Lecture 1

Mon. Aug. 26, 2019

Classic Book (to learn C)

- kernighan & Ritchie, The “K&R” of C

C is still evolving

- Different standard version
  - K&R C
  - ANSI C
    - C89, C90 (ISO 9899:1990)
  - C99 1999
  - C11 2011
  - C18 2018

C should be...

- Simple
- easy to compile
- typed [weakly]
- support low-level memory access
- ideal for embedded controller, OS (operating system), …

Yet…

- C is powerful
- C is fast

C is a purely procedural language

- No object-orientation whatsoever

Central Dogma

- Object of interest: Computations
- Main abstraction tool: Procedures/ functions
  - Caller / Callee
  - Abstracts away “How things are done”
- Programming means
  - Organizing processes as procedures
  - composing processes thru procedure calls

Procedural Programming in C

- Adheres to the philosophy
- Generates very efficient code
- Exposes as many low=level details you wish to see
- Provides full control over memory management (no Garbage Collection)
- The Programmer is fully in charge

CSE 3100 Master Notes

Resources

What do you need

- C compiler
- linker
- text editor

We will use the GNU toolchain

- Compiler: GCC (or CC)
- Linker: ld (invoked by GCC)
- Editor: vim, nano, emacs

Workflow

- Use text editor to edit source files
- Use compiler to generate ".o" files
- Use linker to link multiple '.o" files into executables


```
#include <stdio.h>

/* comments */
// single linek comments

int main(void)
{
        printf("Hello, world! \n");
        return 0;
}
```

Comments

Compiler ignores everything between "/* */"

The 'main' function

A special function that defines the entry point for the program

- This is where the OS transfers control once the program starts

printf

- C library function to print on the standard output for the process
- takes at least a string as argument
    - Between double quotations like **"This is a string"**
- **'\n'** is a new line character

Including header files

#include

- imports a header files with the specification of functions that exists in a library to be linked w/ the program

Compile

What is CC really doing?

- 3 steps
    - *preprocesses* hello.c
    - *compile* hello.c to hello.o
    - *links* hello.o /w **libc**
    - *writes* executable file **a.out**
- **You can and often will separate the 3 steps!**
    - No necessarily that often unless working with big files
    -

A second program

Purpose

- Read an integer from the standard input: **n**
- Compute the sum of all integers between **1 & n**
- Print the result on the standard output

New concepts

- Standard input and output

Summary

C program consists of functions

- main() is the entrance of a program

function consists of a sequence of statements

Variables can be declared in a function (like main)

- These are local variables
- Variables must be declared

Statements are terminated with ';'

- Empty statements are allowed
  ;;; empty statements

The ‘main function’ taking arguments

‘main’ function can take 2 arguments

- argc: the number of arguments
- argv: an array of arguments

If-else statements

Lecture 3

Mon. Sept. 4, 2019

Operators

- Conventional arithmetic, bitwise, and logical operators
  - + - * / %
  - & | ~ ^ << >>
  - && || !
- Pre/post increment/decrement (as in Java, C++< etc.)
  - i++ ++i j++ --j
  - c = i++; // c will (i - 1)
  - c = ++i; // c will be the same as “i”
- Simple & Compound assignment operators
  - more to come!

Precedence & Associativity

- Precedence determines which operation is done first
  - If operators have the same precedence, use associativity
  - use parentheses

i + j * 10 - k / 20
(i + (j * 10)) - (k / 20)

Assignment Operators

- Assignment Operator
  - LHS (Left Hand Side) is something that can be written to (e.g. to a variable)

LHS = Expression

  - LHS and Expression have “compatible” types

    - The value of Expression is assigned to LHGS & becomes the value of the assignment operation
- Compound Assignment operators (+=, *=, …)
    - var op= expr ⇔ var = var op expr
- ex.):
    - a = x + y; b = c = d = 0;
    - i += 10; // i = i + 10
    - j *= 5; // j = j * 5

Assignments **ARE NOT** statements

- assignments are **expressions** and **"=" is the operator**
    - you can chain them!
    - you can use them inside larger expressions

Integer Data Types

- char
- short int → short
- int
- long int → long
- long long int → long long

***C code can be confusing, people flex their skills in C, but it can be vary unreadable
"Don't be that guy flexing skills, just because you can do it in C" - Wei We**

- Consider x86_64 (64-bit architecture)

size (in bits)	signed	unsigned
8	char -128 … 127	unsigned char 0 … 255
16	short -32786 … 32767	unsigned short
32	int	unsigned int
32	long -2^{31} .. $2^{32} - 1$	unsigned long
64	long long	unsigned long long

- Consider x86_64 (64-bit architecture)

size (in bits)	signed	unsigned
8	char -128 … 127	unsigned char 0 … 255

16	short -32786 … 32767	unsigned short
32	int	unsigned int
32	long -2^{31} .. $2^{32}-1$	unsigned long
64	long long	unsigned long long

How much space?

- How to determine the amount of space for some type?
- Operator size of gives the # of bytes needed for a type of variable
    - You will need this later to dynamically allocate Space

Character (char) Data Type

- char has 8 bits (a byte)
- ASCII Code (American Standard Code for Information Interchange)
    - Characters are mapped to an integer in 0 … 127
    - An ASCII character can be stored in char
- Classes in ASCII
    - 0 … 31: “Control” character (a.k.a non printable)
    - 48 … 57: Digits
    - 64… 90: Upper case letters
    - 97… 122: Lower case letters

So…

- The character “h” is none other than … 72
    - char h1 = “H’, h2 = 72; // h1 & h2 have the same value
- Observe how…
    - ‘0’ through ‘9 are consecutive!
    - ‘A’ through ‘Z’ are consecutive!
    - ‘a’ through ‘z’ are consecutive!

char ch = ‘8’;
int x = ch - ‘0’ // what is the value of x?

Non-Printable Characters?

- These are sometimes useful
    - Showing the constant (literal)

‘\n’	newline

'\r'	carriage-return
'\f'	form-feed
'\t'	tabulation
'\b'	backspace
'\x7'	audible bell (x indicates hexadecimal)
'\07'	audible bell (0 indicates octal)

Basic Data Types: Floating Point

- A few floating point types
    - Consider x86_64 again

size (in bits)		

Automatic Type Conversion

- When an operator has operands of different types, the operands are **automatically** converted to a common type by the compiler
    - In general, a lower rank operand is converted into the type of the higher rank one, where
        - *char* < *short* < *Int* < *long* < *long long* < *float* < *double* < *long double*
        - **long double is of highest rank**
    - E.g. "1" gets converted to double before performing the addition in the expression "1 + 2.5"
- Automatic conversion can also occur across assignments
    - The value of the expression on the right hand side may be widened to the type of the LHS, e.g., "double d = 1"
    - Or **narrowed** (possibly with information loss), e.g., "int i = 2.5";
        - Read book for more details!

Type Casting: Explicit Type Conversion

- Useful to convert an operand to another type before doing arithmetic
    - *(<Type>)<expression>*

- Ex.): integer or double?

int x = 10; **int y = 3;** **double z = x / y**	

What about Booleans?

- K&R and C89/C90 do not have a Boolean data type
  - 0 “means” FALSE and anything else “means” TRUE
  - Common to use int or char to store Boolean values and define convenience macros
- C99 introduced Boolean

Be Mindful…

- Sometimes the results may not be as expected!
  - What is the size (in bits) of each operand?
  - Are your operands signed or unsigned?
- Ex.):

unsigned int x = 3;
unsigned int y = 7;
unsigned int z = x - y;
	z holds the binary representation of -4 but reading it as an unsigned int yields a very different value!

_Bool b1;
char b2, b3;
int i = 256; // 0x100
 	
b1 = i;
b2 = i;
b3 = i != 0;
	b1 is 1 because i is not 0
	b2 is - because the lowest 8 bits in “i” are 0
	b3 is 1 because i is not 0
	
	Do you want b2 or b3?

Lecture 4

Mon. Sept. 9, 2019

Flow of Control

- Statements are normally executed sequentiallu
- For selective or repeated execution we have all the suual suspects from Java / C++ / Python
  - blocks
  - if & if-else
  - while

- for
- switch
- break
- contine

Blocks (compound statements)

- List of statements enclosed by { and }
  - considered as a single statement
  - No semicolon after closing }
  - can be empty
  - can be nested (block in block)
  - useful for branching/ loop statements
  - can define variables at beginning of blocks
    - can mix declarations and code in c99

Comparison & Logical Operators

- Comparison operators that compare two expressions
  - Pay attention to types

== *!=* > < >= <=

- Logical Operators

&& **||** ***!***

- The result is either 0 or 1 (of int type)
  - 0 = false, 1 = true

Branching: if & else

- “exp” is typically a comparison or logical expression, but can be ANT expression (float/double, pointer, …)
- The statements can be compound statements (blocks)
- Or other if statements!
  - Beware of the dangling else (“else” matches the nearest preceding “if”, use blocks to disambiguate)

***Lot of things are “passable” in C and can compile and run fine, but may not be readable whatsoever by other programmers. Beware of this when using branching - Wei Wei

Ex.):

```
int i, j, min;
if (i < j)
        min = i;
```

Ternary Operator

- Takes **3** expression as operands

  exp ? exp2 : exp3

  - exp1 is evaluated first
  - If exp1 is non-zero (true), exp2 is evaluated and its value is used as the value of the ternary expression
  - If exp1 is zero (false), exp3 is evaluated and its value is used as the value of the ternary expressions
- **ex.):**

  min = i < j ? i : j

While Loop

- Very similar to python,
- ex.): computing sum of 0 .. 99

```
int i = 0, sum = 0
while (i < 100){
        sum = sum + i
        i++;
}
// Same as
while (i < 100) sum += i++;
```

Do-While Loop

- Checks condition after executing loop body
  - the statement is executed at least once
- ex.): computing sum of 0 .. 99

```
int i = 0, sum - 0;
do{
        sum = sum + 1;
        i++;
} while (i < 100);
```

For Loop

- Sometimes called “counting” loop
  - more like swiss-army knife!
- Three expression:
  - initialization, condition, increment
- Equivalent to

```
exp1;
while (exp2){
        <stmt>;
        exp3;
}
```

- 4 ways to use for-loops to computing the sum 0..99

```
sum = 0;
for (i = 0; i < 100; i++) sum = sum +i;
```

2nd way - w/ all initializations inside

```
for (sum = i = 0; i < 100; i++) sum += i;
```

3rd Way - Empty Body

```
for (sum = i = 0; i < 100: sum += i++);
```

4th Way - Comma Operator

```
for (sum = 0, i = 0; i < 100; sum += i, i++);
```

Comma Operator

- Takes 2 expressions

  exp1, exp2

  - exp1 is evaluated first, then exp2 is evaluated
  - exp2 is the result of the whole expression
- Has the lowest precedence
- Associated from left to right

  exp1, exp2, exp3 <-> (exp1, exp2), exp3)

  - Order can make a difference, e.g.,

    for (sum = 0, i 0; i < 100; sum += i, i++);
  - is not the same as

    for (sum = 0, i = 0; i < 100; i++, sum += i);

Multiway branching using “else if”

- if statement can contain multiple else statements

Switch example

```
// Assume all variables are defined as int
```

```
switch(i){
        case 0;
                n0 ++; break; // Note the break statement
        case 1;
        case 2;
```

Break Statement

- Most commonly used in switch statements
    - Prevents “fall-through” to the next case
- Also works in loops (for, while, do-while)
    - Loop execution terminated immediately, control resumes at statement immediately following the loop

Continue Statement

- Skip the rest of the current loop iteration and continue to the next one
- Can be used within for, while, and do-while loops
    - Can appear in a nested if / else
    - If used in nested loops, it applies to the “innermost” enclosing loop
    - For “for” loops, go to the evaluation of the “increment” expression

```
{ // begin loop body
        ...
continue;
        …
} // end loop body
```

Lecture 8

Weds. Sept. 11, 2019

Function Definitions

- No nesting (cannot define functions in a function)
- Return type can be “void” (no return value expected)
    - if missing, compiler assumes int
- return statements

```
return;          // terminates execution and return control to caller
return expr;     // terminate and pass value of expr back to caller
```

- Execution also terminates if end of function body reached
    - return value undefined

Function Declarations (Prototypes)

- Functions can be defined in any order
    - declare a function before first use if definition comes later
    - function prototypes often placed in header files (and reused)

```
#include <stdio.h>
int fahrToCelsius(int);

int fahrToCelsius(int degF) {
        return 5 * (degF - 32) / 9;
}
```

Example: computing b^n

- Power function
    - returns an int
    - 2 parameters: base “b” and exponent “n”, both int
- Functions can declare local variables
    - parameters & local variables can only be accessed inside the function
    - Storage class “auto” by default i.e., discarded when function returns
- Parameters are passed by value
    - “n” is changed in the power function but “i” DOES NOT change

Static & Global Variables

- Static local variables
    - Not visible outside function but retain value across function calls
- Global variables
    - declared outside functions; retained for entire duration of program
    - Storage class “extern” by default
      → can be accessed from functions in other files
    - Static global variables
      → visible only in functions defined in the same file following variable declaration

Be Cautious with static and global variables

- “Nice” functions only depend on their inputs
- Static and global variables have “side-effects”
    - retain values across function calls
    - change the meaning of the function at each call
    - You can understand the function without holding all the code in your head
- unless you have really good reasons and really know what you are doing, DO NOT USE STATIC OR GLOBAL VARIABLES

Function call context

- Function call context includes
    - copies of function arguments (call-by-value_
    - auto local variables
    - return address
- Call contexts managed automatically using the **execution stack**
    - a stack frame is created automatically for each function call
    - the frame lasts for the duration of the call
    - discarded automatically when the function terminates
    - NOTHING in the frame survives the call

Lecture 9

Mon. Sept. 16, 2019

A C Primer (5): Arrays and Pointer Basics

Arrays

- New Data types
- Arrays represent a linear, contiguous collection of “things”
- Each “thing” in the array has the same fixed type
- ex.):
    - array of characters
    - array of integers
    - array of doubles…

Array example

```
int x[5]; // define an array of 5 int's
 // accessing array elements is similar to accessing list in Python
// the index starts from 0
x[0] = 1; x[1] = 2; x[2] = 3; x[3] = 4; x[4] = x[3] + 1;
// initialize array with a list
int y[5] = {1, 2, 3, 4, 5};
// Number of elements is optional if all elements are listed
int z[] = {1, 2, 3, 4, 5};
// Specify the value of first 2 elements. The rest are set to 0
```

```
int a[5] = {1, 2}; // C99. b will have 1, 2, 0, 0, 5.
int b[5] = {1, 2, [4] = 5};
```

Array In Memory

- Think about how array elementts are stored in memory
- Index starts from 0, the last one is 4 = (5 - 1)

String Initialization

- A string is a char array that ends with a 0 (null character)
    - Memory that stores 0 is part of the string
- It can be initialized with a list of characters
- or a string (double-quoted literal)

Arrays as Automatic Variables

- • You can declare arrays inside any function or block
    - Destroyed when exiting from the function or block
- Variable length arrays(VLA, C99) The size of your array can depend on function arguments or other known value

```
int foo(int n,int k) {
      int x[n]; // The value of n is known at this time
// Like other auto variables, x is kept on
// the stack and is NOT initialized

for (int i = 0; i < n; i++)
            x[i] = 0;
      ……
       return -1;
}
```

Array Assignment

- You CANNOT assign a whole array at once to another array
    - Even when the types match

Arrays and Functions

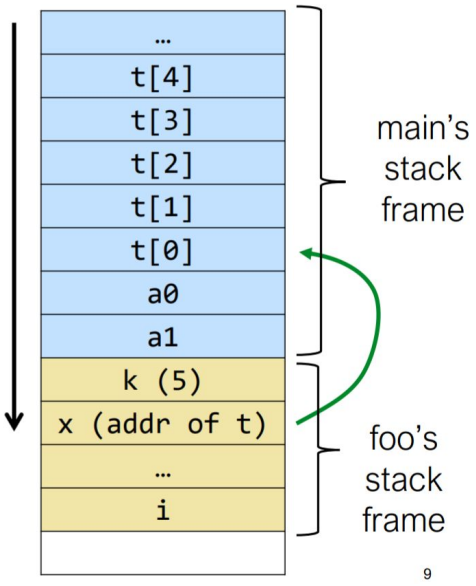
- Arrays can be passed to functions!
    - 1 BIG CAVEAT
- Calling convention in C
    - by value for everything
    - except arrays
- Arrays are always passed BY REFERENCE

- passed as “pointers” - we’ll look at pointers soon
- Functions cannot return arrays
  - No easy assignments

Array argument Example

- Address of “t” is passed to foo()
- Modifications to “x” are visible in main

```
int foo(int x[], int k) {
      for (int i = 0; i < k; i++)
            x[i] = i;
      return x[0];
}

int main() {
      int t[5];
      int a0 = foo(t, 5);
      int a1 = t[4];
      printf("%d %d\n" , a0, a1);
      // more code
}
```



Multidimensional arrays

```
// declaration and initialization
int h[2][3] = { {0, 1, 2}, {10, 11, 12} };
```

	0	1	2
0	0	1	2
1	10	11	12

Pointers

- Perhaps the scariest part of C
- Yet…
  - The most useful part of C!
- Pointer is simply
  - a value
  - denoting address of a memory cell

Variables and Memory

- The memory is an array of bytes
- Every byte in memory is numbere: the address!
  - An address is just an unsigned integer
- Every variable is kept in memory, and is associated with 2 number
  - The address
  - The value stored at that address

Implicit Address Use

- address are used implicitly by the compiler all the time

```
int foo (int v)
{
      int a, b;  // allocate storage space for a and b
      a = 1;           // store 1 to memory, at a's address
      b = a;           // load value from a's address, write to b's
address
      v = v + b;       // load value from v's address, add to value at
                  // b's address, and write result to v's address
      return v;
}
```

	Address	Value
	100012	
b	100008	
a	100004	1

Explicit Use: Pointers

- A pointer is a variable that holds an address of something
- Declaration

// declare p to be a pointer to an int

- the value of p is an address of an int, p itself has an address

Referencing and dereferencing

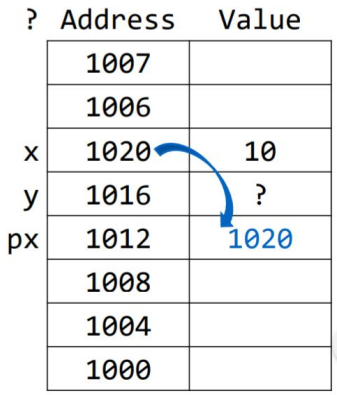
- Two new operators
  - & Reference - "get" the address of something
  - * Dereference - "use" the address

Ex.):

```
int x = 10, y;
// px is a pointer to int, i.e., the address of an integer
int *px;          // px itself has an address

px = &x;          // &x is the address of x. Save it to px

// *px: use px as an address to get the value at that location
y = *px;          // and save the value in y

// save 20 to location pointed to by px (use px as an address)
*px = 20;         // px has x's address, so x becomes 20
```

Picture It

```
// assume 32 bits in and address
int x = 10, y;
int *px;
px = &x;
y = *px;
*px = 20;
```



Revisit the Example

```
int foo(int x[], int k) {
    for (int i = 0; i < k; i++)
        x[i] = i;
    return x[0];
}
```

x is the starting address af an array, which is the same as the address of element 0.

```
// x is the address of an int
int foo(int * x, int k) {
    for (int i = 0; i < k; i++)
        x[i] = i; // use the pointer as an array
    return x[0];
}
```

Array Question 1

```
// how many integers are in a, and in b?
int a[] = {1, 2, 3, 4};
int b[4] = {1, 2, 3};
```

Both have 4 integers

Array Question 2

```
// how many bytes (characters) in c? and in d?
char c[] = {'a', 'b', 'c', 'd'};
char d[] = "abcd";
```

in “c” is 4, in “d” is 1

Array Question 3

```
int a[10];
If a[0]'s address is 1000, what is a[4]'s address?
```

1016

Array Question 4

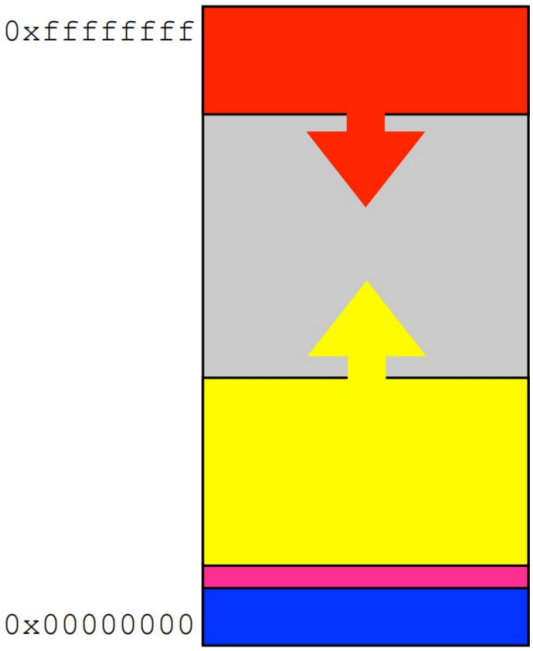
```
char t[10][20];
If t[0][0]'s address is 1000, what is t[1][1]'s address?
```

1041

Lecture 10 - C6: Dynamic Memory Allocation

Weds. Sept. 18, 2019

Recall the memory model…



- Three pools of memory
  - Static/global
  - Strack
  - Heap
- Each pool features
  - Different lifetime
  - Different allocation/deallocation policy

Static/global memory pool

- This is where…
  - All constant (including string literals) are held
  - Global variables
  - All variables declared
- Allocated when - the program starts
- Deallocated when - the program terminates
- FIXED size - compiler needs to know the size & make reservations

Stack

- This is where…
  - Memory comes from local variables in functions!
- Easy to manage because it is automatic!

- Allocated automatically when entering the function
- De-allocated automatically when you leave the func.
  - Scope is that of func.
  - Should not be used after the fuc. returns
    - For ex): indirectly used via a pointer ← DO NOT DO

default stack size is 2 MBs
need to increase stack size for large arrays and deep recursion

Heap

- This is where…
  - Memory comes from **manual** “on-the-fly” allocations
- Who is in charge?
  - **The programmer** for both allocation / deallocation
- Lifetime of memory blocks?
  - As long as they are not freed

Requesting memory on the heap

- `#include <stdlib.h>`
- `void* malloc(size_t size);`
  - size_t is an unsigned integer data type defined in <stdlib.h>
  - used to represent sizes of objects in bytes
- If successful, a call to malloc(n) returns a **generic pointer (void *)**
  - It points to a memory block of **“n”** bytes on the heap
- If not successful, NULL is returned

```
char* p = malloc(100); // request 100 bytes
```

Generic pointers: void*

- Pointer to a memory block whose content is “untyped”
- use for raw memory operations or in generic funcs.
- Automatic casting when assigned to other pointer types
  - `int * pox = malloc(6 * sizeof(int));`
- Requires casting before dereferencing for read / write
  - `*(int *)pv; // use pv as an int *`
- NULL, a special pointer value useful for initializations, error handling
  - `#define NULL ((void*) 0)`

Can’t always get what you want

- A call to malloc() may fail
  - for ex. when out of memory
- In this case you get back a NULL value

- Not much to do except report the error (and terminate nicely)

- Idiom

```
char*p = malloc(100); // request 100 bytes
if (p == NULL) {
      // report error and finihs
      perror("Not enough memory");
      exit(1);
}
```

How much Space?

