# Concurrency

Race condition: is when the result of a computation depends on the sequence of execution

A non-re-entrant function is one that cannot be used concurrently.

Cannot be safely interrupted and entered again

Need to be able to protect shared resources.

How can tasks be concurrent?

1. Unintentional concurrency
   1. Tasks are unaware of each other
   2. Hard to spot
   3. Often solved with external resource management
      1. OS manages this for you
2. Intentional concurrency
   1. Tasks work together to solve problems
   2. Difficult, yet common 🡪 we have library support that can help

Goal: tasks that compete for resources must not have their outcome affected by that competition.

We must guarantee a “serializable” order such that interleaving is equivalent to the correct execution

* We can accomplish this by restricting to 1 access to a critical resource at a time.
  + Mutual execution (mutex)
* Critical resource: a shared resource that requires exclusion
* Critical section: code that accesses a critical resource.

Mutex ensures serial access to a critical section

Mutex enables a task to lock resources so no other task can use them.

* If a task locks a resource and another task attempts to lock that resource, they become blocked.
  + It must wait until the other task is done and unlocks the resource.   
      
      
    A piece of paper with writing on it

    Description automatically generated with medium confidence
* 2 problems:
  + 1. Starvation: when a task never gets to access a resource, it is starved
    - E.g., 1 task locks a resource and never gives it up.
    - A task may not get a turn on the CPU in time to lock it.
    - Task 1 is high-priority; it may always get scheduled before task 2
  + 2. Resources R1 & R2, Tasks T1 and T2 can cause deadlock in the case below  
    Text, letter

    Description automatically generated  
    - To avoid a deadlock, we can use while((trylock())
* We must avoid deadlock and starvation while also avoiding race conditions while understanding all possible interleaving of tasks
  + This is left to programmers

# Threads

Processes – abstraction to help the OS to enable concurrency

A process requires:

* Machine code
* Address space
* State

What if we could have more than one program running in a process image? 🡪 threads

Why would we have more than one program running in a process image? To share data structures and functions

* E.g. one thread to monitor user input (slow)
* Another to update graphics (fast)
* Process the user input (bursty)

Breaking into threads:

* Sometimes simplifies programming
* Leverage multiple cores.

Threads enable:

* Tight sharing of resources between jobs
  + IPC is slow, hard, and tricky
* We get shared memory
  + But introduces concurrency issues like interleaving

Threads stand for thread of execution

* Most thread discussions mean more than one thread in one process
* Each thread has its own program counter, registers, stack, etc.
* But we share code, data, heap, address space, and a bit more
  + But this lets threads stomp all over each other

The thread is the active part of the process

Conventional vs threaded   
  
A picture containing letter

Description automatically generated

Threads share charactersitcs with processes

* State (running, blocked,…)
* Executes sequentially
* They can call functions, libraries, system calls, etc.

Creating threads is much cheaper than fork ()

* No copying of resources

Switching between threads is cheaper

* Shared heap, PCB, etc. means fewer resources to swap on a context switch

2 types of threads

1. User threads:
   1. Created in user space
   2. OS has no knowledge
   3. Thread is given part of the process! Time slice
   4. Thread operations are cheap, generally managed by some library

Used for

* 1. Cooperative systems: a thread may periodically call a function to let another thread run.
  2. Async programming: a system timer triggers a scheduling function
  3. Limited to 1 core

1. Kernel threads
   1. Created and managed by the operating systems 🡪 system calls
   2. OS can schedule and swap threads just like a process
   3. One thread can block while others run
   4. Compared to user threads, threading operations are more expensive.

Pthread

To start a thread, give a function as a starting point,

Can only pass 1 parameter (void\*)

We can not access the stack of another thread

Pthread-create()

* Create a new thread at a function
* Returns a data structure that represents the thread

Pthread-exit

* Terminates the calling thread

Pthread-join

* Waits for a specific thread to exit and get a status from it

If a joinable thread is not joined, it will cause a memory leak

Synchronization with pthreads

* Locks (mutex)
* Standard way to control access to a critical section
* Condition variables
  + Allow a wait/signal behaviour.

