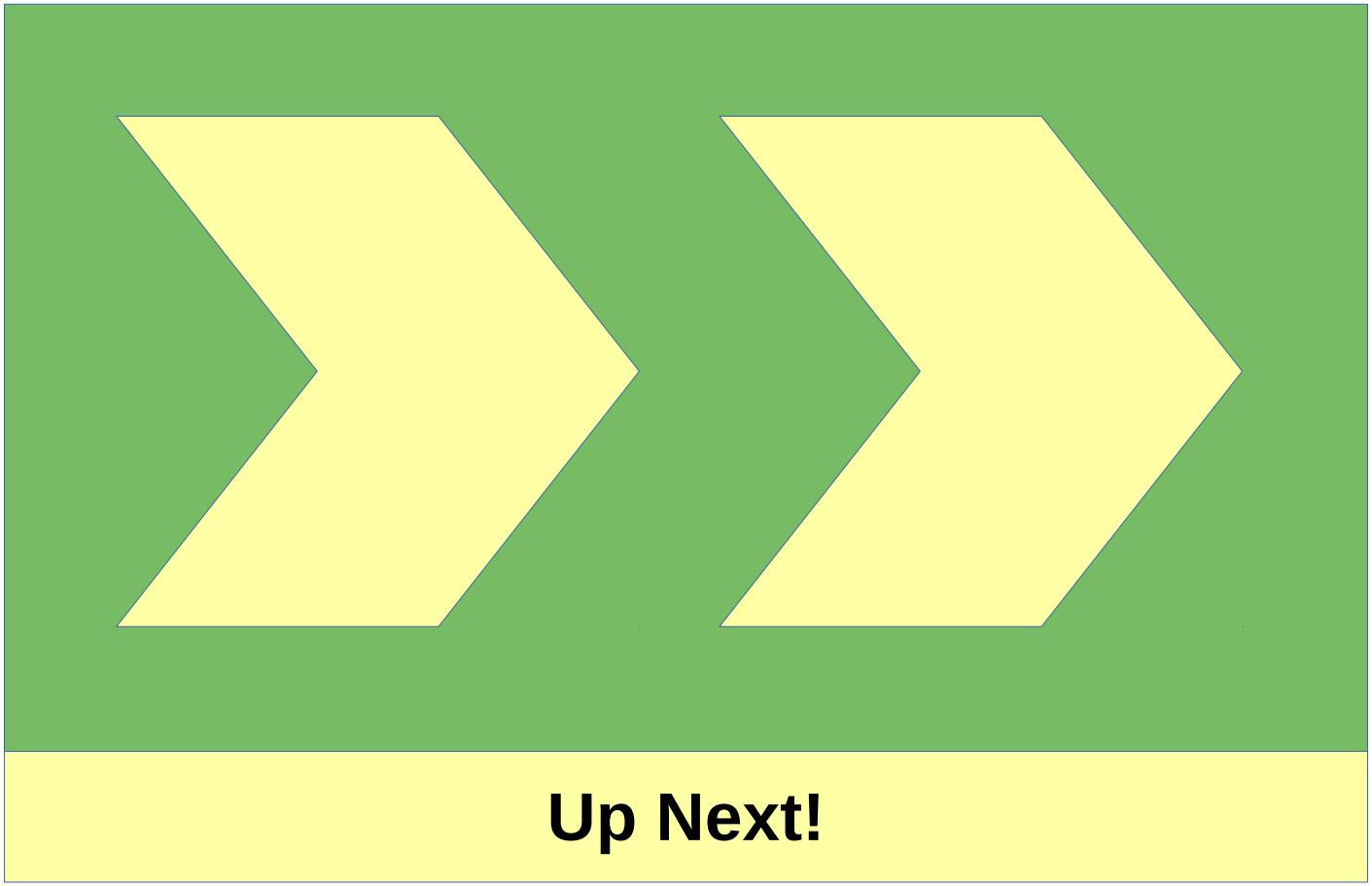
Producer

```
send char(c)
{
      wait(space) //buffer
      buf[in] = c;
      in = (in + 1)%N;
      signal(chars);
}
```

Consumer

```
char rcv()
{
      char c;
      wait(chars);
      c = buf[out];
      out = (out – 1)%n
      signal(space) //buffer
      return c;
}
```

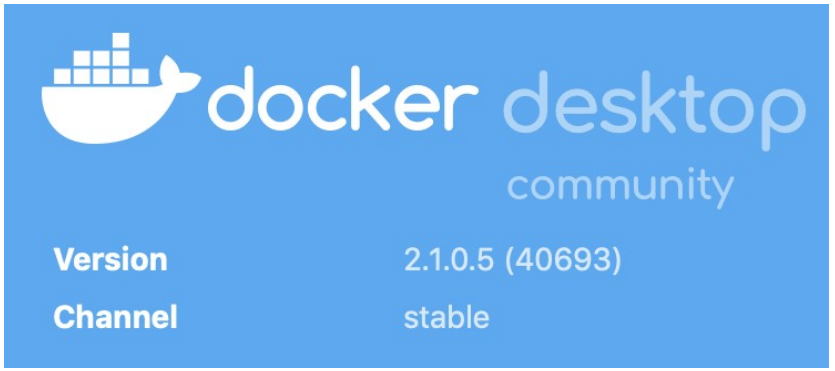
Let's Code!



ALLEGHENY COLLEGE

Commands to Run
From (Linux) Bash

- Build the container :
  - docker build -t gccdev .
- Run the container :
  - docker run -it gccdev
- Mount local drive and run container :
  - docker run -it --mount type=bind,source=$PWD,target=/home/gccdev gccdev



Note: the directory where you run this becomes your local directory in the container.