- You need to tell malloc() how many bytes you need
- To request space for an array, need
  - # of elements
  - amount of space for each array element
    - sizeof(t) returns # of bytes needed for a value for type T
- ex.):

```
int* pox = malloc(6 * sizeof(int)); // request space for 6 ints
if (pox == NULL)
        report error and finish;
```

Another Way

- `#include <stdlib.h>`
- `void* calloc(size_t, nmemb, size_t size);`
- calloc() is implemented in terms of malloc()
  - calloc() also initializes the content to 0

```
int* pox = calloc(6, sizeof(int)); // request space for 6 ints
if (pox == NULL)
      report error and finish;
```

Adjusting the Size

- `#include <stdlib.h>`
- `void * realloc(void* ptr, size_t size);`
- What if you change your mind?
  - you requested 100 bytes, but now need 200

```
char* p = malloc(100); // request 100 bytes
…
p = realloc(p, 200); // p may change!
```

- Before a call to realloc(p, size), p must be
  - A pointer returned by a previous malloc/calloc/realloc

- or NULL, in which case the call is equivalent to malloc(size)

Deallocation

- `#include <stdlib.h>`
- `void free(void *ptr);`
- Straightforward
  - simply call the lib. func. “free”
  - takes a pointer to the block to free

```
free(ptr);
```

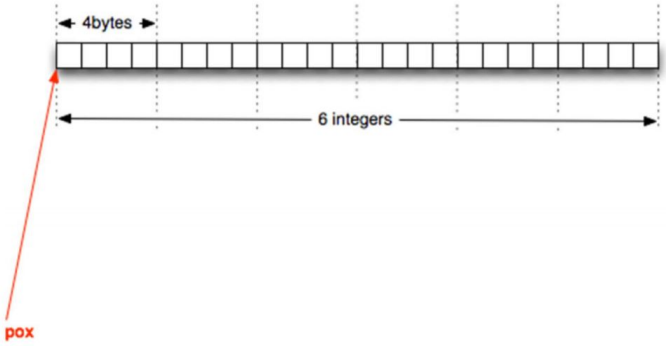
- Do not free a pointer twice!
  - After freeing a pointer, set it to NULL!

Two key rules

- Rule 1: Everything you request should be freed, eventually
- Rule 2: Only free what is allocated via malloc/calloc/realloc/
- Consequences of not following the rules
  - Memory “leaks”
    - Your program will eventually run out of memory
  - Undefined behavior and horrible crashes
    - Freeing unallocated mem. or already freed mem.
    - May cause a memory error and program crash
    - Wrose, may corrupt heap and cause crash later
    - Even worse, program may keep running, totally corrupting your data, without you knowing

Pointers and arrays

- `after pox = malloc(6 * sizeof(int))`



- That looks like an array
  - and you can use pox as if it is

Example: Use pointer as array

```
// programmers have to manage memory themselves before C99
void doSomething(int n){
     int * pox;
     pox = malloc(sizeof(int)*);     // request mem from heap…
     *pox = 0;       // set the int at the address pox to 0
     pox[0] = 0;     // same thing
```

```
    pox[1];          // the int after pox[0]
    // more lines...
    free(pox);       // remember to free!
}
```

Array of pointers

- Pointers, like integers, can be placed in an array

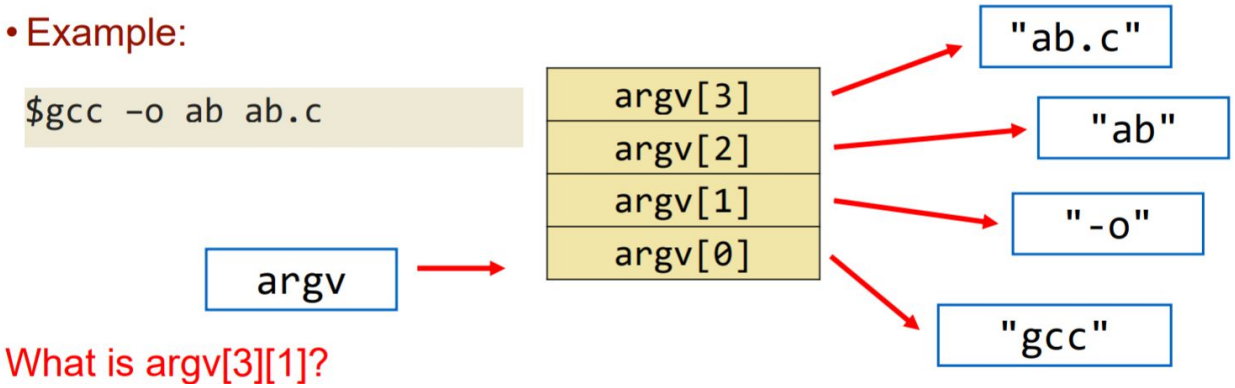
```
int a0;
int a1;
int a2;


char *p0 = malloc(10);
char *p1 = malloc(10);
char *p2 = malloc(10);
```

```
// array of int's
int a[3];

// array of pointers
char * p[3];

// Can also do with a loop
p[0] = malloc(10);
p[1] = malloc(10);
p[2] = malloc(10);
```

Example: Command line argument

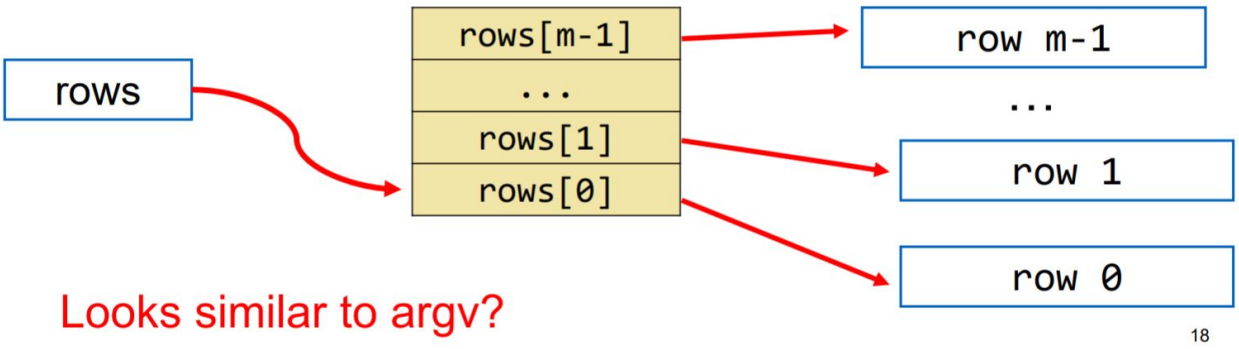
```
int main (int argc, char *argv[]);
```

- argc: the # of arguments on the command line
- argv: array of pointers to characters
  - the # of elements is argc
  - each element in an array points to null-terminated strings



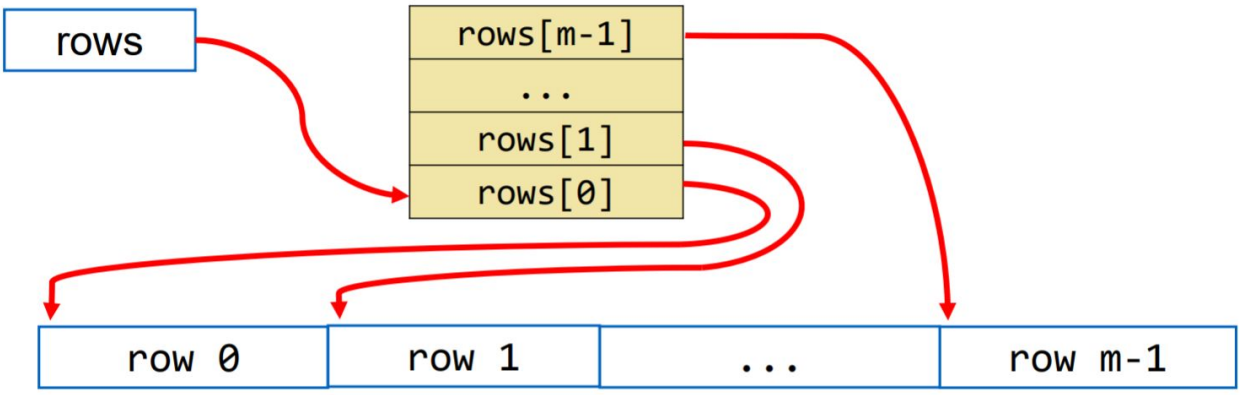
Example: allocating 2d dynamical array

```
void doSomething(int m, int n)
{ int **rows;
  rows = malloc(sizeof(int *) * m);  // array of pointers
  for (int i = 0; i < m; i++)
    rows[i] = malloc(sizeof(int)*n); // one int array for each row
  ...
  for (int i = 0; i < m; i++)
    free(rows[i]);
  free(rows);
}
```



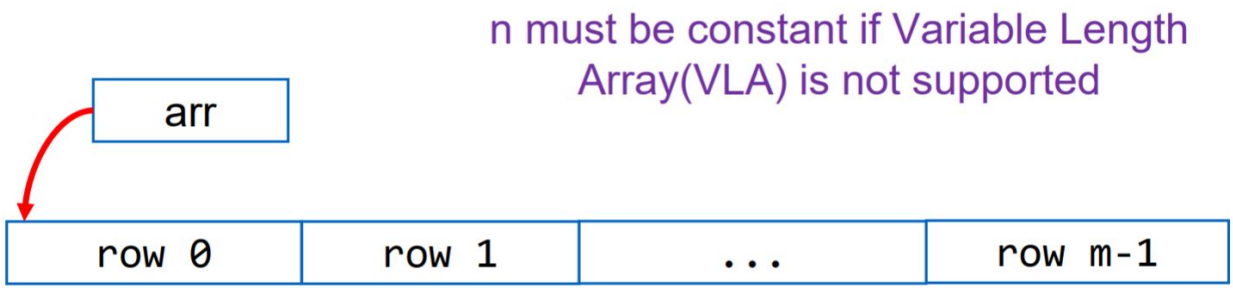
Example: allocating 2d dynamical array (take 2)

- Requesting all the memory space needed by data with one malloc() all
  - Instead of call malloc() m times, for each row
- Calculate rows[1], rows[2], and so on (pointer arithmetic! next lecture)
  - You can even calculate the address of each element (e.g. rows[10][5])



Example: allocating 2d dynamical array (take 3)

```
void doSomething(int m, int n)
{     int(* arr)[n];  // arr is a pointer to an array of n int's
      arr = malloc(m * n * sizeof(int));    // one malloc() for data
to access
      arr[1][2] = 1;
      arr[2][3] = arr[1][2] + 10;
      …
      free(arr);  // free memory
}
```



Pointers taking the address of…

- A static?
  - The address is never going to go "bad"
  - The static lives as long as the program!
- A stack [automatic] variable?
  - The address is valid as long as the variable!
  - When the function returns… The address is bogus
- A heap variable?
  - The address is valid as long as the variable is!
  - The variable disappears when explicitly de-allocated (freed)

Lecture 11 - C7: Pointer Arithmetic & Structures

Mon. Sept. 23, 2019

Pointers are addresses

- The value of a pointer is a byte address
  - Unsigned integer used to number bytes in memory
  - Range is between
    - 0x00000000 and 0xFFFFFFFF [32-bit]
    - 0x0000000000000000 and 0xFFFFFFFFFFFFFFFF [64-bit]
- Corollary
  - If a pointer is an integer, you can do arithmetic
  - To computer addresses

Pointer Addition Example

- Suppose p is a pointer to an int, and its value is 10000
- p + 1 is not the next byte address
- It is the address of next item (of type int)

Adding a pointer and an integer

- Add an integer to a pointer, the result is a pointer of the same type
  - It is dif. from reg. integer addition
  - The integer is **automatically scaled** by the size of the type pointed to
- Suppose p is a pointer to type T, and k is an integer
  - Then both “p + k” and “k + p”
  - Are valid expressions that evaluate to a pointer to type T
  - Have a byte address equal to

```
(unsigned long)(address stored in p) + k * sizeof(T)
```

C standard does not allow arithmetic on void *

gcc has an extension, treating sizeof(void) as 1

Address	Value		To access values
1020		p + 5	*(p+5) OR p[5]
1016		p + 4	*(p+4) OR p[4]
1012		p + 3	*(p+3) OR p[3]
1008		p + 2	*(p+2) OR p[2]
1004		p + 1	*(p+1) OR p[1]
1000		p = 1000	*p OR p[0]
996		p - 1	*(p-1) OR p[-1]
992		p - 2	*(p-2) OR p[-2]

Pointers Subtraction

- Subtract one pointer from another: both must have the same type
- The result is the number of data items between the two pointers!
    - Not the number of bytes

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

int main() {
   int *p = malloc(sizeof(int)*10);
   int *last = p + 9;
   int  dist = last - p; // both are int *
   printf("Distance is %d\n",dist);
  free(p);
  return 0;
}
```

Output

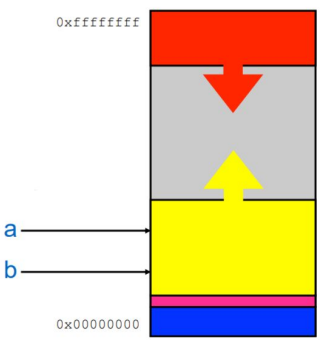
```
$ gcc ptrsub.c
$ ./a.out
Distance is 9
$
```

Pointer comparison

- You can also compare two pointers
- Purpose
    - Check boundary conditions in arrays
    - Manually manage memory blocks
- Semantics
    - Simply based on memory layout!
    - Compare bits in pointers as unsigned integers!

Check if you are done:

```c
while (b < a) {
    // do something
}
```



Effects of casting types?

- If you cast a pointer type…
  - Any subsequent pointer arithmetic will use the type you choose
- Do not wanting scaling? Casting a pointer to (char *)
  - Because sizeof(char) is 1

```
int * t;

char * p = (char *) t + 8;   // 8 is not scaled

char * q = (char *) (t + 8); // 8 is scaled
```

Arrays and pointers

- Arrays and pointers can often be used interchangeably

```
int  a[10];
int *p = a;

// all of the following evaluate to the value of a[0]
*p              p[0]        *a          a[0]

// all of the following evaluate to the value of a[1]
*(p+1)          p[1]        *(a+1)      a[1]

// all of the following evaluate to the address of a[0]
// type is int *
p               &p[0]       a           &a[0]
```

"array of int" becomes "pointer to int" (array decay)

Example: arrays and pointers

- Equivalent ways of initializing an array

```c
int a[10], *p = a;  // not *p = a; it is int *p; p = a;

for(int i=0; i<10; i++) a[i] = i;  // array indexing

for(int i=0; i<10; i++) p[i] = i;  // indexing via pointer

for(int i=0; i<10; i++) *(p+i) = i; // explicit pointer arithmetic

for(i=0; i<10; i++) i[p] = i;   // obfuscated but valid C!

for(i=0; i<10; i++) *p++ = i;   // common pointer use idiom
```

```c
int * tp = p;
p ++;
*tp = i;
```

Arrays and pointers are NOT the same

```c
int   a[10];
int   *p = a; // a is converted to int *

// a is still an array after &
&a   // pointer to array of 10 int's  int (*)[10]
&p   // pointer to a pointer to int   int **

// a is still an array after sizeof
sizeof(a) // 40 because a is an array of 10 int's
sizeof(p) // 8  because p is a pointer

p++;   // can increment p
a++;   // cannot increment a; this will not compile
       // Similar to n++ vs 2++
```

Example: Arrays and pointers are NOT the same

```c
#include <stdio.h>

void foo(int *x)
{
   printf("%lu\n", sizeof(x));
}

int main()
{
   int a[10], *p;

   p = a;  // a is converted to int *
   // a is still an array in sizeof
   printf("%lu %lu\n", sizeof(a), sizeof(p));
   foo(a); // a is converted to int *
}
```

Output

```
% gcc array.c
% ./a.out
40 8
8
```

Structures

- Mechanism to define new types
  - Also known as “compound types”
  - Use to aggregate related variables of different types
- Structures
  - Structures can have a type name
  - Can have “members” of any types
    - Basic types
    - Pointer
    - Arrays
    - Other structures
- Structure variable definition
  - Specifies variable name
  -

```c
struct student_grade {
    char *name;
    int  id;
    char grade[3];
};
…
struct student_grade student1;
struct student_grade
    student2, student3;
struct student_grade all[200];
```

Structure example

```c
struct Person {
   int     age;
   char gender;
};

int main(){
   struct Person p;

   p.age = 44;
   p.gender = 'M';

   struct Person q = {44, 'M'};

   return 0;
}
```

Structure *type declaration*

Structure *variable definition*

Syntax for field access similar to Java and Python

Structure *variable definition* and *initialization*

Example: Array of Structures

- Member name is a char array
- Cabeats
  - Names cannot be more than 31 characters long
  - Four person in family
  - Indexed 0..3
- Array of structures for the whole family
- Nested initializers

```c
#include <stdlib.h>

struct Person {
   char   name[32];
   int    age;
   char   gender;
};

int main()
{
   struct Person family[4] = {
      {"Alice",34,'F'},
      {"Bob",40,'M'},
      {"Charles",15,'M'},
      {"David",13,'M'}
   };
   int juniorAge = family[3].age;
   return 0;
}
```

typedef

- Struct. names can be long
- C provide the ability to define type abbreviations
  - typdef declaration
    - Give existing type new type name
- Make code more readable
- Structure and typedef declarations often combined

```c
struct Person {
   char   name[32];
   int    age;
   char   gender;
};
typedef struct Person TPerson;
int main()
{
    TPerson family[4];
    ...
    return 0;
}
```

Operations on struct

- Assignment
  - All struct members copied
- Can be passed to funcs.
- If pass by value, cannot change members in funcs.
- Passing or returning large structures can be costly

```c
typedef struct Person {
   char   name[32];
   int    age;
   char   gender;
} TPerson;
```

***use pointers to structures!

```
TPerson a, b, c;

…
a = b;
c = searchPerson("name");
```

Pass Structure by reference

```
typedef struct Person{
      char  name[32];
      int   age;
      char  gender;
}

TPerson * init_Person(TPerson *p, char * name, int age, char
gender){
      strcpy(p->name, name);      // (*p).name
      p->age = age;               // (*p).age
      p->gender = gender;         // (*p).gender
      return p;
}
// Study the demo code is in the demo repo
// especially the lines that copy name
```

Structure Alignment

- Structure members are aligned for the natural types

Alignment requirements on x64 architecture	
char	1
short	2
int	4
long	8
float	4
double	8

```
struct struct_random {
    char    x[5]; // bytes 0-4
    int     y;    // bytes 8-11
    double  z;    // bytes 16-23
    char    c;    // byte 24
};     // Total size 32
```

```
struct struct_sorted {
    double  z;    // bytes 0 - 7
    int     y;    // bytes 8 - 11
    char    x[5]; // bytes 12 - 16
    char    c;    // byte  17
};      // Total size 24
```

Arrays and pointers

- Copying arrays

```
// using array indexing
void copy_array0(int source[], int target[], int n)
{
  for(int i = 0; i < n; i++)
    target[i] = source[i];
}

// using pointers
void copy_array1(int source[], int target[], int n)
{
  for(int i = 0; i < n; i++)
    *target++ = *source++;
}
```

```
int * tp = source;
source ++;
*target = *tp;
target ++;
```

Typecasting Pointers

- C lets you cast pointers in any way you like
- You can “forge” pointers to point wherever you wish…

```
float f;
char p = * (char *) &f;          // 1st byte representing f
char ch = * (char *) 1000; // access any byte in memory
```

- THAT’S WHAT MAKES C VERY ATTRACTIVE FOR LOW-LEVEL PROGRAMMING
- THAT IS ALSO VERY POWERFUL AND THUS DANGEROUS

Returning more than one value from functions

- Use references (caller prepares the storage)

```
long int strtol (const char* str, char** endptr, int base);
```

- Use arrays (caller prepares the storage)

```
int pipe(int pipefd[2]);
```

- Return a structure (costly if structure is large)
- Return a pointer (to array or structure)
    - Must be dynamically allocated or static. RTM (read the manual)!

```
char strdup(const char *str1);
```

```
struct tm *localtime( const time_t *time );
```

- Use global variables (DON'T)!