How pthreads works

Text, letter

Description automatically generated

Condition variables

* Enable us to block until a condition is true

How the condition variable works

Text, letter

Description automatically generated

Condition variables are very error prone

Problem: these condition variable functions are not atomic.

Solution: use a lock. Shown below

Graphical user interface, application

Description automatically generated

Problem: T1 is holding the lock, so T2 can’t get it to signal

Solution: unlock before sleeping

Problem: spurious wakeups

* While waiting on a condition variable, a task may unblock even there is no signal
  + Many reasons, one of which is for kernel implementation. Shown below

Text

Description automatically generated

# Semaphores

New approach:

* Let’s take the signal approach:
  + Instead of being busy waiting on a lock, a task can sleep until a resource is available

In the OS, we can use atomic operations to implement variables to signal between tasks

We define a semaphore:

* Init: start an int at some positive value
  + E.g., 1, 3, 17, etc…
    - Number of concurrent tasks allowed into a code section
* Wait(): decrement the int
  + If the value is <= 0
    - The task will block
  + Otherwise, continue
* Signal(): increment the value:
  + If the value is > 0, unblock
  + 1 waiting process

Counting semaphore happens as shown below

Semaphores need a queue to manage blocked tasks

* Raises the question of who we wake up.
  + - Scheduling
  + 2 options
    - Undefined order
      * Weak asynchronous semaphore
    - Define order
      * Scheduling like methods (FIFO, etc.)
        + Strong asynchronous semaphore

Semaphore code as shown below

Text, letter

Description automatically generated

The previous pic is implemented in the kernel (guaranteed not to interrupt [atomic])

We do not let programmers turn off interrupts to waiting semaphores because they can abuse this ability by allowing their program to run fast by stopping multi-programming, as their program can never be interrupted

Wait/signal are short, robust, predictable

Semaphores are in the kernel

Requires system calls

They are more computationally heavier than a busy wait mutex most of the time

Scale better than mutexes with resources, threads, and wait times

# Synchronization

Aka producer/consumer problem

E.g., task a prepares some work (like receiving server requests, data to process, etc.) task b does the work  
in this case, task a is the producer and task b is the consumer

Managing dynamic speeds of tasks so that production and consumption are (sort of) at the same rate.

E.g., if task a is faster, give more CPU time to task b and vice versa

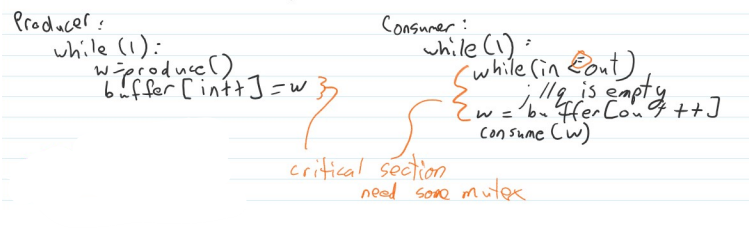
Quite difficult to do explicitly and happens to come up frequently in concurrent programs

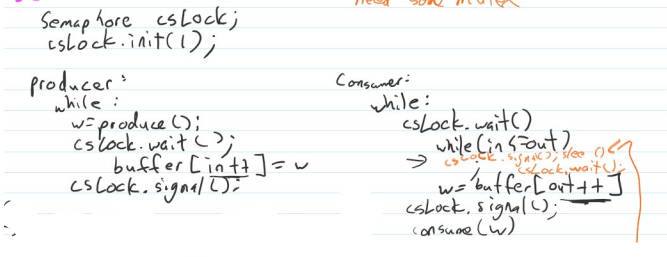
Solution: use a buffer

* If producer gets ahead, store for consumer to do later
* If buffer is full, use more buffer.