```
errno
```

Typedef

```
// Example of typedef. Think about how you would define a variable
typedef    int   BOOL;
typedef    char  name_t[100];
typedef    char  *Pointer;
```

Self-referential structures

```
struct Person{
      int   age;
      char  gender;
      char  name[32];
      struct     Person * parents;     // A pointer to this type of
struct
}     person1, person2;          // Can define variables here
```

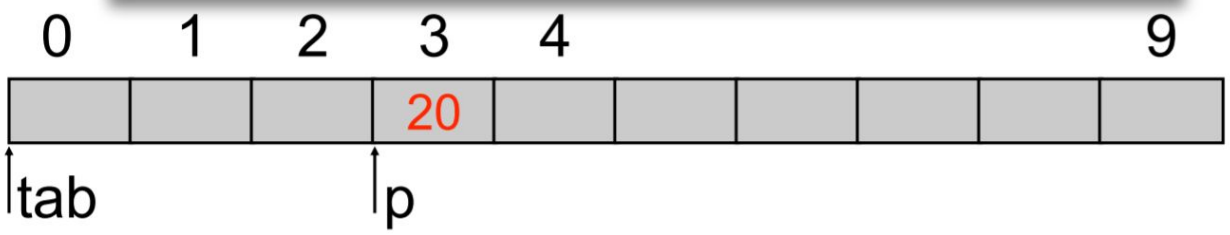
Self-referential structures - 2

```
struct student{
      char  name[128];
      // Can have a pointer to a struct defined later.
      // However, you cannot define an array of book here (e.g.
books[8])
      struct book * books;
};
struct book {
      char  title[128];
      struct student * owner;l
      struct book * next;   // A pointer to this type of struct
};
```

Example of Pointer Arithmetic

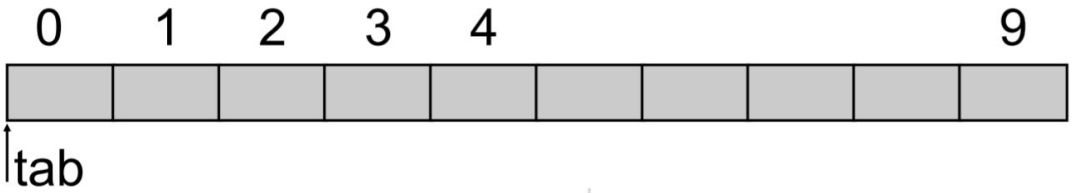
- Simple illustration

```c
#include <stdlib.h>

int main()
{
   int *tab = (int*)malloc(sizeof(int)*10);
   tab[3]   = 10;
   int *p   = tab + 3;
   printf("What is at tab+3? = %d\n",*p);
   *p = 20;
   printf("What is at tab[3]? = %d\n",tab[3]);
   return 0;
}
```



But what about memory addresses?

- Same story…!



0x11110000

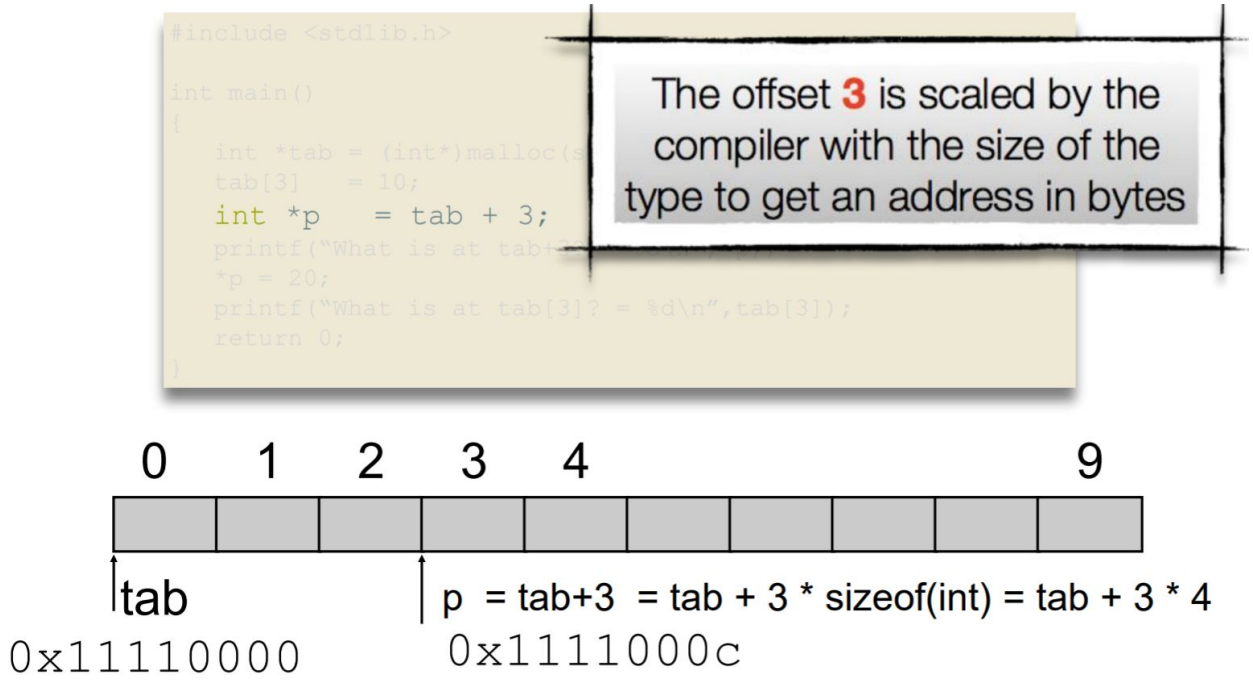
tab+1 = ?

~~0x11110001 ?~~

0x11110004 ?

WHY?
Simply because tab is a pointer to an int and an int is 4-bytes wide!

Bottom Line



Memory Alignment Requirements

- Memory used to store a value of type X MUST
  - be lined-up on a multiple of natural alignment for X
- Why?
  - Performance!
- If you do not respect alignment requirements…
  - BUS ERROR (sigbus)
  - The O.S. will kill your program

Good News and Bad News

- The C compiler handles alignment 99% of the time
- Programmers have to handle the rest - when you do pointer arithmetic of course!
  - Do not assume the location of the fields
- When you call sys. routines w/ specific alignment needs
  - Your arguments must comply
  - Use compiler annotations to force specific alignment (beyond our scope, simply remember that this exists!)

Example

```c
#include <stdio.h>
struct Person {
   char   name[32];
   int    age;
   char   gender;
};
typedef struct Person TPerson;

TPerson init(char name[], int age, char gender) {
  TPerson p;
  int i;
  for(i = 0; name[i]>0; i++)
     p.name[i] = name[i];
  p.name[i] = '\0';
  p.age = age;
  p.gender = gender;
  return p;
}

void print_info(TPerson p) {
  printf("name: %s, age: %d, gender: %c\n",
         p.name, p.age, p.gender);
}
```

```c
int main() {
    TPerson family[4];
    family[0] = init("Alice",34,'F');
    family[1] = init("Bob",40,'M');
    family[2] = init("Charles",15,'M');
    family[3] = init("David",13,'M');
    print_info(family[0]);
    print_info(family[1]);
    print_info(family[2]);
    print_info(family[3]);
    family[1] = family[0];
    print_info(family[1]);
    return 0;
}
```

Output

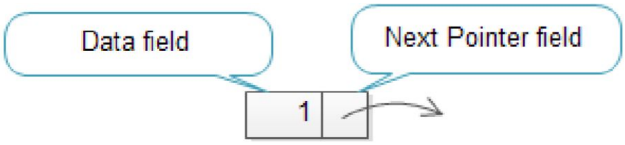
```
./a.out
name: Alice, age: 34, gender: F
name: Bob, age: 40, gender: M
name: Charles, age: 15, gender: M
name: David, age: 13, gender: M
name: Alice, age: 34, gender: F
```

Lecture 12 - C8: Linked Lists, Enums, Func. Ptrs.

Weds. Sept. 25, 2019

Example: Linked List

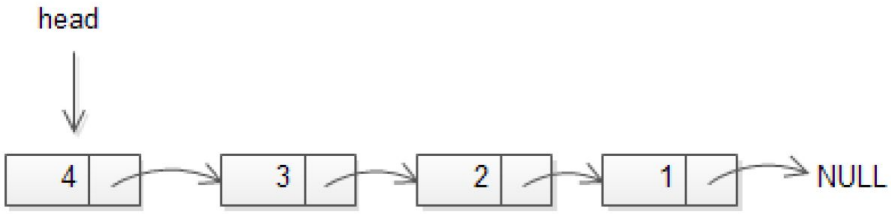
```c
// a data structure that consists of a chain of nodes
// Starting from head, a node has a reference to the next node
typedef struct node_tag{
      int   v;                // data
      struct node_tag * next;     // A pointer to this type of struct
} node;                           // Define a type. Easier to use.
```



Head

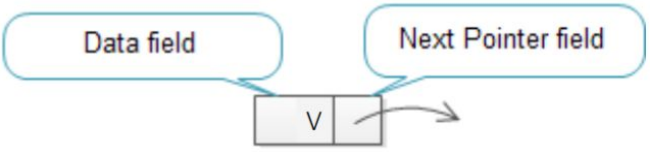
```
node * head;          // head is a pointer, not a node!
head = NULL;          // at beginning, it is empty
```

- After adding nodes into the list,



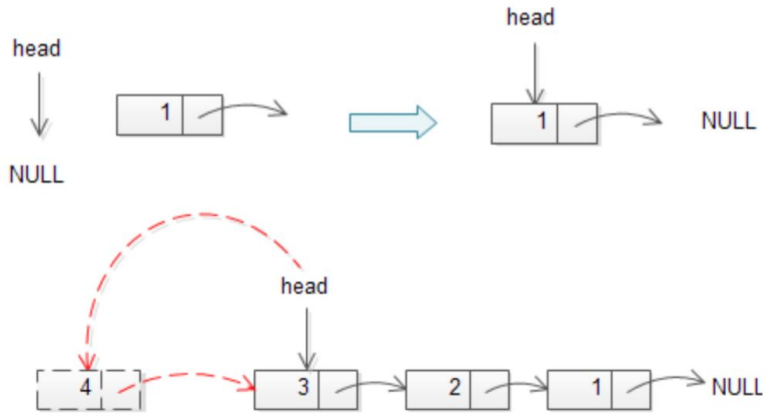
Create a node

```
node* new_node(int v) // create a node for value v
{
      node * p = malloc(sizeof(node)); // allocate memory
      assert(p != NULL);    // you can be nicer

      // Set the value in the node.
      p->v = v;  // you could do (*p).v
      p->next = NULL;
      return p;  // return
}
// is it similar to creating objects using "new"?
```



Prepend

```
node * prepend(node * head, node * newnode)
{
      // how?
}
```



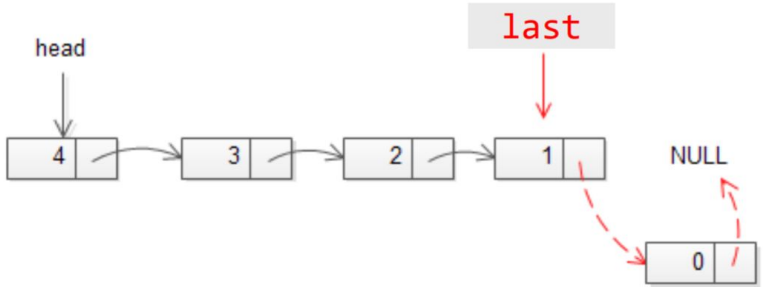
- cannot perform the following or else the head will point to the 1st node, and the 1st node points back to the head

Find the last one

```
node * find_last(node * head)
{
      if (head != NULL);
}
```

Append

```
node * append(node * head, node * newnode)
{
      node * last - find_last(head);   // find the last one
      if(last == NULL)      // if the list is empty, new node is the
head
            return newnode;
      last -> next = newnode;
      newnode -> next = NULL;
      return head;                  // return the (unchanged) head
}
```



Enumeration types (ABC 7.5)

- User-defined integer-like types:
- Names look like C identifiers
    - are listed (enumerated) in definition
    - treated as integer

Enumeration types

```
// enum start from 0 by default
enum week {Sun, Mon, Tue, Wed, Thur, Fri, Sat};
enum week dow = Mon;

// But can be initialized; Warning is 2, Error is 3, etc.
enum status {OK = 1, Warning, Error, Fatal};
```

Type qualifier: const

```
// constant int
const int  a = 10;          // cannot change a
// a pointer to constant int
const int  *pa = &a;  // can change pa, but not *pa
// a constant pointer to an int
int * const pb = &b         // can change *pb, but not pb
// constant pointer to a constant int
const int * cont pc = &a;   // cannot change *pc or pc

// cannot change the source string
char * strcpy(char * dest, const char* src);
```

Function pointers

```
/* function returning */
int func();
/* function returning to integer */
int * func();
/* pointer to function returning integer */
int (*func)();
/* pointer to function returning pointer to int */
int * (*func)();
```

Pointer to function example

```
int   mymax(int a,int b)
{
      return (a > b) ? a : b
}
// a pointer to function
int (*pf)(int a, int b);

// assign a value to the pointer
pf = mymax;            // C99 style. Note that it is not mymax()
pf(3, 5);
pf = & mymax;
(*pf)(3,5);
```

Use of function pointers

- Call-back mechanism
  - Generic functions (ex. coming next)
  - pthread_create()
  - Dynamic signal handlers, ….
- You can store function pointers in arrays
  - And arrays stored in structures!
  - And you can simulate objects in Object Oriented Languages!

Example: quicksort in C library

- The prototype (in <stdlib.h>)

```
void qsort(void * base
           size_t nel,
           size_t width,
           int (*compare)(const void *, const void *));
```

- qsort takes…
  - base:
  - nel:
  - width:
  - compare: a pointer to a function that compares two values

****sort only knows*

Why passing a function to qsort?

- Need to tell qsort() how to compare items in the array

- we have a generic quickSort implementation
  - Do not want to implement one for each type of data
- The qsort() implementation calls the comparator to rank elements

```
int (*comprae)(const void *a, const voi *b);
```

- the function takes the address of 2 items to be compared,
- and returns:
  - 0 if *a EQUALS *b
  - a positive value if *a is GREATER THAN *b
  - A negative value if *a is LESS THAN *b

Example of compare() function

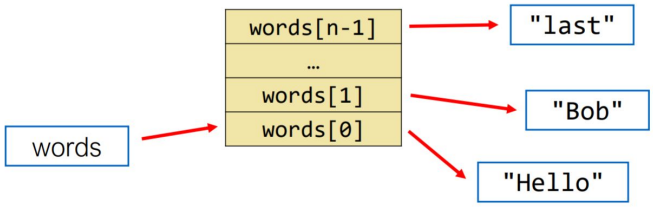
- when qsort() compares items, it provides the address of two elements to be compared.

```
int compare_int(const void *a, const void *b)
{     // qsort( does not know the type, but you know
      return *(int * a) a - *(int *)b;
}
int compare_double(const void (a, constn void *b)
{     // qsort( does not know the type, but you know
      return
}
```

Example: sort array of strings

- Example are pointers to strings
  - Need to compare string, instead of pointer

```
int compare_string(const void *a, const void *b){
// how to compare *a and *b?
// for example, a is &words[0] and b is &word[1]
```



Compare string pointers

- An element in array words is (char *).
- a is the address of an element of type (char *). So, a's type is (char*)*

```c
int compare_string(const void * a,const void * b)
{
   char *s1, *s2;
   s1 = *(char **)a;     s2 = *(char **)b;

   return strcmp(s1, s2);  // use library function to compare
}

// or on one line
int compare_string(const void * a,const void * b)
{
   return strcmp(*(char**)a,*(char**)b);
}
```

Calling quicksort()

- See complete demo code in the demo repo

```c
int compare_string(const void* a,const void* b)
{
   return strcmp(*(char**)a,*(char**)b);
}

int some_function(void)
{
   …
   char** words = malloc(sizeof(char*)*n);
   …
   qsort(words,n,sizeof(char*),compare_string);
   …
}
```

Type casting to char ** before dereferencing

Lecture 13 C9: I/O and Files

Fri. Sept. 27, 2019

errno

- Most C library functions can “fail”
    - When they do, they return a flag reporting failure… (-1)
    - Some set of global variable to report the exact error code

errno

// to use errno, include <errno.h>

- Check manual page to interpret the error code
- Print a more descriptive message with perror()
- Avoid functions that set errno in multithreaded code
    - Prefer thread-safe version when available

```
void perror(const char *str);
```

Files and directories

- A file is an object that stores information, data, etc.
-
- Example: files you create with an editor (.c, .h, Makefile, readme, etc.) executable generated by the compiler, and gcc itself other devices, like screen, keyboard, …
- In Linux, files are organized in directories
    - A directory can have subdirectories and files
    - The top directory is /
- A path specifies the location of file/directory in the file system

/home/john

- In Unix/Linux, everything is a file

The stdio library

#include <stdio.h>

- Declares FILE type and function prototypes
    - FILE is an opaque type (system dependent) for operating on files
        - It is a structure, but do not try to change it directly!
    - Use library functions to access FILE objects, via pointers (FILE *)
- Defines “standard” streams stdin, stdout, stderr
    - Created automatically when program starts
    - They are files!
- The library is linked automatically by the compiler

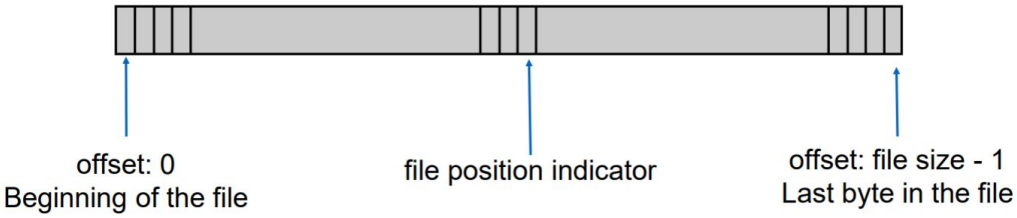
Files and I/O API

- In C, a file is simply a sequential stream of bytes
- The “f” family of functions (fopen, fclose, fread, fgetc, fscanf, fprintf,…) are C library functions to operate on files

- All these use a FILE* abstraction to represent a file
- The C library provides buffering
  - That's why sometimes you do not see output of printf immediately We will learn another set of functions provided by OS

File as stream of bytes

- Before use a file must be “open”
  - This sets a position indicator for reading and/or writing
- Each read/write starts from current position, and moves the indicator
  - Writing after last byte increases the file size
- Position indicator can also be changed with fseek
- All open files are closed when program ends
  - Good practice to close explicitly when no longer needed



Opening Streams

FILE *fopen(const char *filename, const char *mode)

- Open the file filename in mode as a stream of bytes
- Returns a pointer to FILE (FILE *) or NULL (and errno is set)
- Mode
  - “r” : Reading mode
  - “r+” : Read and write
  - “w” : Writing mode, file is created or truncated to zero length
  - “w+” : Read and write, but the file is created or truncated
  - “a” : Append mode, the file is created if it does not exist
  - “a+” : Read and append, the files is created if ti does not exist. Reading starts at the beginning, but writing done at the end

Closing Streams

int fclose (FILE *stream)

- Close a stream
- Returns
  - 0 if it worked
  - EOF if there was a problem (and errno is set)

fgetc / fputc (one byte at a time

```
int fgetc( FILE *stream);
int fputc(int c, FILE *stream):
```

- Read or write one (ASCII_ character (8-bits) at a time
    - Can be slow for large files
- fgetc reads a character from the stream and returns the character just read in (as unsigned char extended to int)
    - Returns EOF when at the end of file or on error
- fputc writes the character received as argument to the stream and returns the character that was just written out
    - Returns EOF on error

getc / putc and ungetc

```
int getc(FILE *stream);
int putc(int c, FILE *stream);
```

- Same as fgetc/fputc except they may be implemented as macros
- Use fgetc/fputc unless you have strong reasons not to

```
int ungetc(int c, FILE *stream);
```

- Pushes last read char back to stream, where it is available for subsequent read operations
- Only one pushback guaranteed

getchar / putchar

```
int getchar(void)
// same as fgetc(stdin)
```

- Reads a character from stdin
- Returns the character just read in, or EOF on end-of-file errors

```
int purchat(int c)
// same as fputc(c, stdout)
```

- Writes the character received as argument on stdout
- Returns the character that was just written out, or EOF on errors

More than one byte: get a line

```
char *fgets(char *buf, in size, FILE *in)
```

- Reads the next line from in into buffer buf
- Halts at ‘\n’ or after size-1 characters have been read
    - NUL is placed at the end
- Returns pointer to buf if ok, NULL otherwise
- Do not use gets(char *)! – buffer overflow

```
int fputs(const char *str, FILE *out)
```

- Writes the string str to out, stopping at '\0'
- Returns number of characters written or EOF

Formatted output

```
int fscanf(FILE *stream, const char *format, …);
int fprintf(FILE *stream, const char *format, …):
```

- Formatted input from file and output to file
- Like scanf() / printf(), but not from stdin or to stdout

For binary data

```
size_t fread (void *ptr, size_t sz, size_t n, FILE *stream);
size_t fwrite(void *ptr, size_t sz, size_t n, FILE *stream);
```

- Read / write a sequence of byte from / to a stream
- Return the number of items read or written
    - If smaller than n, EOF or error

```
Example:
int A[10][20];
size_t n = 10 * 20;
if (fwrite(A, sizeof(int), n, fp) != n) {
     // error
```

Moving file position indicator

```
long ftell(FILE *stream);
```

- Read file position indicator
- Return -1 on error

```
int fseek(FILE * stream, off_t offset, int whence);
```

- Set the file position indicator
- Return 0 on success and -1 on error

```
Example:
fseek(fp, 0, SEEK_SET);          // move to the beginning
fseek(fp, 200, SEEK_CUR);        // move forward 200 bytes
fseek(fp, -1, SEEK_END);          // move to the last byte
```

More useful stdio functions

```
//Check if end-of-file is set (after a read attempt!)
int feof(FILE * stream);

//Force write of buffered data
```

```
int fflush(FILE * stream);
```

- Read the manual pages!
- Check the return values!

Lecture 14

Mon. Sept. 30, 2019

- forgot what we did lol

Lecture 15

Weds. Oct. 2, 2019

Practice Exam 1

Lecture 16 - P1: Intro to Processes

Mon. Oct. 07, 2019

Process Basics

- A process is an instance of a program being executed
    - Core operating system (OS) concept
- In a multiprocessing OS
    - Multiple programs can be executed at the same time
    - Multiple instances of a program can be executed at the same time
- Executing multiple programs
    - Single-core: time-sharing
    - Multi-core: true parallelism + time-sharing

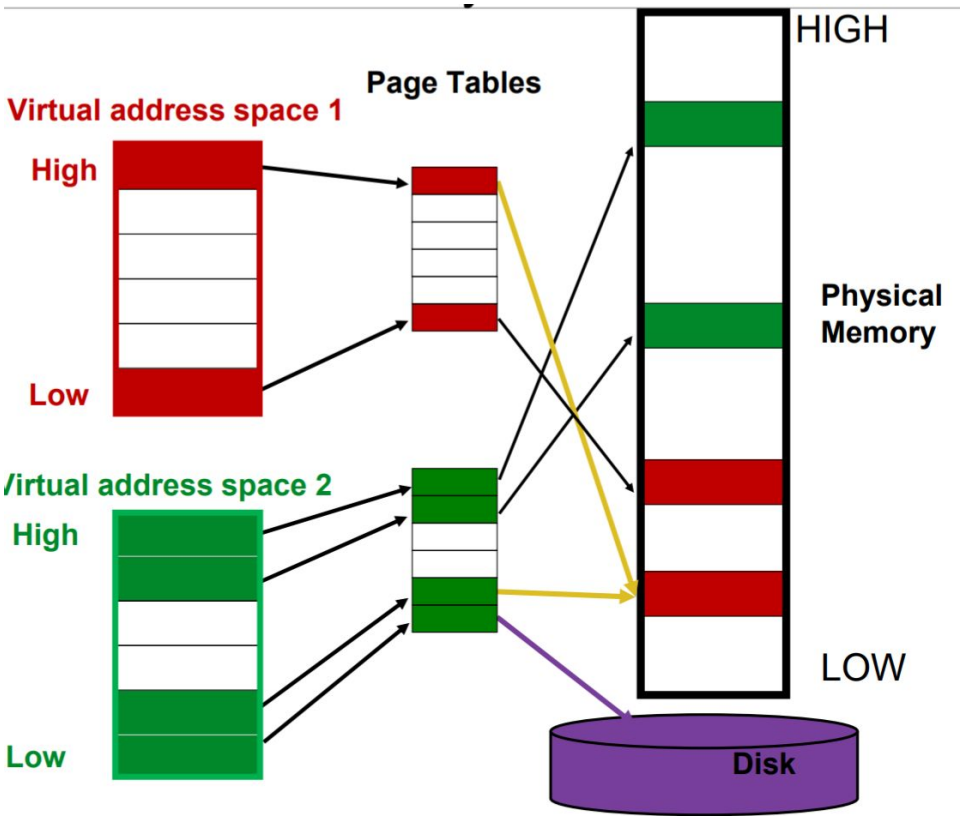
Process Management: OS View

- OS maintains a process table
    - Each process has a table entry, called process control block (PCB)
    - Typical PCB info
- OS scheduler picks processes to be executed at any given time
    - When a process is suspended, its state is saved in PCB
    - What about the process memory?

Process management	**Memory management**	**File management**
Registers	Pointer to text segment info	Root directory
Program counter	Pointer to data segment info	Working directory
Program status word	Pointer to stack segment info	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

Paged Virtual Memory: How Processes Share Memory

- Physical memory is shared by all processes
- Page table maps virtual address to physical address
- Multiple virtual pages can be mapped to the same physical pages



Process Management: User's View

- Events which cause process creation
  - System initialization
  - User request to create a new process (e.g., shell command)
  - Executing a shell script, which may create many processes
- Events which cause process termination
  - Normal program exit
  - Error exit
  - Fatal error, e.g., segmentation fault
  - Killed by user command or signal (Ctrl-C)

Useful Commands

- **ps** - List running processes
- **pstree** - Display the tree of processes
- top - Dynamic view of memory & CPU usage + processes that use most resources (to exit top, press q)
- **kill** - Kill a process given its process ID
  - **Try -9** option if simple kill does not work
- Additional functions/options in man page of each command

CSE 3100 Master Notes

Process Management: Programmer's View

- Process birth
  - Processes are created by other processes!
  - A process always starts as a clone of its parent process
  - Then the process may upgrade itself to run a different executable
    - Child process retains access to the files open in the parent
- Process life
  - Child process can create its own children processes
- Process death
  - Eventually calls exit or abort to commit "suicide"
  - Or gets killed

Birth via Cloning

- The function to create a new process in your code

```
#include
pid_t fork(void);
```

- Child is an exact copy of the parent
  - Both return from fork()
- **Only difference is the returned value**
  - In the **parent** process:
    - fork() returns the process identifier of the child (> 0)
    - If a failure occurred, it returns -1 (and sets errno)
  - In the **child** process: fork() returns 0 (zero)

Concurrency

- Parent and child processes return from fork() concurrently
  - They may return at the same time (on a multicore machine) or one after the other
  - Cannot assume that they return at the same time or which one "returns first" (even on a uni-core)
    - Order is chosen by OS scheduler

Cloning Effect

- On memory
  - The parent and child memory 100% identical
  - But are viewed as distinct by OS ("copy-on-write")
  - Any memory change (stack/heap) affects only that copy
  - Thus the parent and child can quickly diverge
- On files
  - *All files open in the parent are accessible in the child!*

    - I/O operations in either one move the file position indicator
- In particular
    - *stdin, stdout*, and *stderr* of the parent are accessible in the child

What can the parent do?

- Depends on application!
    - It could wait until the child is done (dies!)
        - Typical of a shell like bash/ksh/zsh/csh/….
    - It could run concurrently and check back on the child later
    - It could run concurrently and ignore the child
        - If child dies it enters a zombie state

Waiting on a child

```
#include <sys/wait.h>

pid_t wait(int * status);
pid_t (waitpid(pod_t pid, int * status, int options);
```

- Purpose
    - Block the calling process until a child is terminated
        - Or other state changes specified by options
    - Report status in *status (which is ignored if NULL is passed)
        - The cause of death
        - The exit status of the child (what he returned from main)
    - Return value identifies the child process (or -1 on error)
    - Run “man -S2 wait” for full details

Zombies

- A dead process, waiting to be 'reaped' (checked by its parent)
    - You cannot kill it, because it is already dead
    - Most resources released, but still uses an entry in the process table
- Parents should check their kids
    - On some systems, parents can say they do not want to check
- When a parent dies, ‘init’ becomes the new parent
    - Then the zombie child is reaped

System calls

- APIs used to request services from the OS kernel
    - Example: fork()
    - System calls are more expensive than normal function calls
    - Manuals for system calls are in section 2

man -S2 intro ; man -S2 syscalls

Summary

- Clone a process with fork()
    - The child is exactly the same as the parent
    - Check the return value
- Parents wait for child processes
    - Reap the zombies!

Lecture 17 - P2: Exec and Low-Level I/O

Mon. Oct. 09, 2019

Process upgrades

- Usually…
    - A fresh clone wants to run different code
- This is done by
    - Loading another executable3 into the process address space
        - [picked up from the file system of course]
- Note: **open files are NOT AFFECTED** by the upgrade operation

The exec family

- The act of “upgrading’ is done by the child with a system call
    - Many variants ‘many -S3 excel” for all details

```
#include <unistd.h>
int execl(const char *path, const char *arg0, …
                        /*, (char *) NULL */ );
```

- The path to the executable to load inside our own address space
- A list of arguments to be passed to the new executable
- A final NULL pointer to give the “end of argument list”
- If successful, execl() does not return! Started a new process

Exec example

- We will turn the child process into the following executable

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

int main(int argc,char* argv[]) {
   int i, sum=0;
   for(i=1;i<argc;i++)
      sum += atoi(argv[i]);
   printf("sum is: %d\n",sum);
   return 0;
}
```

This is a simple "adder" program that computes the sum of its integer arguments

Parent Program

```c
int main() {  // complete code is in demo/padder
   char *cmd1 = "./adder", *cmd2 = "addder";
   pid_t child = fork();
   if (child == 0) {
      printf("In child!\n");
      execl(cmd1,cmd1,"1","2","3","10",NULL);
      printf("Oops.... something went really wrong!\n");
      perror(cmd1);
      return -1;
   } else {
      printf("In parent!\n");
      execl(cmd2,cmd2,"100","200","300",NULL);
      printf("Oops.... something went really wrong!\n");
      perror(cmd2);
      return -1;
   }
}
```

How is executable found?

- Specify a path, like /bin/ls
- Specify a file, and the system searches in directories listed in PATH
    - echo $PATH in bash to see directories separated by ':'

in execl(const char *path, const char *arg0, …

```
                        /*, (char *) NULL */ );

// execlp() searches paths for file
int execlp(const char *file, const char *arg0, ….
                        /*, (char *) NULL */ );
```

execv family

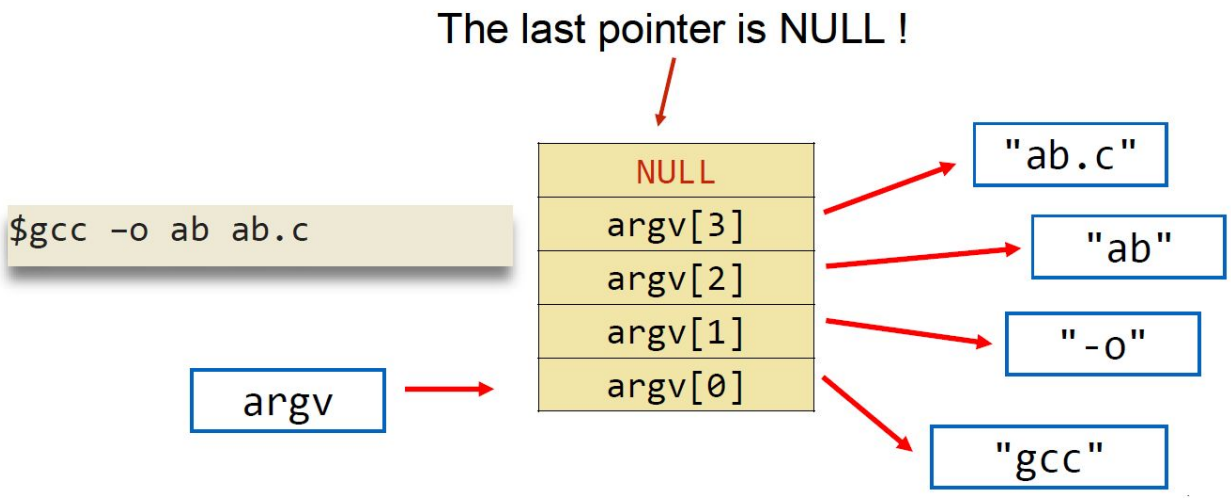
// If the number of arguments is unknown at compile time

```
#include <unistd.h>
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
```

- The arguments in execl in are placed in an array
    - argv is the argv you see in the main function!
- execv needs a path while execvp can search file in PATH
- Start a new process if successful. Similar to execl

argv to execv and execvp

- Note the NULL pointer at the end
- Why?



File APIs

- Remember the (C standard library) IO APIs
    - The "**f**" family (fopen, fclose, fread, fgetc, fscanf, fprintf,...)
    - All these use a FILE* abstraction to represent a file
        - Additional features: user-space buffering, line-ending translation formatted I/O, etc.
- UNIX has lower-level APIs for file handling
    - Directly mapped to system calls

        - Open, close, read, …
    - Use file descriptors [which are just integers]
    - Deal with bytes only

Some low level file APIs

- Read the man pages (man -s2 …) for more functions

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int open(const char *path, int oflag);
int close(int fd);

ssize_t read(int fd, void *buf, size_t nbyte);
ssize_t write(int fd, const void *buf, size_t nbyte);

off_t lseek(int fd, off_t offset, int whence);
```

Open a file

```
#include <fcntl.h>
#include <unistd.h>

int open(const char *path, int oflag);
```

- Parameters
    - path: the path to the file to be open/created
    - oflag: read, write, or read and write, and more (on the next slide)
- The function returns a file descriptor, a small, nonnegative integer
    - Return -1 on error

Flags in open()

- Must include one of the following:

O_RDONLY (read only), O_WRONLY (write only), or O_RDWR (read and write)

- And or=ed (|) with many optional flags, for example,
    - O_TRUNC: Truncate the file (remove existing contents) if opening a file for write
    - O_CREAT: Create a file if it does not exist

Example:

```
// remember open() returns -1 on error
```

```
fd1 = open("a.txt", O_RDONLY);   // open for read
fd1 = open("a.txt", O_RDWR):     // open for read and write
fd1 = open("a.txt", O_RDWR | O_TRUNC);read, write, truncate the file
```

Create a file with open()

```
// a mode must be provided if O_CREAT or O_TMPFILE is set
int open(const char *path, int oflag, int mode);
```

- mode: specify permissions when a new, or temporary, file is created

```
open("b.txt", O_WRONLY|O_TRUNC|O_CREAT, 0600);

// open b.txt for write. If the file exists, clear (truncate) the
contents.

// if the file does not exist, create one, and set the permission so
that the owner of the file can read and write, but other people
cannot.
```

File descriptor vs stream

```
#include <stdio.h>
int fileno(FILE *stream);
// returns a file descriptor for a stream
```

FD	FILE *
0	stdin
1	stout
2	stderr

File descriptors after fork and exec

- Opened files are NOT AFFECTED by the upgrade operation

```
pid_t pid = fork();
assert(*pid >= 0);
if (pid == 0){
      // Child process can access FDs 0, 1, and 2
      // if execl() is successful, gcc can access FDs 0, 1, and 2
      execlp("gcc", "gcc", "a.c", NULL);
      // If control gets here, execlp() failed.
      // Remember to terminate the child process!
```

```
    return 1;
}
```

Lecture 18 - P3: Redirection

Mon. Oct. 14, 2019

Review

- Function open() returns a file descriptor, a non-negative integer
- the file descriptor is used later in funcs. like read() and close()
- Every opened file has a file descriptor
  - stdin: 0
  - stdout: 1
  - stderr: 2
- Files opened in a process remain open after fork() and exec(0

Shell redirections

- Available when executing commands in your shell (e.g. bash)
  - Implemented with the close/open/dup technique

```
$ command <infile > outfile
```

  - < infile: Take input from file infile
  - > outfile: Send output to file outfile
- Other variants
  - >> outfile: Append output to file outfile
  - 2> outfile: Send errors to file outfile
  - &> outfile: Send both output and errors to file outfile
- Read the manual for more variants like 2>>, 2>&1, etc.

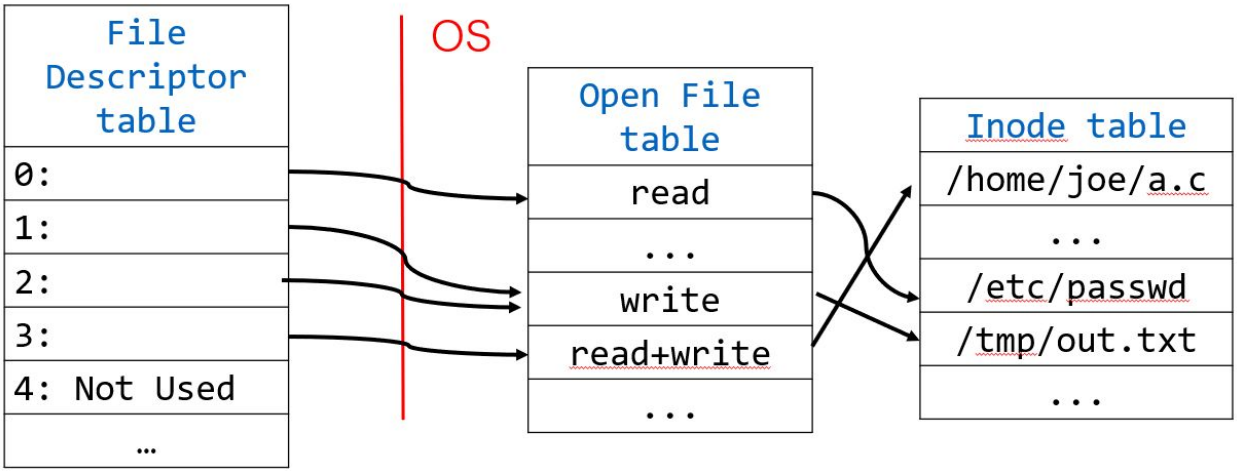
Shell redirection examples

```
$ sort < file.txt > sorted.txt
```

- sort will read lines from file.txt, instead of terminal
- The output or sort will be saved in sorted.txt
  - You cannot see it on screen
- The statements in sort are not changed
- They read from stdin (0), and print to stdout (1)

File Descriptor Table

- Each process has a **file descriptor table**
  - Holds indices of entries into the **Open File Table** managed by OS
- The system-wide **Open File Table**
  - Records the mode of the opened files (e.g., reading, writing, appending)
  - Holds index into the **Inode Table** that has the actual file name and location on disk



Duplicating File Descriptors

- Do not change file descriptor table directly
- Used open() and close() and two new functions

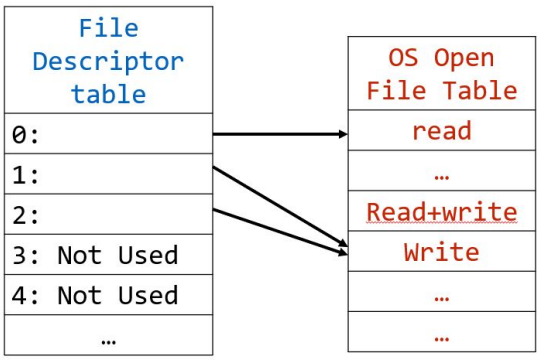
```
#include <unistd.h>

int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

- dup() copies oldfd to the **first available entry** (in FD table)
- dup2() copies oldfd to newfd
  - Closes newfd if it is in us

***There is dup3(), but it is not in POSIX. We should not use it in this course.

Example: stdout redirect

- A program can change its standard output
- How?



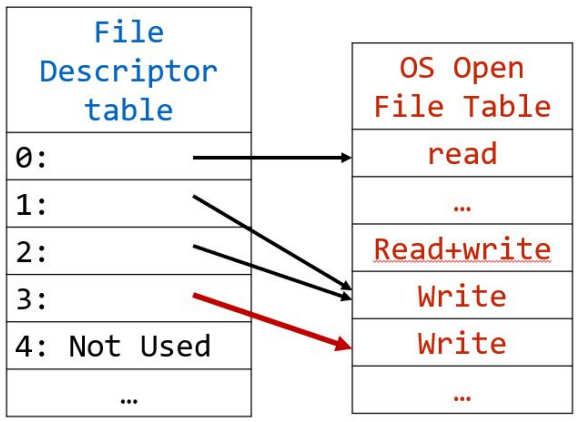
Steps for redirecting stdout

1. open(). open file (and save the file descriptor in fd)
2. dup2(). copy fd to 1 (so that the file descriptor 1 points to the file just opened)
3. close(fd)

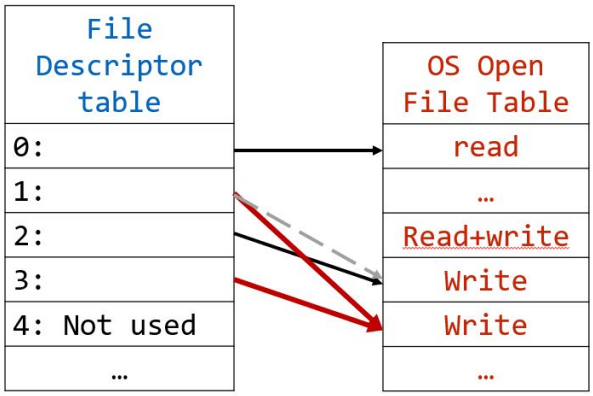
Example: stdout redirect

<table>
<tr>
<td>

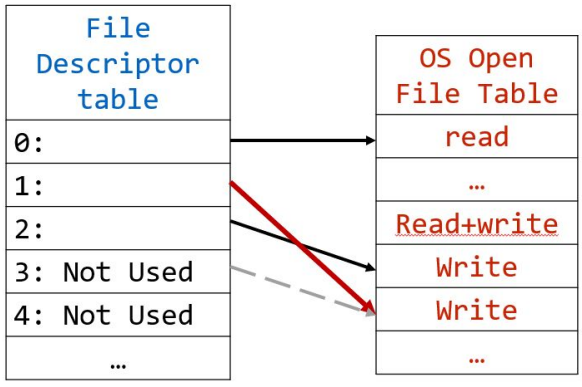
- Open the new file for writing; 3 is the returned fd
  - We will use 3 instead of a variable in this example

</td>
<td> 
</td>
</tr>
<tr>
<td>

```
// Method 1: two functions. not atomic
close(1);  dup(3);
// Method: better. dup2() closes newfd first
dup2(3, 1);
```

</td>
<td> 
</td>
</tr>
<tr>
<td>

```
close(3);
```

</td>
<td> 
</td>
</tr>
</table>

Implementing redirections

- How does bash do redirection for other processes?
- When bash starts a new process, it does fork() and exec()
- Recall that the file descriptor table **is preserved during fork & exec**

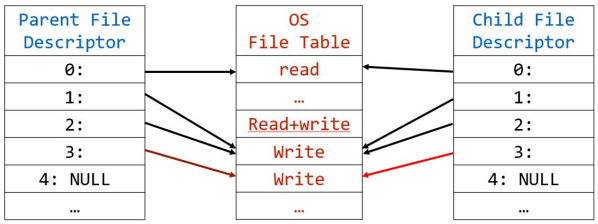
Idea:

- In child process, set up proper file descriptors before upgrading
    - Simply change the files corresponding to stdin, stdout, or stderr

Example: redirecting stderr for another program

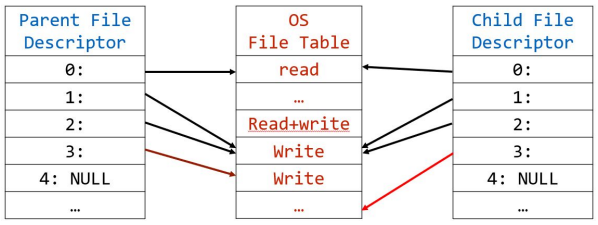
<table>
<tr>
<td>

- Assume the parent uses FD 3
- After fork(), the child has the same file descriptors as the parent
    - Close FDs that are not needed!

</td>
<td> 
</td>
</tr>
<tr>
<td>

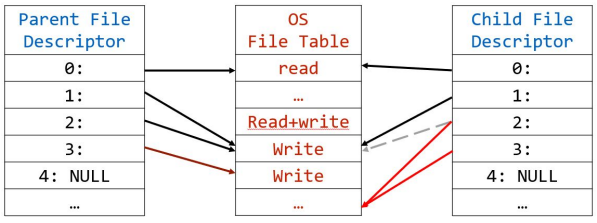
How do you close FD 3 in child process?

- Child: open the file (to save error output)

</td>
<td> 
</td>
</tr>
<tr>
<td>

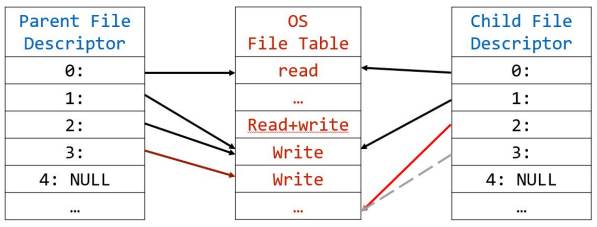
- Child:

```
dup2(2, 3);
close(3); // not yet run
```

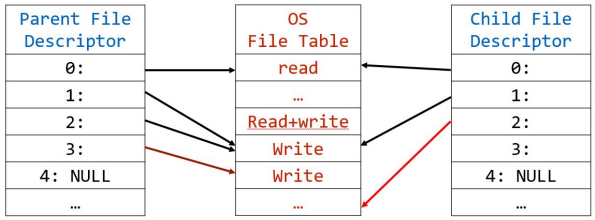
</td>
<td> 
</td>
</tr>
<tr>
<td>

- Child:

```
dup2(2, 3);
close(3); // now running
```

</td>
<td> 
</td>
</tr>
</table>

- Child: exec and the new executable has redirected stderr!
- On exec, open file descriptors are preserved!



Redirecting stdout for a child process

- A process would like to start a new program, and redirect stdout of the new process to a file
- Where & when should the file be opened?
- Select the best options.

a. Before fork(), in parent
b. After fork() in parent
c. Before exec in child (after fork())
d. After exec in child (after fork())
e. None of the above

Summary of Steps in Child Process

- Close FDs that are opened in parent and not needed in child
- open(). Open a file (and save the file descriptor in fd)
- dip2(). Copy FD to the right place
- close(fd)
- Exec

More questions on redirecting stdout?