Correctness required

* Only 1 task can read/write to the buffer at a time
  + If not, race condition
* Consumer can not read from an empty buffer
  + Wait, let the producer catch up producing work
* Producer and consumer pseudocode is shown below



Producer and consumer pseudocode with semaphore mutex is shown below  
 

Problems with the above:

1. Deadlock
2. Assumed infinite memory
3. Consumer has a busy loop

Producer and consumer deadlock fix pseudocode is shown below

Text, letter

Description automatically generated

To fix the infinite buffer assumption issue

* Implement a circular buffer
* Producer should block if buffer is full

Producer and consumer code infinite buffer fix pseudocode is shown below

Text, letter

Description automatically generated

To create a whole data structure,

* Lock in and out separately instead of csLock
* Producer accessing the buffer will not slow down the consumer
  + Now multiple producers, multiple consumers with “fine-grained locking”
* We should replace in and out with n locks for a buffer size of n to lock each part of the array
  + Producer gets next available slot
  + Use the critical section in an if statement
    - If we can’t enter a critical section (trylock())
      * Implies someone took the lock
    - Try the next one

# Secondary Storage

Primary storage includes

* RAM
* Registers
* Etc.

Primary storage is volatile: once the power is gone, so is the data

Secondary storage is not volatile

Secondary storage includes:

* HDD
* SSD
* USB
* Punch cards
* Tapes
* Etc.

All of which are long term

A file system manages the info on SS

A file is simply a named (Unique ID) collection of information

Usually, a sequence of bytes

Files are data structures.

Properties of files:

* Permanent
* Can be read from or written to
* May be used by others
* Low-level 🡪 just bytes in storage
* Programs use files to implement higher level systems

In most systems the file is an OS concept

Files often are the only way to store data (for users and programmers basically)

Why files?

* They are often a standard way to store data
* Robust 🡪 user programs should not be trusted to access hardware
* Security 🡪 if in OS, OS can check every file access for permissions
* Mutex:
  + a way to let multiple programs share/protect files concurrently
  + allow multiple programs to use storage without overwriting each other

Files have

* data (chunk of bytes that are of some length)
* attributes (at the front) which store things like name, owner, permissions, etc…

Basic operations on a file

* Create
* Delete
* Read
* Write
* Seek 🡪 reposition the read/write pointer
  + In many systems, read/write are contiguous (next/together in sequence) and remember where they left off
    - Easy to use
* Truncate
  + Uncommon 🡪 shrink a file
* Change attributes
  + Highly OS + File system dependent

File types

* OS can associate a file type with a label to give hints on what to do with it
* Attributes: generally limited to what we can do with it, are not extensible (can’t be extended)
  + .exe, .bat, .txt, etc.
    - Flexible, but not reliable to the OS

Directories

* A directory is just an abstraction for organization
* Usually, hierarchical
  + Generally, a tree with a “root”
    - Usually, DAG
    - Directed
    - Acyclic
    - Graph

Implementation

* Programmers want a linear series of bytes
* The OS’s job is to make the linear series of bytes possible while using various types of secondary storage
  + The OS does that by taking in the various SS that are attached to the computer, it then provides one single interface for all of those SS, then any user program can access all these devices and all the files on them through the file system

Minimal Read/Write

* Technically can be difficult to read 1 byte
* Read/write in bursts is more efficient

Sector

* Is the smallest addressable unit of storage on a device
  + This is enforced by hardware
  + 512 Bytes or 1024 Bytes
    - Helps with optimization in the OS and hardware

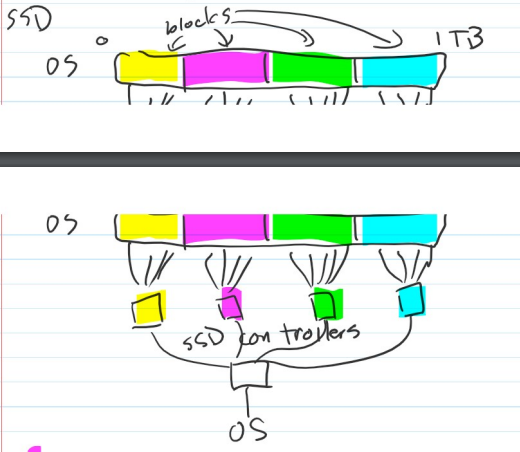
Downsides of bursty work and sectors:

* “Slack space” in other words wasted space.
  + If I can only address 512 bytes, what if I only want to store 100 bytes
* Reading small segments takes longer
* Reduces cache effectiveness
* Bad sectors are more expensive in terms of wear and tear
  + If one byte goes bad, the entire sector goes bad, and you cannot use it anymore

Traditionally the OS had to know the SS geometry

HDD 🡪 cylinders, heads, sectors

OS had to map logical byte (user view) to a physical location (hardware view)



So we get 1 contiguous view of SS

Partitioning

* Drives can be logically split into regions
* Isolation 🡪 files on 1 partition cannot take space on another
* Each partition become a virtual disk, can have its own file system, NTFs, mac journal, EXT3/4

Some files systems have max size

Master boor record (MBR) stores pointers to partitions and file system type

GUID Partition Table = MBR

In windows in UNIX

C: /dev/sda  
D: /sda1  
E: /sda2

# Basic File Allocation

How do we put files on SS

1. Contiguous allocation

It looked like this  
  
Text, letter

Description automatically generated

Problem: where are the files? Where are the empty spaces?

Solution: put a lookup table at the front of disk

It looked like this  
  
Text

Description automatically generated

Problem: fragmentation as files get added, modified, or deleted.  
We get left with many small spaces that add up  
  
Text, whiteboard

Description automatically generated

To fix this, we can defrag (compacting file system to remove gaps)

We can also break up a file. So, a file will have parts of it stored in many parts of the empty spaces

What if there is no space?

Then the OS is out of options. Fail and tell the user

Chained link allocation

Each file section could store a pointer to the next section or EOF

This will fill up the small chunks of free space (high utilization)

Seeking is “cheap” as we only follow a pointer

Files do not get fragmented

Heavy fragmentation eventually harms performance

Problem: the size of the table determines how many files we can store

To solve this: we can link to another table or use chained link tables (end of table points to the next)

This becomes an issue with larger files

How do we track free space?

Treat free space as one big file

Chain links the free space

We can use bitmap: 1 bit maps to one block

e.g., 1 bit maps to 1 sector

still need to search table for enough 1

It will still take space:

1GB has 2097152 sectors @ 512 bytes = 256KB

Indexed regions

Keep an array of groups of free space

Clusters (of sectors)

The file system itself may decide the sector is too small

Cluster is a software-enforced sector 🡪 sets minimum allocatable unit

Size is a multiple of sector size

Example: cluster of 2 sectors = 1024B (512\*2)

Large cluster sizes improve read/write performance but increases slack space

# File Allocation Table (FAT)

Anytime we searched on disk, we had to do a lot of seeking. We collected file information stored in 1 table

Instead, let’s collect all pointer from all clusters and put these in the table upfront, then we only have to hop in one location of disk. This allows forward planning, reducing seek.

The table becomes something like a bitmap, instead of storing 0/1. We store next location for the file chunk or EOF in the FAT.

Each fat entry corresponds with a cluster on disk with a ratio of 1:1.

Text, letter

Description automatically generated

Want to read a file on disk?

1. Read the corresponding FAT entry

2. check if FAT entry is EOF.

2a. If it is, read the data cluster that corresponds to that FAT entry

3. If not, Jump to FAT entry specified in the location we read in step one.

3a. Repeat until EOF.

4. Read all data clusters corresponding to the FAT entries just read.

Directories

Directories are just files defined by the file structure.

In FAT they are a list of directory entries.

Directories also have attributes like name, pointer to start of data, etc.

Letter

Description automatically generated