- Where & when should the new file be opened?
- Where & when should you call close(1)/
- Where & when should you call dup?
- Where & when should close (newfd) be called?

Summary

- A program can direct input/output
  - It is done with open(), close(), dup(), or dup2()
- FDs are preserved on exec
  - Close file descriptors that are not needed

Lecture 19 - P4: Inter-Process Comm. w/ Pipes

Weds. Oct. 16, 2019

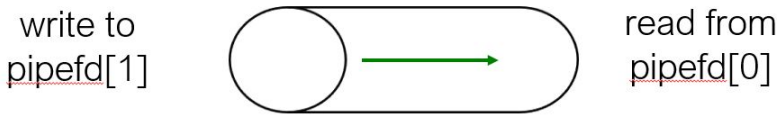
Inter-process communication (IPC)

- Files
- Pipes
- Named pipes
- Sockets
- Message queues
- Shared memory
- Synchronization primitives
    - Semaphores, Signal, etc

pipe()

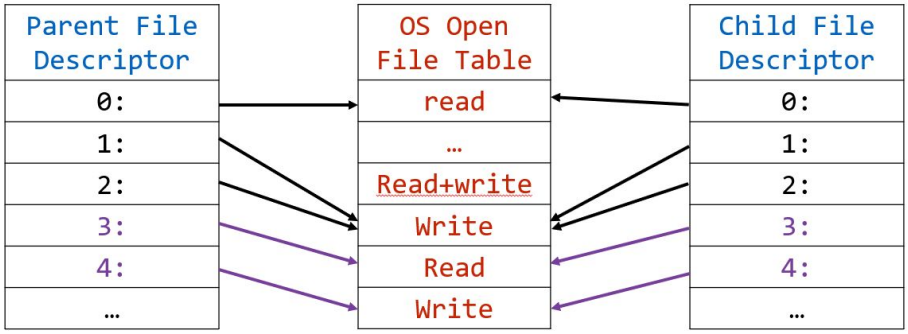
#include <unistd.h>
int pipe(int pipefd[2])

- Create a one-way pipe (a buffer to store a byte stream)
- Two FDs in pipefd. pipefd[0] is the read end, pipefd[1] is the write end
- Return 0 if successful
- Pipes allow IPC. One process writes and the other one reads



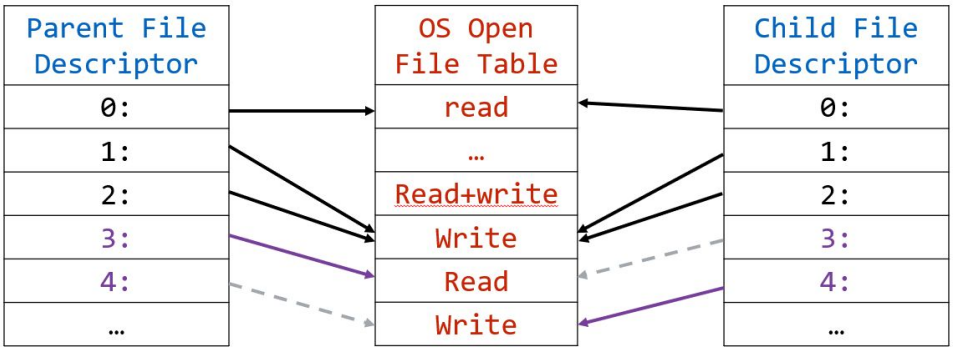
Connecting two processes

- Parent creates a pipe and gets two FDs (e.g. 3 and 4)
- After fork(), the child has 3 and 4, too
- One process can write to FD 3, and the other can read from FD 4
    - Close unused FD!



Closing FDs not in use

- If the pipe is for parent to read and for child to write
  - Parent: close(4)
  - Child: close(3);
- Then the child can write to and parent can read from the pipe. See demo code!



Questions

- What would you do if you need two-way communication between parent and child?
- After exec, the new program gets the file descriptors for the pipe, too
- How can the new program use the pipe?
  - A program is aware of FDs 0, 1, and 2, but not 3 or 4

Pipeline in shell

- Shell supports pipelines

```
cmd | cmd2 arg 21 arg22 | cmd3 arg31
```

- stdout of a command is connected is to stdin of the nex command
  - Done with pipes on Linux/Unix
  - cmd1 writes to a pipe and cmd2 reads from it
- Example:

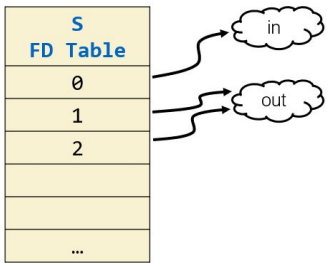
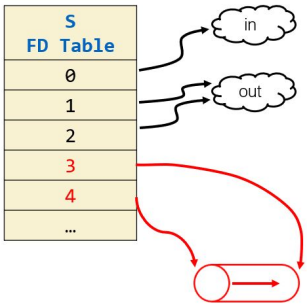
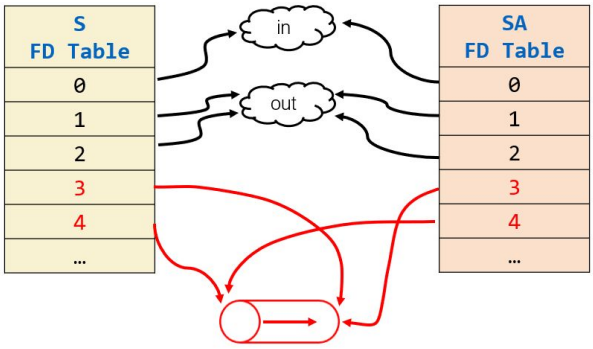
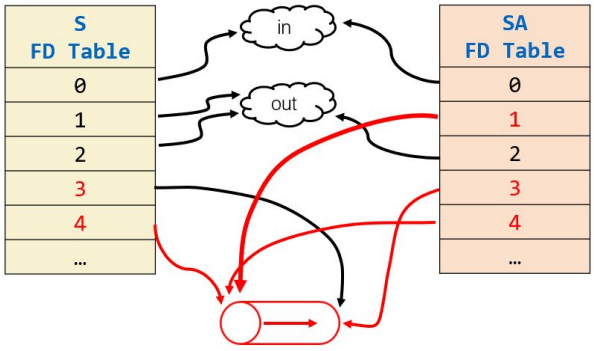
```
ls | tr a-z A-Z | wc
```

Example: connect two programs with a pipe

Start a pipeline in program S (aka, the shell): A | B

- High-level strategy (missing clean up!)
  - Create a pipe
  - Fork #1
    - In child process
      - Redirect stdout to the write end of the pipe
      - Start A, by calling exec
  - Fork #2

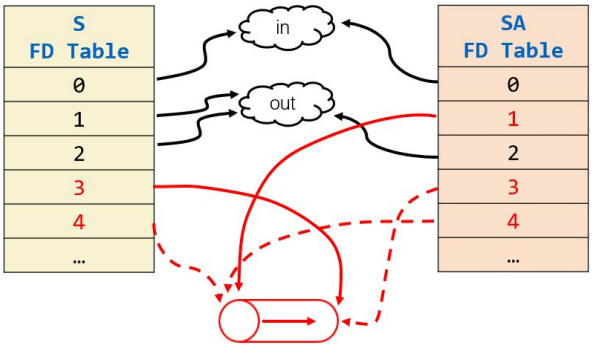
- In child process
  - Redirect stdin to the read end of the pipe
  - Start B, by calling exec

At the beginning • S has only 0, 1, and 2 open	
Pipe Creation • S creates a pipe by calling pipe() ○ A pair of FDs is returned	
Fork #1 • S: fork() ○ FD table is duplicated	
Redirect in first child process • SA: dup2(4, 1) ○ Or close(1); dup(4);	

CSE 3100 Master Notes

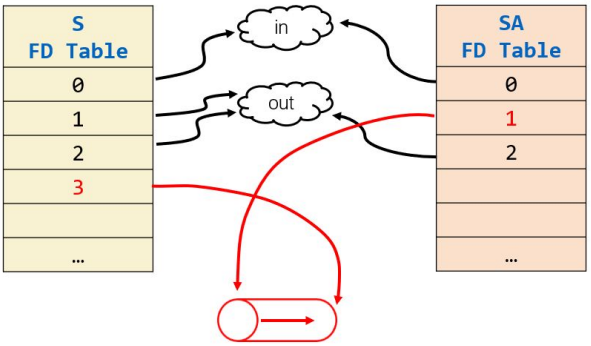
Clean up #1

- S: close(4)
- SA: close(4), close(3)
- SA can then exec into A



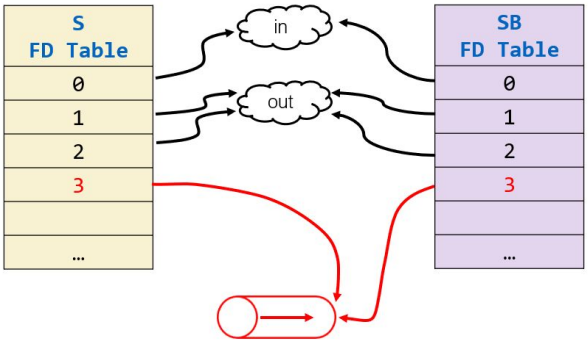
After clean up #1

- S: close(4)
- SA: close(4), close(3)
- SA can then exec into A



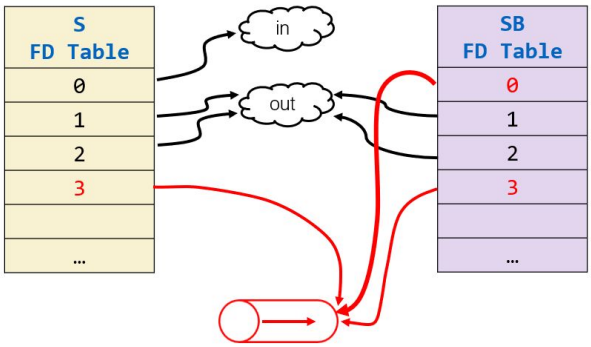
Fork #2

- S: fork()
  - Note that 4 has been closed in S



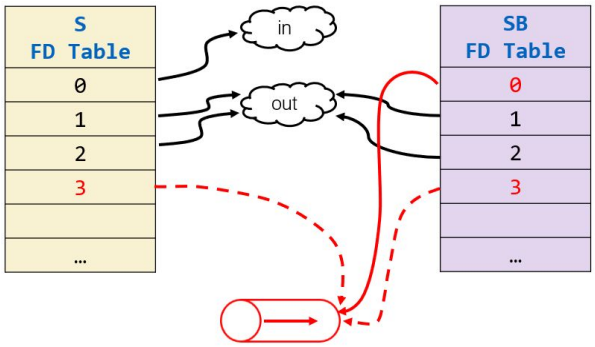
Redirect in second child process

- SB: dup2(3,0)
  - or close(0); dup(3);



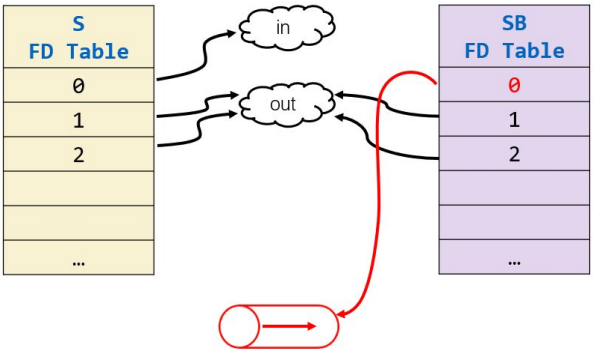
Clean Up #2

- S: close(3)
- SB: close(3)
- SB can then exec into B



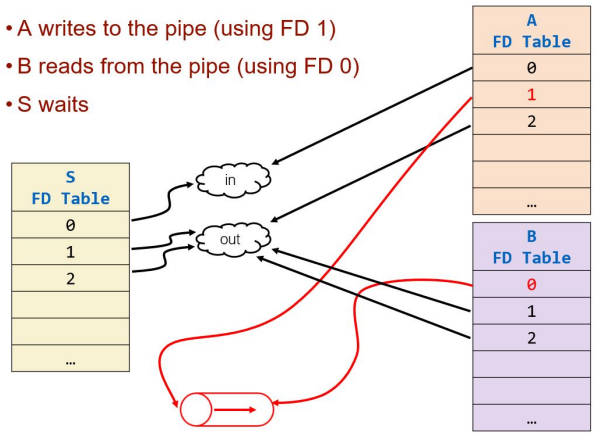
After clean Up #2

- S: close(3)
- SB: close(3)
- SB can then exec into B



Final set up

- A writes to the pipe (using FD 1)
- B reads from the pipe (using FD 0)
- S waits



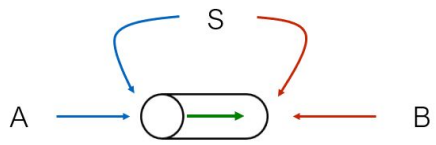
FDs of a dying process

- When a process ends, all its open FDs are automatically closed
- What happens to the processes on the other end of the pipe?

Example:

Assume S does not read or write, but have FDs of the pipe

- If both A and S die, B gets EOF when all buffered data are consumed
- If A dies, B will wait for more data (assuming S may write)
- If both B and S die, A gets an error (SIGPIPE) when writing
- If B dies, A will wait if the pipe is full (assuming S will read)



Close file descriptors
a process does not use!

Going further…

- You can repeat this to create a long pipeline
  - e.g., connect B's stdout to stdin of another process C
- Draw pictures to find how pipes are used
  - And what FDs need to be closed

Remember

- Processes are running in parallel once they are created
  - Although we showed the operations in sequence
- All processes in the pipeline are running concurrently on Linux
  - As soon as data are sent in the pipe…
  - The next process can pick them up on the work

Atomicity of read() and write()

```
n_r = read(fd, buf, N);
n_w = write(fd, buf, N);
```

- wite() and read() returns the # of bytes actually read/written
- The number maybe less than the requested

Atomicity - The degree a program is guaranteed to be isolated from other operations that may be happening at the same time

- Atomicity of write() is guaranteed if the # of bytes < PIPE_BUF
  - The bytes will be consecutive
  - The default value of PIPE_BUF is 4096 on Linx
- For read(), it is fine if all writes and reads are of the same size
  - Otherwise, need special handling

Starting a 2-stage pipeline - 1

```
// A | B

pipe(pipefd)          // pipefd is an array of 2 int's
pid_a = fork()              // for A
if (pid_a == 0){            // child process for A
     dup2();                // setup stdout for A
     close both FDs in pipefd
     exec to start A        // remember to exit from child on error
```

```
}
close(pipefd[WR_END]);      // No need to keep it open in parent
```

Starting a 2-stage pipeline - 2

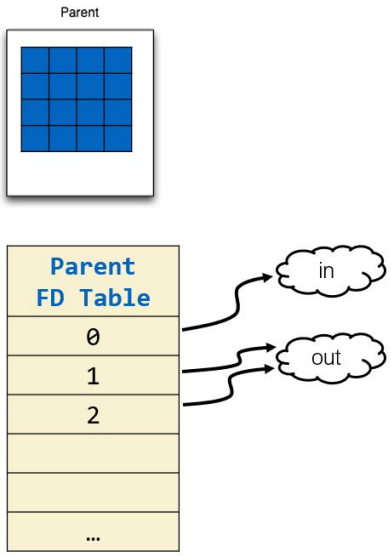
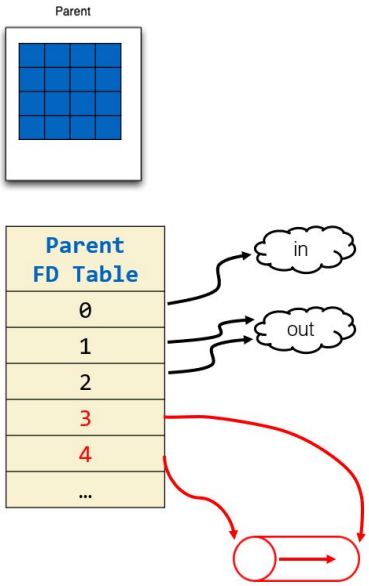
```
pid_b = fork()              // for B
if (pid_b == 0){            // child process for B
      dup2();               // setup stdin for B
      close(pipefd[RD_END]);
      exec to start B       // remember to exit from child on error
}
close(pipefd[RD_END]);      // No need to keep it open in parent
```

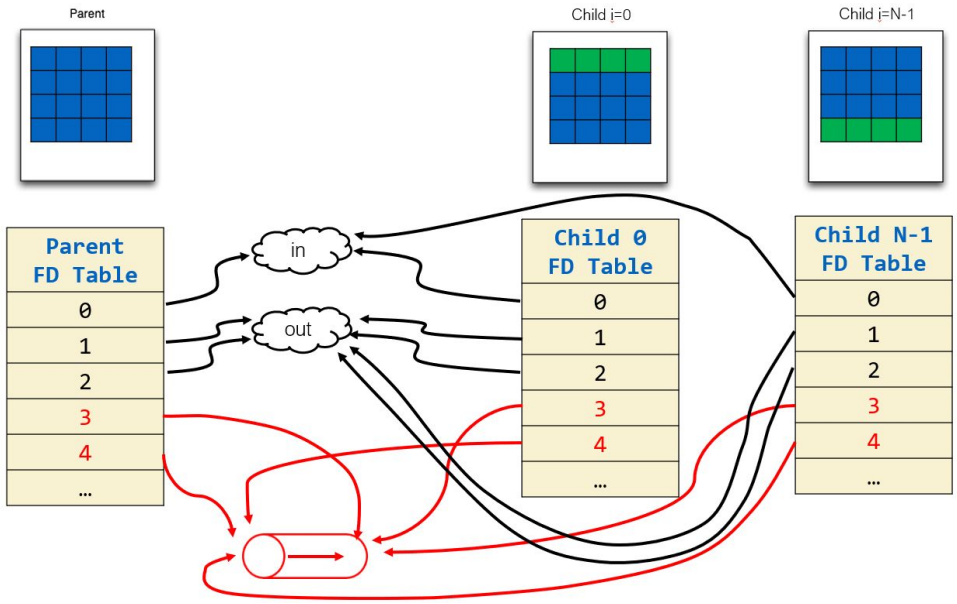
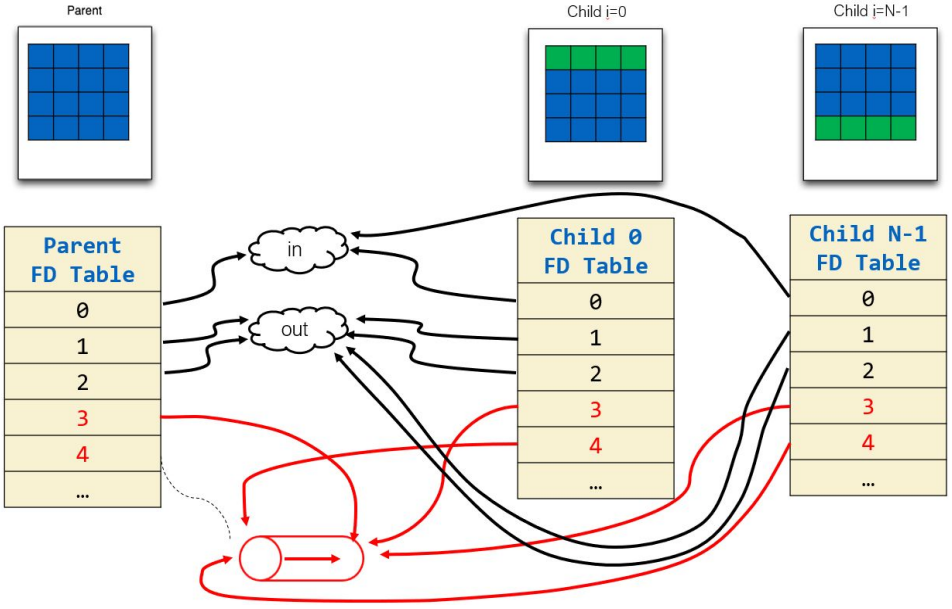
Using Pipes to Sum Matrix Rows Concurrently

- See the complete code in the demo repo.

```
int main(void)
{
   int i, row_sum, sum = 0, pd[2], a[N][N] = {{1, 1, 1}, {2, 2, 2}, {3, 3, 3}};

   if (pipe(pd) == -1) error_exit("pipe() failed");  /* create pipe */

   for (i = 0; i < N; ++i)
      if (fork() == 0) {            /* create a child process for each row */
         row_sum = add_vector(a[i]); /* compute the sum of a row */
         if (write(pd[1], &row_sum, sizeof(int)) == -1) /* write to pipe */
            error_exit("write() failed");
         return 0;                                    /* exit from child */
      }
   /* better to close the write end in the parent */
   for (i = 0; i < N; ++i) {
      if (read(pd[0], &row_sum, sizeof(int)) == -1) /* read from pipe */
         error_exit("read() failed");
      sum += row_sum;                    /* calculate the total */
   }
   printf("Sum of the array = %d\n", sum);
   /* wait for child processes*/
```

26

Parent Process	
Pipe Creation	

After forking (without cleanup)	
After forking (with cleanup)	

Lecture 20 - T1: Instruction & Basic Management

Mon. Oct. 21, 2019

Overview

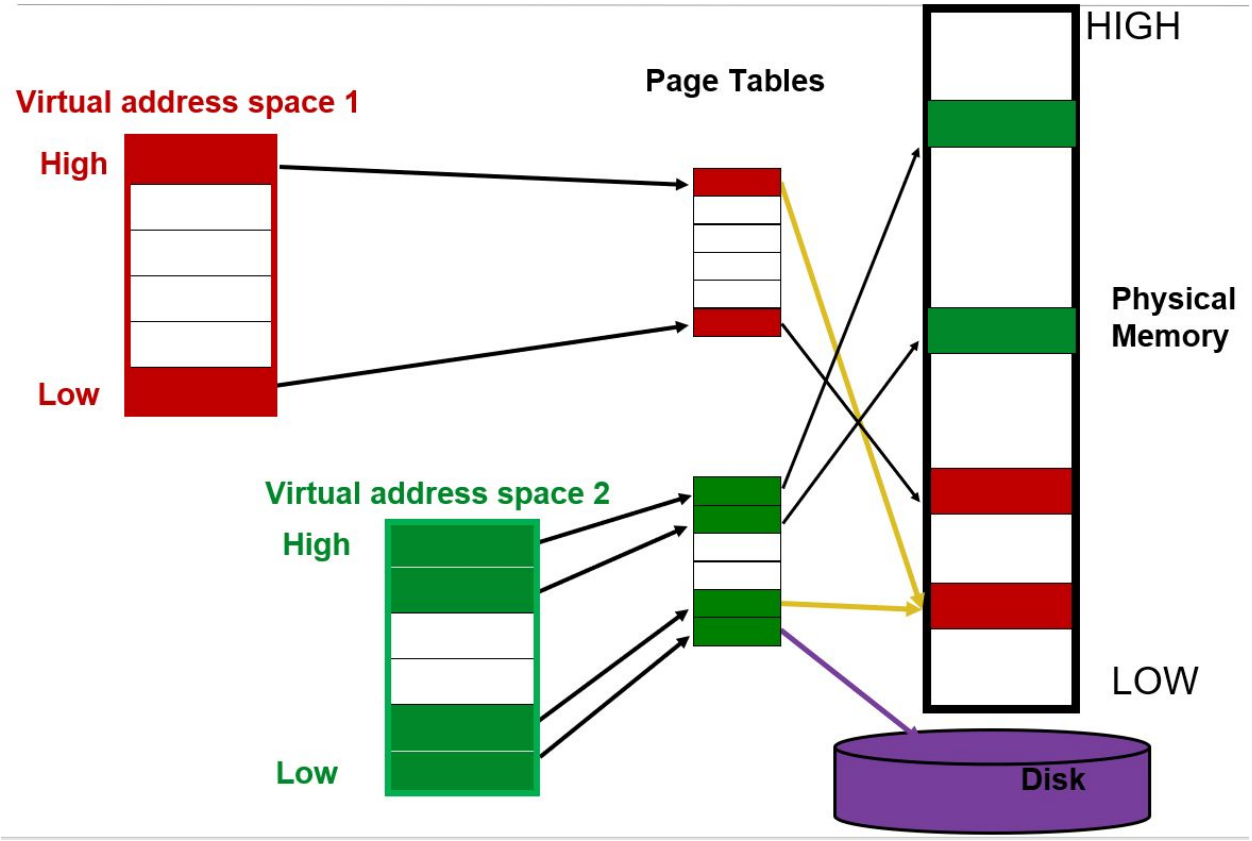
- Motivation & Concurrency

    - Thread abstraction
- Thread creation & termination

Motivation

- Processes for concurrency
    - Process do run concurrently on O.S.
        - On multi-core or single-core processors, through time-sharing
- Processes also provide protection

Each process has its own virtual address space



Why?

- Well…
    - Nobody can interfere
    - Nobody can move his “stuff”
    - Nobody can play with his “stuff”
    - Nobody can break his toys
- It is very very safe!

Downsides?

- Nobody to play with
  - You can get bored quite quickly
- Nobody to do your chores
  - Do everything yourself!
- You can't leave the castle
  - It is also your prison
- Communication with the outside is tricky and very limited (but safe)
  - Banging on a pipe…
  - Smoke signals…
  - Sockets…

Where are her buddies (e.g., processes)?

- In other castles!
  - Equally alone / isolated
  - You cannot get together easily
  - Sharing is limited

Threads?

- Inviting other 'living creatures" iinside your castle
  - These "creatures" can
    - Move around independently of you
    - Do work / chores on your behalf
    - Communicate on your behalf
    - Play with you
    - Use/share your toys
- Essentially
  - No limits on their abilities in their address space
  - As powerful as the "Lord of the Castle"

Refining our definitions

- Process is a bundle grouping
  - A virtual address [memory]
  - A collection of files / sockets [IO]
  - A collection of concurrent threads [execution units]
- Thread is a light-weight entity
  - Can run concurrently w/ other threads
  - Share resources in the process [having same rights]

- Confined to a single process [cannot “move” to another]
- Can be created and destroyed [different life cycles]

Lightweight ‘processes” that share the address space (and other resources in a process)

The Circle of Life

- How is a thread created
  - By another thread! `pthread_create()`
  - It is given a stack to execute and a function to run
- What about the 1st thread?
  - When a process is created, there is a single thread [starts alone]
- How does a thread die?
  - Voluntarily, after completing its task `pthread_exit()`
  - Requested by another thread `pthread_cancel()`
  - When the process dies (along with all thread in it)
    - Any thread calls exit()
      - Note that exit() is called when main() returns!

!!! Any thread can bring down the whole castle !!!

The pthreads API

- ANSI/IEEE POSIX 1003.1 - 1995 standard
- These types of routines:
  - Thread management: create, terminate, join, and detach
  - Mutexes: mutual exclusion, creating, destroying, locking, and unlocking mutexes
  - Condition variables: event driven synchronization

The Pthreads API naming convention

Routine Prefix	Function
pthread_	General pthread
pthread_attr_	Thread attributes
pthread_mutex_	mutex
pthread_mutexattr	Mutex attributes
pthread_cond_	Condition variables
pthread_condattr	Conditional variable attributes
pthread_key_	Thread specific data keys

The Pthreads API

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

- Add '-pthread' option to compile on Linux

```
cc -pthreads a.c
```

- PThread functions do not set errno on errors
- Many types are defined in pthread library. They are opaque objects
  - Cannot make assumptions on the representation/implementation

For ex.): should use pthreads_equal() to compare two thread IDs

main() function of thread

void * thread_main(void * arg)

- The "main" function of a thread
  - Can be any name you like
- Takes a pointer as the only parameter
  - It can point to anything, int, char, string, or a structure
- Return a pointer
  - It can point to anything, int, char, string, and a structure
  - **However, do not point to local variables on stack -**

Threads creation

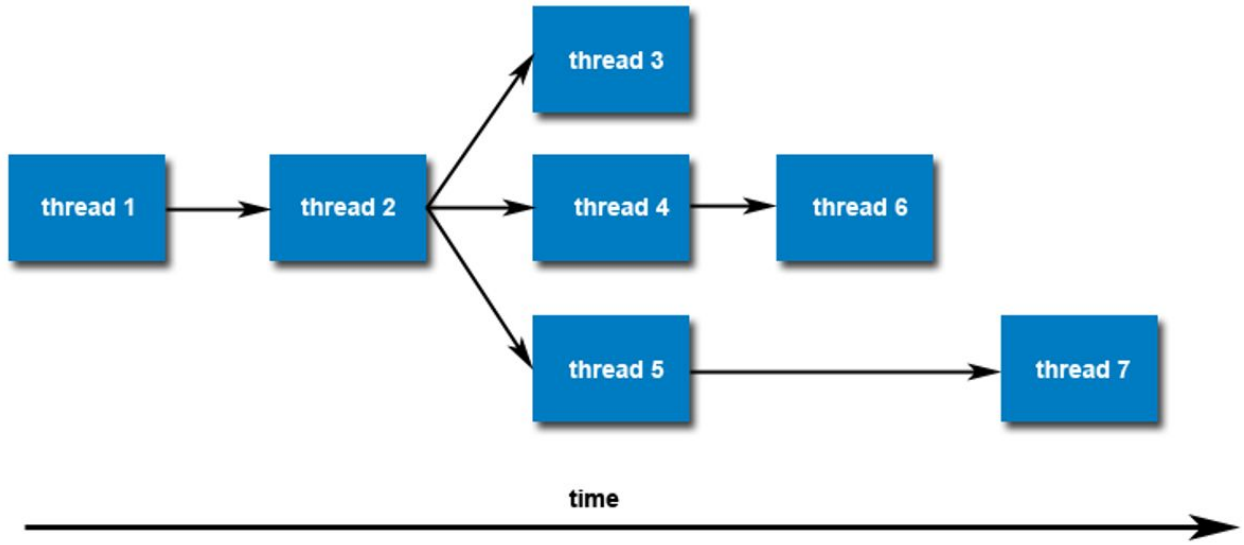
```
int pthread_create(
                pthread_t *  thread,
                pthread_attr_t * attr,
                void * (*start_routine)(void*),
                void * arg);
```

- Returns 0 if successful, and non-zero (> 0) if error
- *thread is the returned thread ID, if successful
- attr specifies the attribute for the thread. NULL for defauly
- start_routine()
- void * arg is passed to start_rountine()

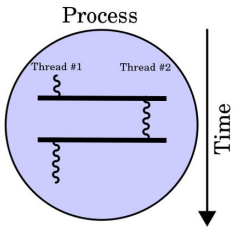
- Thread equivalent of fork(), but it does create (not "clone")

Thread creation

- Once created, threads are peers, and may create other threads
- There is no implied hierarchy or dependency
  - All threads are equal!

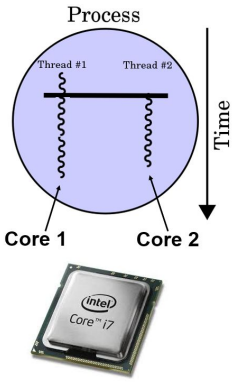


What about concurrency?



- Single-core CPU: Timesharing
  - When one thread is waiting for an IO to complete…
  - … another thread can use the CPu

What about thread concurrency?



- Multi core CPU: true concurrency
  - MIMD architecture: multiple instructions, multiple data
  - Threads can execute in parallel, one on each core
  - OS can still preempt threads
  - Useful when #threads >> #cores → timesharing is still used!

Thread Termination

- Return from the start_routine function, or
- Call pthread_exit()

```
void pthread_void(void* status)
```

- The function always succeeds and does not return
- Status can be obtained by other threads
- Similar to process termination
  - `main()` returns, or `exit()` called by any thread

Joining a thread

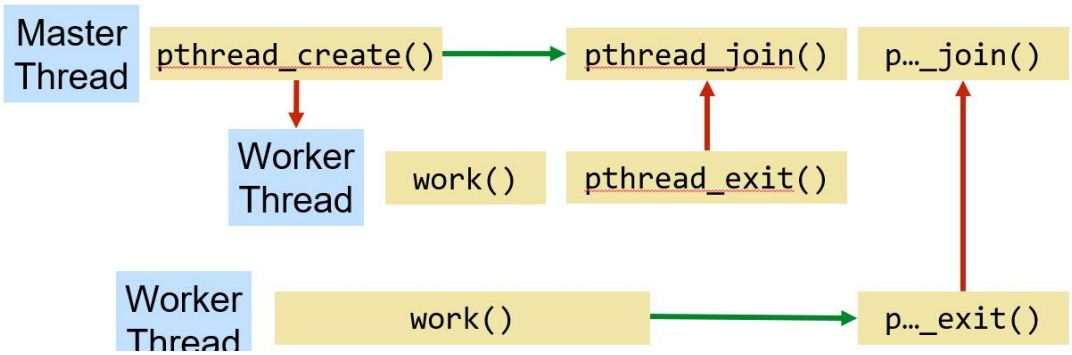
```
int pthread_join(pthread_t tid, void** status)
```

- Wait for a thread to complete
  - Blocks the calling thread until thread id terminates

- Can obtain the exit status of the thread
  - Pass NULL to ignore the return value
  - Why is the type of status void (**)? - A pointer to a pointer that can be changed
- Equivalent of **`waitpid()`** for processes

Joining a thread - 2

- Joining is a simple way to accomplish synchronization
  - The calling thread can obtain the target thread's termination return status if it was specified in the target thread's call to pthread_exit()



Reading

- Book "Programming with POSIX Threads"
  - Chapter 1, 2, 3, 4,
  - Chapter 5, Section 1- 4
- Nice to read too
  - Chapter 6, Sectoins 1-5
  - Chapter 8 [debugging]
  - Chapter 9 [reference]
  -

Passing Arguments and Getting Results Back

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

#define NUM_THREADS   8

struct thread_data
{
   int  thread_num;
   char* message;
   int   len;
};

void* PrintHello(void* threadarg)
{
   struct thread_data* my_data = (struct thread_data*) threadarg;

   sleep(1 + 5*(my_data->thread_num % 2) );

   my_data->len = strlen(my_data->message);
   printf("Thread #%d: %s  length=%d\n", my_data->thread_num, my_data->message,
    my_data->len);

   pthread_exit(NULL);
}
```

```
int main(int argc, char* argv[])
{
   pthread_t threads[NUM_THREADS];
   struct thread_data thread_data_array[NUM_THREADS];
   char* messages[NUM_THREADS];
   int rc, t;

   messages[0] = "English: Hello World!";
   messages[1] = "French: Bonjour, le monde!";
   messages[2] = "Spanish: Hola al mundo";
   messages[3] = "Klingon: Nuq neH!";
   messages[4] = "German: Guten Tag, Welt!";
   messages[5] = "Russian: Zdravstvuyte, mir!";
   messages[6] = "Japan: Sekai e konnichiwa!";
   messages[7] = "Latin: Orbis, te saluto!";
```

```
   for( t=0; t<NUM_THREADS; t++ ) {
      thread_data_array[t].thread_num = t;
      thread_data_array[t].message = messages[t];
      printf("Creating thread #%d\n", t);
      rc = pthread_create(&threads[t], NULL, PrintHello, &thread_data_array[t]);
      if (rc) {
         printf("ERROR; return code from pthread_create() is %d\n", rc);
         exit(-1);
      }
   }
   int grand_total = 0;
   for( t=0; t<NUM_THREADS; t++ ) {
      printf("Joining thread #%d\n", t);
      rc = pthread_join( threads[t], NULL );
      if( rc ){
         printf("ERROR; return code from pthread_join() is %d\n", rc);
         exit(-1);
      }
      grand_total += thread_data_array[t].len;
   }
   printf("Grand total = %d\n", grand_total);
   pthread_exit(NULL);
}
```

28

Common ways to use threads

- Pipeiline
    - A task is broken into a series of sub-operations, each of which is handled in series, but concurrently, by a different thread (think automobile assembly line)
- Manager/worker

    - A single thread, the manager assigns work to other threads, the workers. The manager handles all input and parcels out work to workers
    - Two common forms: static worker pool and dynamic worker pool
- Peer
    - Similar to the manager/worker model, but after the main thread creates other threads, it participates in the work

Applications of threads

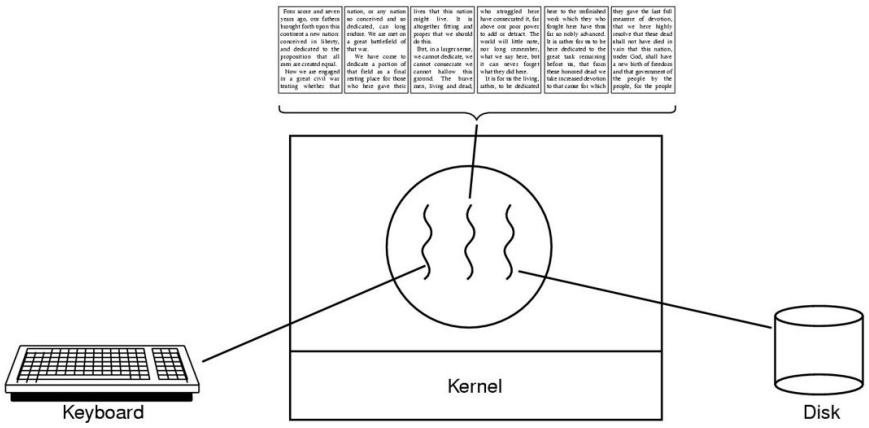
- Parallel computing
    - On multi-core machines, threads can be executed at the same time
- Overlap CPU work with I/O
    - While one thread is waiting for an I/O, others can perform CPU work
- Asynchronous event handling
    - e.g., a web server can both transfer data from previous requests and manage the arrival of new requests
- Priority/real-time scheduling
    - Important tasks can be scheduled with higher priority
- Computer games
    - Each thread controls the movement of an object

Why is it hard for the princess to do everything by herself?

- Imagine an application like Microsoft Word
- When you are tying, the process needs to
    - Listen for keystrokes and display the text on screen
    - Save data periodically to disk so you do not lose data in case of crash
    - Reform the document while edit the text
- How can a single process (i.e., one princess) can do all these concurrently?

The answer is she cannot without affecting application performance

We solve it by creating multiple threads!



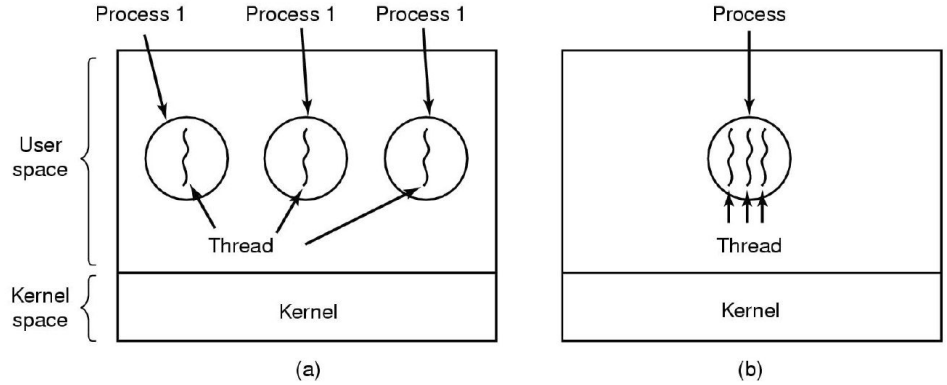
A word processor with three threads.

Usage Examples?

- We will create multiple threads, one thread for each sub tasks
  - One thread controls a GUI
  - One thread reformats the document
  - One thread does background tasks (e.g., savings!)
- Other examples
  - Worker threads do parallel computations
    - e.g., in matrix-vector multiplication: do all the rows in parallel
    - e.g., simulate agents in parallel [think agents in games!]

The Classical Thread Model(1)

a. Three processes each with one thread
b. One process with threads



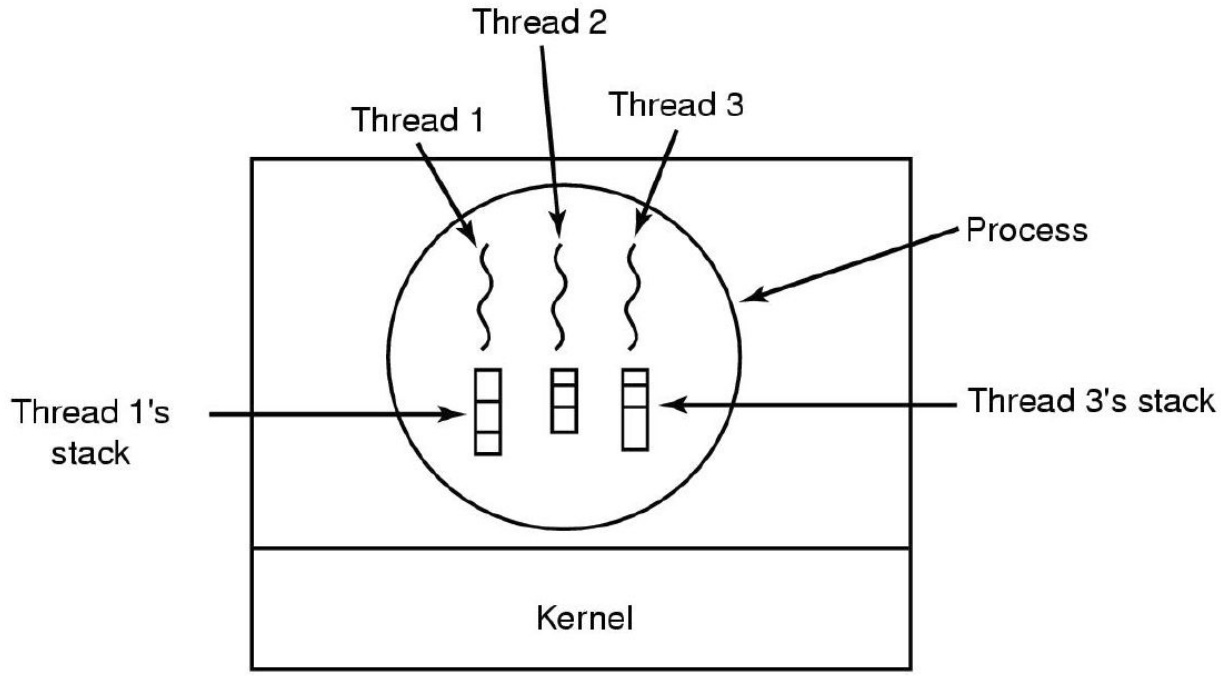
Classical Thread Model (2)

Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

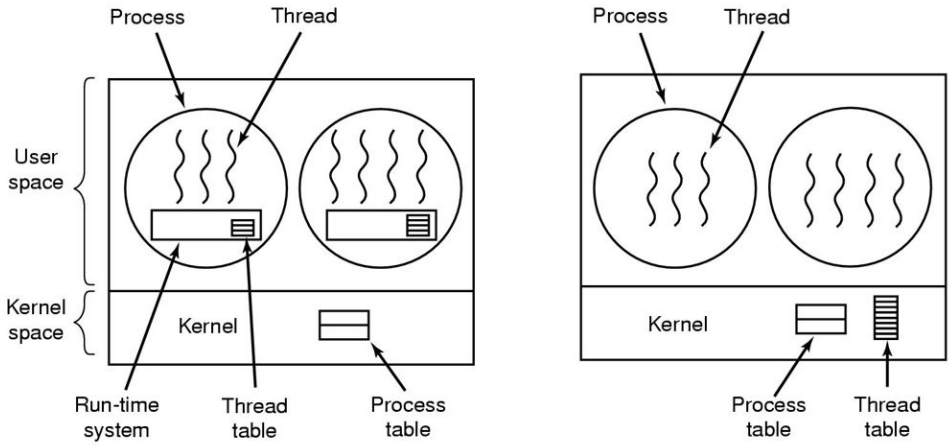
Shared by all threads in a process.

Private to each thread.

Classical Thread Model (3)



Implementing Threads



a. A user-level threads package
b. A threads package managed by the kernel

The Plot

- What you wish to share (the toy) is
    - The variable x
    - it exists somewhere in the shared virtual address space
    - It has no special status (compared to other regions!)
- The protagonist who both want to play with the toy are
    - thread 1
    - thread 2

Your Objective

- Protect the toy(x)
    - Make sure nothing “bad” happens to it
    - In particular, make sure it has the correct value at the end
- Recall
    - x exists in memory
    - So you wish to protect memory (an integer)

Idea

- You cannot protect the memory hold “x”
- But…
- You can specify a protocol for everyone using “x”
    - If a thread wishes to increase “x” …
        - It must grab a specific “lock” [lock]

        - If it has the “lock”, it can do what it wants to x [critical section]
        - When does with x it must release the “lock” [unlock]
    - If it does not have the lock, it must wait

You do not protect x directly
Instead, you discipline the code that touches x

Idea 1 - Is this slide necessary? - Maifi

- Put x in a “mini-prison”
    - If thread 1 wishes to increment “x”
        - It must take “x” out of its prison, increment it, and put it back
    - If thread 2 wishes to increment “x” and the cell is empty…
        - It must wait until “x’ is back in its cell!
- Does this work? [conceptually]
- Does this work? [practically]

32-bit Implication

- Virtual address Space size on 32-bit OS
    - Linux: 2G
    - Windows: 1G
- Typical stack size per thread
    - 8 Megs
- Memory usage goes to
    - Executable: ~ 1 to 50 megs
    - Heap: ~ 1 to 200 megs
    - Stacks: ~ # of threads * 8 → 100 threads yield 800 megs
- Total near the limit of the address space size.

Circumventing the limit?

- Several “ways” to hop along
    - Make smaller stacks! [but beware of recursion]
    - Separate tasks in several process that
        - Communicate via pipes
        - Communicate via shared virtual memory
    - Use a 64-bit OS!
        - Remember 8 megs = 2^{23}
        - Address space size = 2^{64}
            - What’s left is still: 2^{63}
        - How many stacks can you have? ~2^{40}

CSE 3100 Master Notes

Threads and MIMD Architectures

- **MIMD** - Multiple Instruction Multiple Data
- Process has
    - Multiple execution units
    - All executing independently
    - All executing diff. instructions
    - All operating on diff. pieces of data

Threads and MIMD Architecture

- Threads
    - Are OS abstractions that capture computation streams
    - Can be scheduled on a MIMD processor
    - All concurrent threads execute diff. instructions on diff. data

Timesharing

- Idea is simple
    - Diff. pattern of interaction [work / IO] for threads
    - When one thread is “waiting” for an IO to complete…
    - … another thread can use the CPU for some computing.
    - At any one point in time, CPU is used by only ONE thread
- True concurrency
    - There are > 1 CPUs
    - Threads are executing truly in parallel, one on each CPU
    - OS can still preempt threads
    - Useful when # threads >> # cpus! → time sharing is also used!

Threads?

- **definition [wikipedia]** - In computer science, a thread of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler (typically as part of an operating system).

User Level Thread

- Can be implemented on top of OS that does not support threading
- Each process maintains thread table
- Per process run time system for thread switching
- Each process can have its own customized algorithm

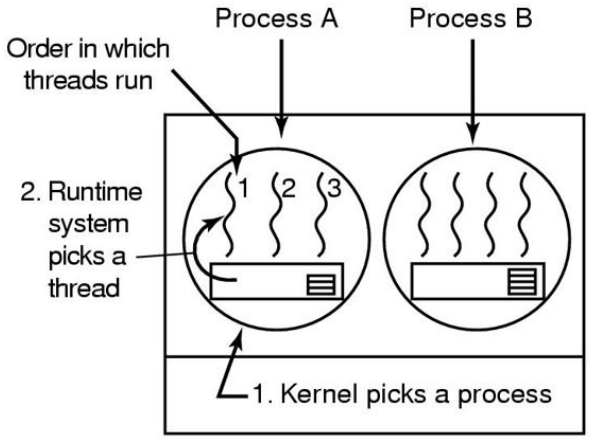
- But
    - Blocking system call in one thread can be a problem

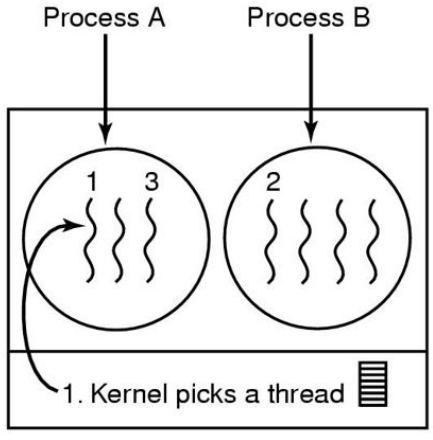
  - Change all block to non-block call by changing OS
  - Within a single process, there is no interrupt. If a thread does not give up voluntarily, other threads in the same will not get a chance
- Threads are mainly for processes that often blocks for I/O
  - User level thread does not make much sense as the goal was to avoid expensive kernel switch. Once switch is done, not much work to switch between threads

Kernel Level Thread

- Kernel has a thread table for all threads
  - Hold thread specific register, stack, state
- Thead recycling
  - Instead of deleting thread data structure, preserve it and reuse it
- But
  - What happens when multithreaded process forks?
  - Signals are per process. Which thread should receive the signal?

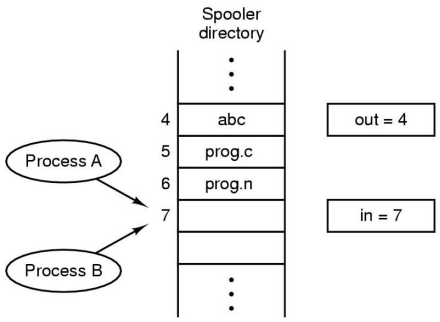
Let's check...





Threads & Castles

- In reality
  - When a process is created there is only 1 living creature (1 thread)
  - The first thread no diff. from other threads
  - All threads have equal rights (unless otherwise specified)



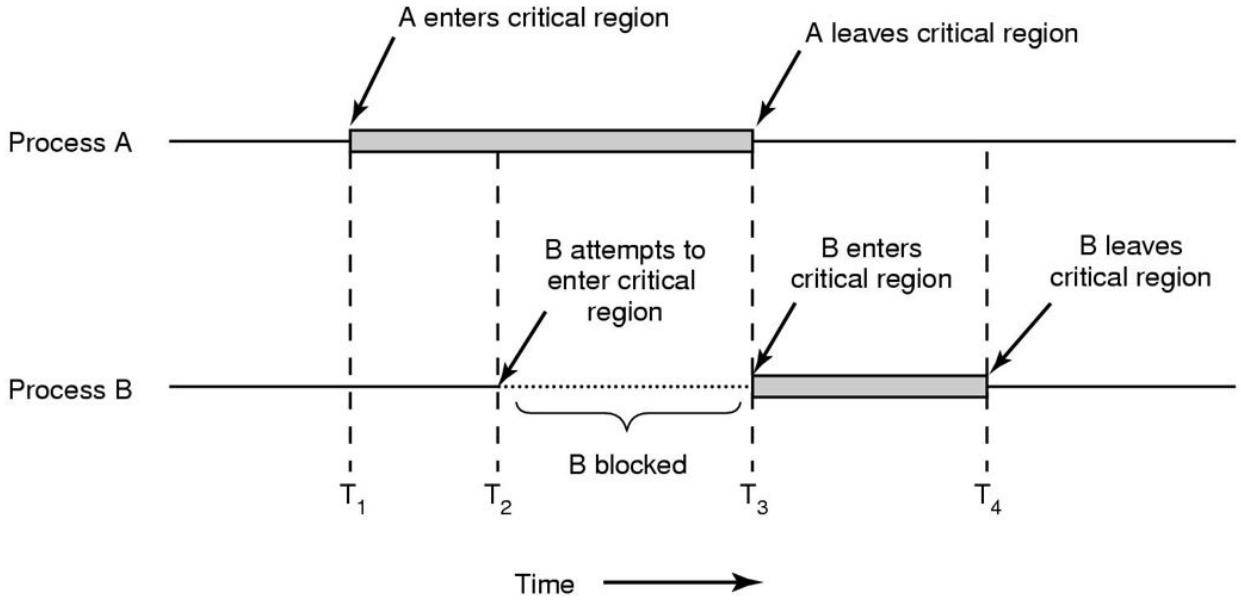
Race Conditions

- Two processes want to access shared memory at the same time

Critical Regions (1)

Conditions required to avoid race condition:

- No two threads may be simultaneously inside their critical regions
- No assumptions may be made about speed or the # of CPUs
- No threads running outside its critical region may block other processes
- No threads should have to wait forever to enter its critical region



Critical Regions (2)

- Mutual exclusion using critical regions.

Lecture 21 - T2: Resource Sharing

Weds. Oct. 23, 2019

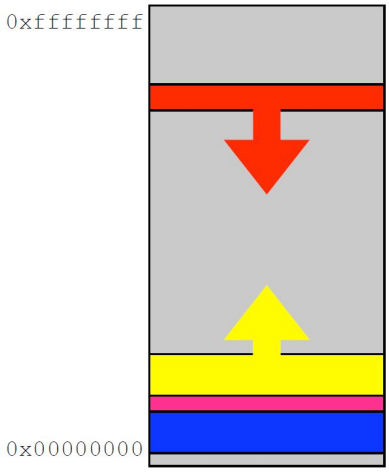
Review

```
#include <pthread.h>
// Compile and link with '-pthread'
```

```
int pthread_create(   pthread_t* thread,
                      pthread_attr_t* attr,
                      void* (*start_routine)(void*),
                      void* arg);
// Terminating itself or return from start_routine()
voidpthread_exit(void *retval);

// avoid zombie threads
int pthread_join(pthread_tthread, void ** retval);
```

Virtual Address Space

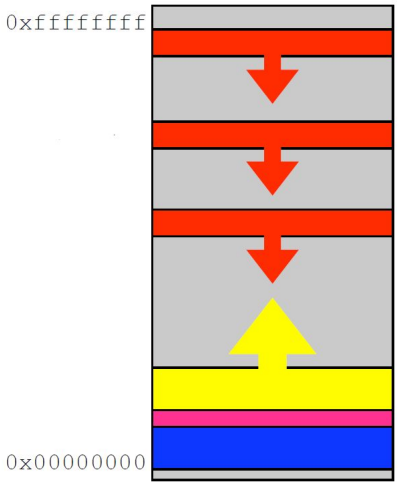
Process with one thread



Each thread gets a stack - Stacks have a max size -- kepss stacks separated to avoid accidental overlap

Process with 3 threads

- **Global data is shared**



32-bit Implication

- Virtual address Space size on 32-bit OS
  - Linux: 2G
  - Windows: 1G
- Typical stack size per thread
  - 8 Megs
- Memory usage goes to
  - Executable: ~ 1 to 50 megs
  - Heap: ~ 1 to 200 megs
  - Stacks: ~ # of threads * 8 → 100 threads yield 800 megs
- Total near the limit of the address space size.

Circumventing the limit?

- Several “ways” to hop along
  - Make smaller stacks! [but beware of recursion]
  - Separate tasks in several process that
    - Communicate via pipes
    - Communicate via shared virtual memory
  - Use a 64-bit OS!
    - Remember 8 megs = 2^{23}
    - Address space size = 2^{64}
      - What’s left is still: 2^{63}
    - How many stacks can you have? ~2^{40}

Advantages of Threads

- Simpler programming model
- Easier to coordinate (shared address space, and data)
- Lighter weight than processes
  - Takes less resources than process to manage
- In case of substantial CPU and I/O, thread improves performance

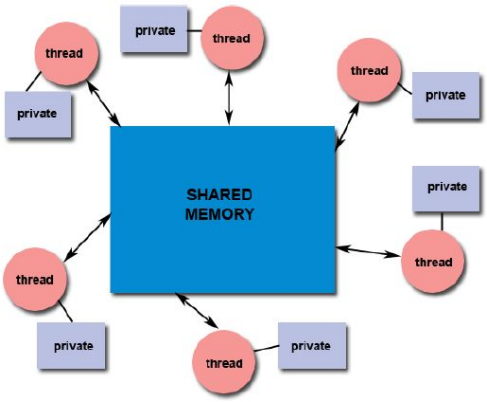
- Light-weight
  - Lower overhead for thread creation
  - Lower context switching overhead
  - Fewer OS resources

Time (sec.) for creating 50,000 processes/threads

Platform	fork()			pthread_create()		
	real	user	sys	real	user	sys
AMD 2.4 GHz Opteron (8cpus/node)	17.6	2.2	15.7	1.4	0.3	1.3
IBM 1.9 GHz POWER5 p5-575 (8cpus/node)	64.2	30.8	27.7	1.8	0.7	1.1
IBM 1.5 GHz POWER4 (8cpus/node)	104.1	48.6	47.2	2.0	1.0	1.5
INTEL 2.4 GHz Xeon (2 cpus/node)	55.0	1.5	20.8	1.6	0.7	0.9
INTEL 1.4 GHz Itanium2 (4 cpus/node)	54.5	1.1	22.2	2.0	1.3	0.7

https://computing.llnl.gov/tutorials/pthreads

- Shared State
  - Simpler programming model
  - Don't need IPC-like mechanism to communicate between threads



The larger, the better.

Platform	MPI Shared Memory Bandwidth (GB/sec)	Pthreads Worst Case Memory-to-CPU Bandwidth (GB/sec)
Intel 2.6 GHz Xeon E5-2670	4.5	51.2
Intel 2.8 GHz Xeon 5660	5.6	32
AMD 2.3 GHz Opteron	1.8	5.3
AMD 2.4 GHz Opteron	1.2	5.3
IBM 1.9 GHz POWER5 p5-575	4.1	16
IBM 1.5 GHz POWER4	2.1	4
Intel 2.4 GHz Xeon	0.3	4.3
Intel 1.4 GHz Itanium 2	1.8	6.4

Example: array sum

- Use two threads to compute the sum of integers in an array

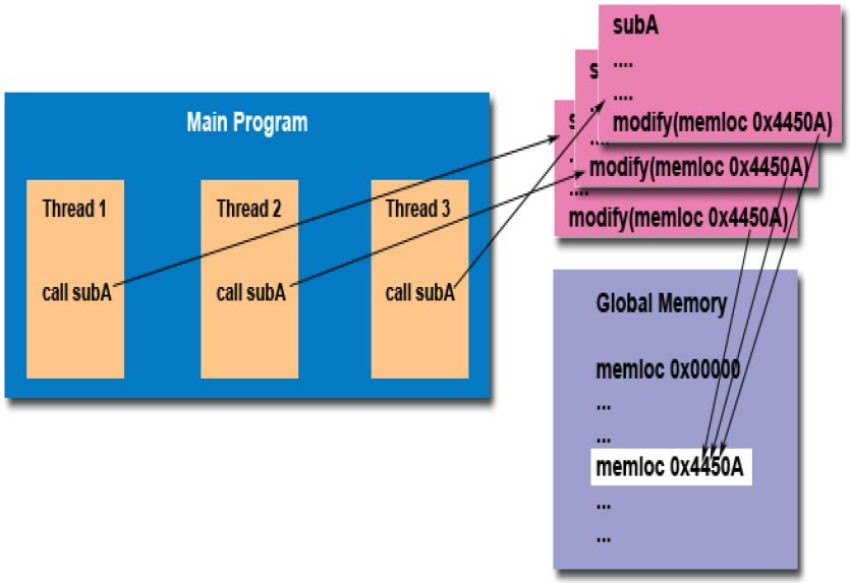
Disadvantages of threads: What if:

- A thread something stupid like
  - A division by zero?
  - Dereferencing a null pointer?
  - Corrupting a block of memory?

- Access a bad file descriptor?

The entire process crashes and burns!

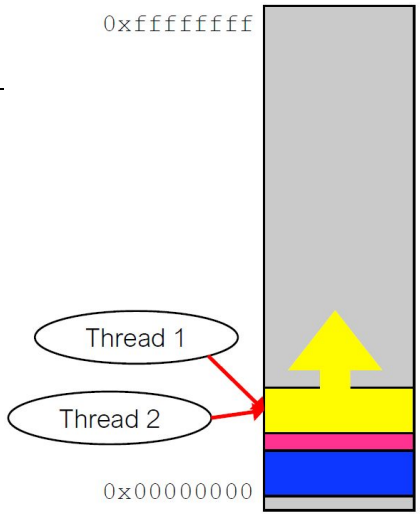
Disadvantages of threads: Shared State.



You may not be aware of shared resources!

Shared data: global, heap, and even local

```
int a[100];
// a, defined outside of functions,
// can be accessed in all threads.

int main(void)
{
      int i;      // stack
      char * p = malloc(1000); // heap
      void * arg = &i;
      // pass p or arg to threads
}
```



However, Sharing is Unsafe - A simple counting program…

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

long count = 0;

void* increase(void *arg){
    long i, inc = *(long *)arg;
    for (i=0; i<inc; i++)
        count++;
    pthread_exit(NULL);
}

int main(int argc, char* argv[]){
    pthread_t tid1, tid2;
    long inc = atol(argc >= 2 ? argv[1] : "100");
    pthread_create(&tid1, NULL, increase, &inc);
    pthread_create(&tid2, NULL, increase, &inc);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    printf("counter is %ld\n", count);
    return 0;
}
```

Example

- Consider the two threads each doing the following

```
int x = 0;
```

```
void increase(int cnt) {
   int i;
   for(i=0;i<cnt;i++)
      x = x + 1;
}
```

```
void increase(int cnt) {
   int i;
   for(i=0;i<cnt;i++)
      x = x + 1;
}
```

What will happen?

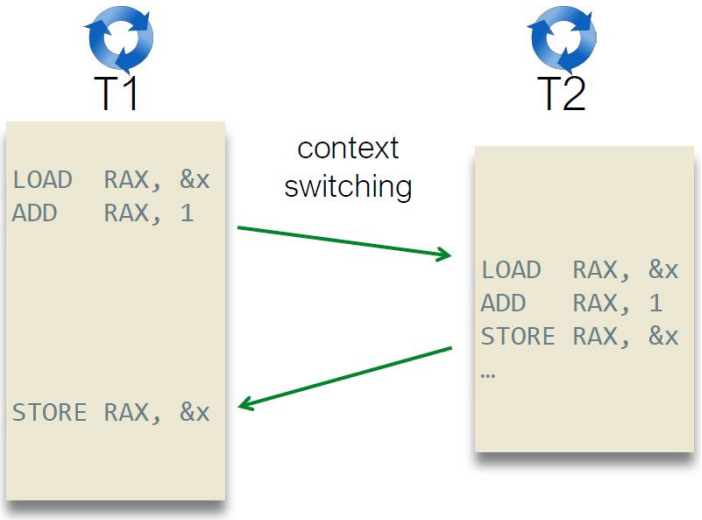
What is happening?

- The addition
  - Becomes more complex in assembly
  - Something like (pseudo-code)

```
void increase(int cnt)
{
   int i;
   for(i=0;i<cnt;i++)
      x = x + 1;
}
```

```
// x = x + 1;

LOAD RAX, &x
ADD  RAX, 1
STORE RAX, &x
```

- Threads execute this concurrently
- Even on a single core, the execution of a thread can be interrupted



Lesson

- Even sharing a single integer can go wrong!
- What to do?
- We need coordination!
    - RULES and PROTOCOLS
    - To establish how share data safely and keep everyone happy

The Road Ahead

- What we will do
    - Define PROTOCOLS and DATA STRUCTURES to safely share
- Examples
    - Mutexes / Spinlocks
    - Conditions
    - Semaphores
    - Barriers
    - Producer / Consumer
    - Reader / Writer
    - …

How unsafe can this be?

- Lots of subtle issues
  - It is very easy to get it wrong
- Good multi-threading programming must be disciplined

Detaching a thread

`int pthread_detach(pthread_t tid)`

- The “parent thread doesn’t need to wait
- A thread can detach another thread
- When detached thread terminates, its resources are automatically released
- A thread can detach itself:

  `pthread_detach(pthread_self());`
- Only threads created as joinable and not detached can be joined

Passing Arguments to Threads -1

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

#define NUM_THREADS  8

struct thread_data {
   int  thread_num;
   char* message;
};

void* PrintHello(void* threadarg) {
   struct thread_data* my_data = (struct thread_data*) threadarg;

   sleep(1 + 5*(my_data->thread_num % 2) );

   printf("Thread #%d: %s  length=%zd\n", my_data->thread_num, my_data->message,
    strlen(my_data->message));

   pthread_exit(NULL);
}
```

Passing Arguments to Threads -2

```
int main(int argc, char* argv[])
{
   pthread_t threads[NUM_THREADS];
   static struct thread_data thread_data_array[NUM_THREADS];
   char* messages[NUM_THREADS];
   int rc, t;
   messages[0] = "English: Hello World!";
   messages[1] = "French: Bonjour, le monde!";
   messages[2] = "Spanish: Hola al mundo";
   messages[3] = "Klingon: Nuq neH!";
   messages[4] = "German: Guten Tag, Welt!";
   messages[5] = "Russian: Zdravstvuyte, mir!";
   messages[6] = "Japan: Sekai e konnichiwa!";
   messages[7] = "Latin: Orbis, te saluto!";

   // continue on the next slide
```

Passing Arguments to Threads -3

```
  for( t=0; t<NUM_THREADS; t++ ) {
      //set up struct for thread t
      thread_data_array[t].thread_num = t;
      thread_data_array[t].message = messages[t];

      printf("Creating thread #%d\n", t);
      rc = pthread_create(&threads[t], NULL, PrintHello,
                  (void*)&thread_data_array[t]);
      if (rc) {
         printf("ERROR; return code from pthread_create() is %d\n", rc);
         exit(-1);
      }
      printf("Detaching thread #%d\n", t);
      rc = pthread_detach( threads[t] ); // detach a thread
      if( rc ){
         printf("ERROR; return code from pthread_detach() is %d\n", rc);
         exit(-1);
      }
   }
   pthread_exit(NULL);
}
```

Create a thread as detached

- When calling pthread_create(), one of the attributes defines whether the thread is joinable or detached

- By default (NULL attribute) threads are created as joinable

```
pthreads_ tid; void * arg     // thread id and argument
pthread_attr_t attr;          // an attribute variable

pthread_attr_init(&tattr);    // initialize with default attributes
// set detach state
pthread_attr_setdetachstate(&tattr, PTHREAD_CREATE_DETACHED);
pthread_create(&tid, &tattr, start_routine, arg);     // create thread
pthread_attr_destroy(&attr);                    // destroy the attribute
```

Practice Exam 2

Mon. Oct. 28, 2019

Practice Exam 2 Starter Code

```
//In this practice exam, we use multiple processes to simulate a word game named hangman.
//In the original hangman game, the number of guesses are limited and some graphical
display of a hangman
//is involved to indicate the progress of the game.
//We simplify the game to not to draw a hangman, and not limit the number of guesses.
//The simplified game works as follows.
//Player one chooses a word and indicates the length of the word
// and let the second player to guess the word. For example, the first player
// shows the following string to indicate that the word has 4 letters.
// ----
//The player two suggests a letter, for example, the letter 'e'.
//The player one responses by displaying the correct guess at the right places in the word.
//For example, player one tells player two the following string
// --ee
//Player two suggests another letter, a letter 'f' this time.
//The player one displays the same string
// --ee
//This is because the letter 'f' is not in the word.
//Player two suggests another letter 't'.
//The first player displays
// t-ee
//Then the second player suggests another letter 'r'.
//Player one displays
// tree
//Now since all the hidden letters are displayed. The game is over.
//In this practice exam, we need to use two processes to simulate the two players.
//Also, we need to use pipes for the communications between the two players.
//To be more specific, the child process is player one; the parent process is player two.
//A user will type guesses from the standard input to play the game.
```

```
// Search TODO to find the location where the code needs to be completed.

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <assert.h>
#include <sys/wait.h>
#include <errno.h>
#include <ctype.h>

#define    PFD_READ    0
#define    PFD_WRITE   1

#define MAX_WORD_COUNT 60000            //we have less than 60000 words
#define MAX_WORD_LENGTH 80              //each word is less than 80 letters

void die(char *s)
{
   if (errno)
      perror(s);
   else
      fprintf(stderr, "Error: %s\n", s);
   exit(EXIT_FAILURE);
}

char words[MAX_WORD_COUNT][MAX_WORD_LENGTH];       //2-d array to hold all the
words
int count = 0;                  //number of words, initilized to 0

//read words from the file to the array words declared above
//also update the number of words (update variable count)
//We could have avoided using global variables. Try to revise it yourself.
void read_file_to_array(char *filename)
{
   FILE *fp;

   //open the file for reading
   fp = fopen(filename, "r");
   if(fp==NULL)
      die("Cannot open the word list file.");

   // TODO
   // make sure when each word is saved in the array words,
   // There is no white space in words, we can use fscanf().
   // We could also use fgets(). Need to remove '\n' at the end.
   fclose(fp);
```

```
}

// write a character to a pipe
void write_char(int pd, char value)
{
   if (write(pd, &value, sizeof(char)) != sizeof(char))
      die("write()");
}

// write a string to FD pd , add '\n' at the end
void write_word(int pd, char *word)
{
   size_t len = strlen(word);

   if (write(pd, word, len) != len)
      die("write()");
   write_char(pd, '\n');
}

// read a char from FD pd and save the result in *pc
// return the return value from read()
int read_char(int pd, char *pc)
{
   return read(pd, pc, sizeof(char));
}

// read a line from FD pd
//
void read_word(int pd, char buffer[], int sz)
{
   char c;
   int count = 0;

   while (read_char(pd, &c) > 0)
   {
      if (count >= sz)
         die("line is too long in read_word().");
      if (c == '\n') {
         buffer[count] = 0;
         return;
      }
      buffer[count ++] = c;
   }
   // could handle error better
   die("read() faild in read_word");
}
```

```
//check if the character guess is in the word
//if it is in the word, update the string so_far in the right places
//for example, if guess is 'e', so_far is "----", and word is 'tree'
//then so_far will be updated to be '--ee'
//if guess is 't', so_far is '--ee', and word is 'tree'
//then so_far will be updated to be 't-ee'
// Return value:
// 0:   guess is not in the word
// 1:   guess is in the word
int check_guess(char guess, char *so_far, const char *word)
{
   // TODO
   return 0;
}

int main(int argc, char* argv[])
{

   if(argc!= 2)
   {
      printf("Usage: %s seed\n", argv[0]);
      return -1;
   }
   int seed = atoi(argv[1]);
   assert(seed > 0);

   int pdp[2];
   //pipe creation
   if(pipe(pdp) == -1)
   {
      perror("Error.");
      return -1;
   }

   int pdc[2];
   //pipe creation
   if(pipe(pdc) == -1)
   {
      perror("Error.");
      return -1;
   }

   pid_t pid;
   pid = fork();
   if(pid == 0)
   {
      // TODO
```

```
    //  close some file descriptors

    // read the list in child process
    read_file_to_array("dict.txt");

    char *my_word;
    char so_far[MAX_WORD_LENGTH];

    srand(seed);
    my_word = words[rand() % count];
    fprintf(stderr, "Child: debugging: the word is %s\n", my_word);

    // Note that so_far should have enough space
    size_t  len = strlen(my_word);
    for(int i = 0; i<len; i++)
       so_far[i] = '-';
    so_far[len] = 0;

    //TODO
    //repeatly doing the following
    //      send so_far to parent
    //      receive a guess (a character) from parent
    //      exit from the loop if it was not successful
    //      check_guess
    //Do some clean up before exit from the process
    return 0;
}
else
{
    char guess;
    char so_far[MAX_WORD_LENGTH];

    // TODO
    // close some file descriptors
    // Then do the following in a loop:
    //      read a word from child
    //      print it to stdout
    //      if there is no '-', exit from the loop
    //      read a character from stdin until a ltter is found
    //      report error if EOF found.
    //      turn the character to lower case
    //      send it to child
    // close file descriptors before exit from the process

}

//wait for the child process to finish
```

```
    waitpid(pid, NULL, 0);
    return 0;
}

//below is a sample output
//it can also be found in the file sample-output.txt
/*
   ./hangman 8
   To help debugging: the word is pride
   -----
   e
   ----e
   a
   ----e
   t
   ----e
   i
   --i-e
   r
   -ri-e
   p
   pri-e
   d
   pride
   The word is pride.
   */
```

Lecture 23 - T2: Mutual Exclusion (mutex)

Weds. Oct. 30, 2019

Review

- So far
    - You learned how to create threads (pthread_create)
    - You learned how to terminate thread execution (pthread_exit)
    - You learned how to wait on threads (pthread_join)
- You can carry out independent computations with threads
- However, sharing is a problem
    - Remember the shared counter?

What if the shared counter is the balance on your bank account?

CSE 3100 Master Notes

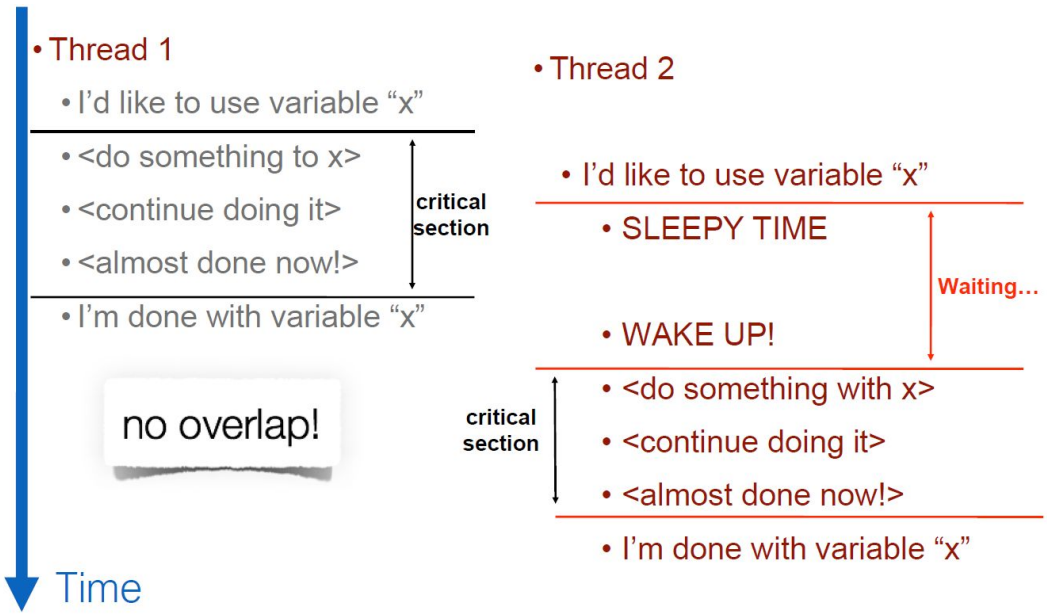
Overview

- Basic Sharing
  - Mutual exclusion
  - Critical selection
  - Mutex
- Application
  - Concurrent access to shared data structures
  - Counter arrays
  - ...

Mutual Exclusion

- Objective
  - Protect shared resources
  - Only one thread can access the resources
- Protocol
  - A set procedures for accessing shared resources
- Example:
  - Lock the resource while using it. Cannot lock if it is already locked
  - Wait if the resource is already lock
- Fact
  - Ask everyone nicely, and expect that everyone behaves

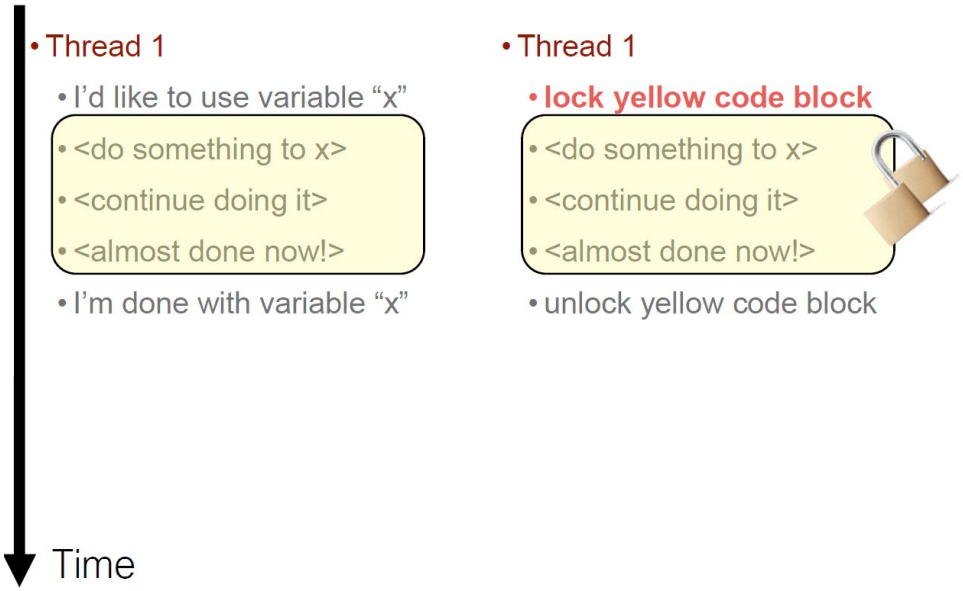
Protocol



Race to the critical section

- Multiple threads race each other to get to critical section
    - Critical section is a code segment that accesses shared resources
- One of them “wins” the race
    - Winner
        - acquire a “lock” first
        - executes critical section
        - release the lock (unlock)
    - Loser
        - waits as the resource is locked
        - wakes up when it is its turn to get the lock
        - executes critical section
        - releases the lock

Protocol



Mutex

- The Lock is a POSIX abstraction
    - Called a **Mutex** (for **MUT**ual **Ex**clusion)
    - Provided by the operating system
- Semantics
    - At most one thread can acquire the lock at any time
    - If a thread “loses” a race, it **falls asleep**
    - Sleepy threads **wake up** when the lock is released

pthread Mutex types and API

```
pthread_mutex_t; // define a mutex
```

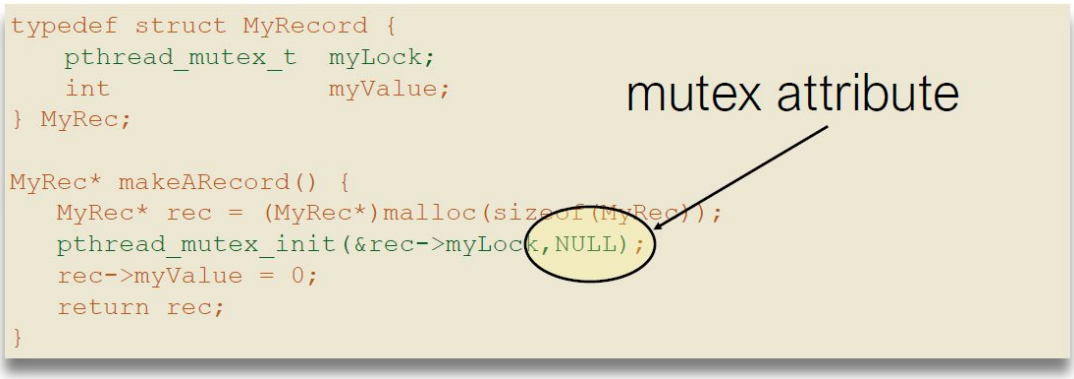
- Functions
  - Initialize the Mutex
  - Destroy a Mutex we no longer need
  - Lock a Mutex
  - Unlock a Mutex
  - [Attempt to Lock a Mutex]

Creation

```
# include <pthread.h>
int pthread_mutex_init(pthread_mutex_t * mutex,
                       const pthread_mutexattr_t * attr);
```

- Initialize a mutex (i.e., allocate OS resources
- Return 0 on success

- Example



Destruction

```
# include <pthread.h>
int pthread_mutex_destroy(pthread_mutex_t * mutex);
```

- Release resources for lock
- Return 0 on successes

- Example

```
typedef struct MyRecord {
    pthread_mutex_t  myLock;
    int              myValue;
} MyRec;

void freeARecord(MyRec* rec) {
   pthread_mutex_destroy(&rec->myLock);
   free(rec);
}
```

Lock / Unlock

```
# include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t * mutex);
int pthread_mutex_unlock(pthread_mutex_t * mutex);
```

- Enter/leave the critical sectoin. Wait (asleep!) if mutex is locked
- Return Value 0 on success (=/= 0? something went horribly wrong)

- Example

```
typedef struct MyRecord {
    pthread_mutex_t  myLock;
    int              myValue;
} MyRec;

void incrementRecordValue(MyRec* rec) {
   pthread_mutex_lock(&rec->myLock);
   rec->myValue = rec->myValue + 1;
   pthread_mutex_unlock(&rec->myLock);
}
```

Key Observations

- Lock and unlock some pairs
- You need ONE MUTEX per “thing” you wish to protect
    - That’s why we package the integer & the mutex in a struct
    - The mutex protects just that integer
    - And nothing else!
- Lock is **BLOCKING**
- Critical sections should be **short**
    - Why?
- Locking incurs a **big** cost
    - Why?

Example 1

- Fix the example we had issue

Example 2: Counter ADT

- A shared counter
    - That was the driving example
    - Make an ADT pairing mutex and counter
    - Have functions to manipulate the ADT
        - increase
        - decrease
        - reset to Zero

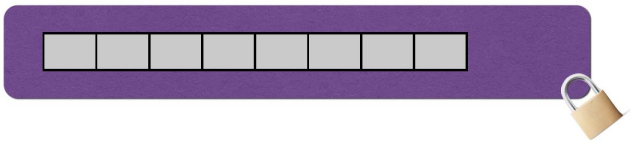
See the Demo Code under t3.counter-adt

Example 3: array

- Sharing an array
- Key decision: Locking Granularity
    - Lock the entire structure?
    - Lock individual parts?

Option 1

- Idea
    - Wrap up the array in a structure
    - Have a single lock for the whole thing
- Issues?



Option 2

- Idea
    - Wrap up each value in a structure (value + lock)
    - Make an array of structures
- Issues?

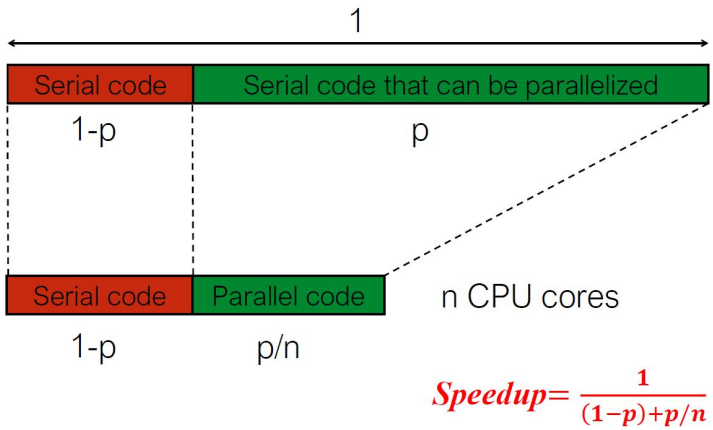
mutex_trylock()

```
#include <pthread.h>
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

- If the mutex is not locked, it will succeed and lock

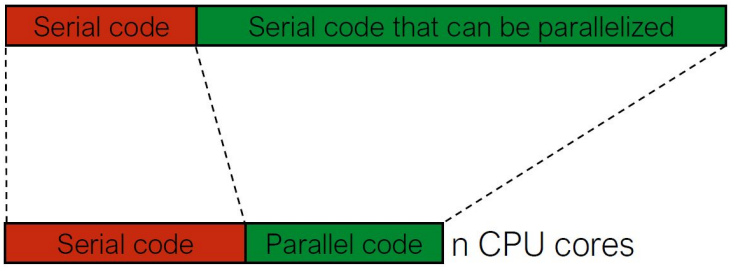
    - If the mutex is already locked
        - It will NOT block
        - It will return an error code
- Purpose
    - Mix breed. It allows polling before locking
        - sometimes, it is fine to use the shared resource later

Amdahl's Law



$$Speedup = \frac{1}{(1-p)+p/n}$$

Practical speedup lower due to overhead



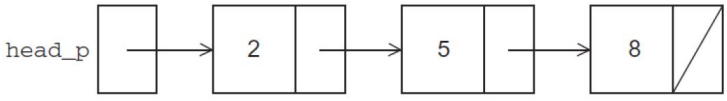
Mutex Attributes?

- A value of type pthread_mutexattr_t
    - An optimal argument to create create a mutex
- APIs to
    - Initialize attribute record (pthread_mutex_attr_init())
    - destroy attribute record (pthread_mutexattr_destroy())
    - modify an attribute in attribute record (set type, get type, etc.)
- Key property: mutex TYPE
    - How to deal with recursive lock?
    - What if other threads try to unlock?

Mutex Type (from the DOC)

- PTHREAD_MUTEX_NORMAL
  - This type of mutex does not detect deadlock. A thread attempting to relock this mutex without first unlocking it will deadlock. Attempting to unlock a mutex locked by a different thread results in undefined behaviour. Attempting to unlock an unlocked mutex results in undefined behaviour.
- PTHREAD_MUTEX_ERRORCHECK
  - This type of mutex provides error checking. A thread attempting to relock this mutex without first unlocking it will return with an error. A thread attempting to unlock a mutex which another thread has locked will return with an error. A thread attempting to unlock an unlocked mutex will return with an error.
- PTHREAD_MUTEX_RECURSIVE
  - A thread attempting to relock this mutex without first unlocking it will succeed in locking the mutex. The relocking deadlock which can occur with mutexes of type PTHREAD_MUTEX_NORMAL cannot occur with this type of mutex. Multiple locks of this mutex require the same number of unlocks to release the mutex before another thread can acquire the mutex. A thread attempting to unlock a mutex which another thread has locked will return with an error. A thread attempting to unlock an unlocked mutex will return with an error.
- PTHREAD_MUTEX_DEFAULT
  - Attempting to recursively lock a mutex of this type results in undefined behaviour. Attempting to unlock a mutex of this type which was not locked by the calling thread results in undefined behaviour. Attempting to unlock a mutex of this type which is not locked results in undefined behaviour. An implementation is allowed to map this mutex to one of the other mutex types.

Sorted Linked List Example

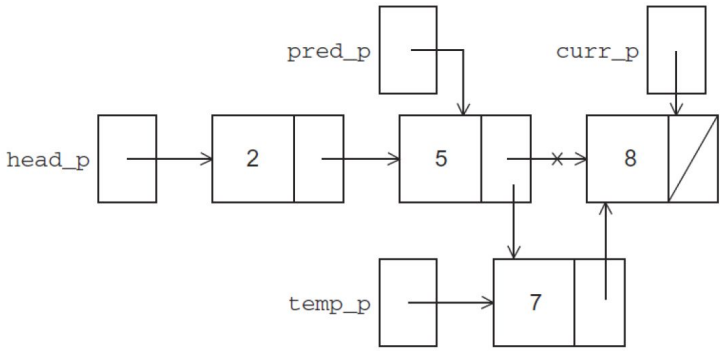



```
typedef struct list_node_s {
   int data;
   struct list_node_s* next;
} LNode;
```

Testing Membership

```
int member( int value, LNode* head_p) {
   LNode* curr_p = head_p;

   while( curr_p != NULL && curr_p->data < value )
      curr_p = curr_p->next;

   if( curr_p == NULL || curr_p->data > value )
      return 0;
   else
      return 1;
}
```

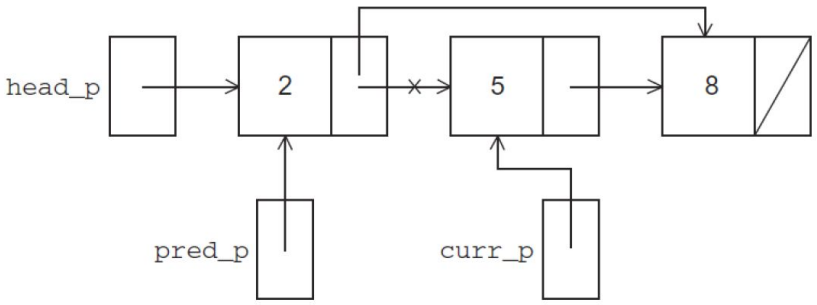
Inserting a new value



Inserting a new value

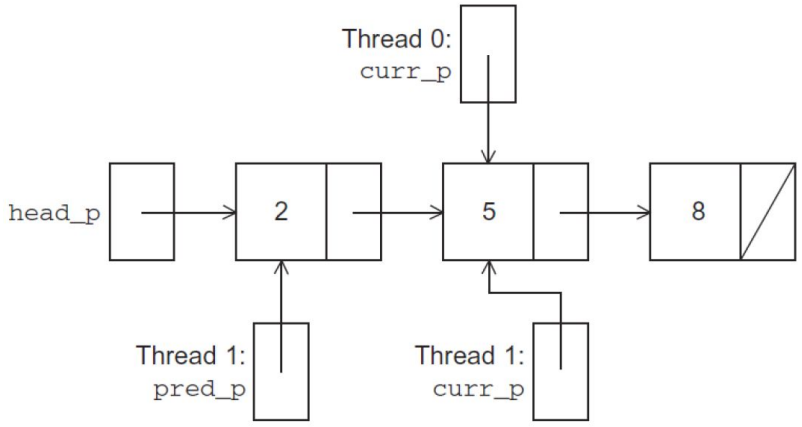
```
int insert( int value, LNode** head_pp) {
   LNode* curr_p = *head_pp;
   LNode* pred_p = NULL;
   LNode* temp_p;

   while( curr_p != NULL && curr_p->data < value ) {
      pred_p = curr_p;
      curr_p = curr_p->next;
   }
   if( curr_p == NULL || curr_p->data > value ) {
      temp_p = (LNode*)malloc( sizeof(LNode) );
      temp_p = value;
      temp_p = curr_p;
      if(pred_p == NULL)  /* new first node */
         *head_pp = temp_p;
      else
         pred_p->next = temp_p;
      return 1;
   } else  /* value already in list */
      return 0;
}
```

Deleting a value




```
int delete( int value, LNode** head_pp) {
   LNode* curr_p = *head_pp;
   LNode* pred_p = NULL;

   while( curr_p != NULL && curr_p->data < value ) {
      pred_p = curr_p;
      curr_p = curr_p->next;
   }
   if( curr_p == NULL || curr_p->data > value ) {
      if(pred_p == NULL) {  /* deleting first node in list */
         *head_pp = curr_p->next;
      } else {
         pred_p->next = curr_p->next;
      }
      free(curr_p);
      return 1;
   } else  /* value not in list */
      return 0;
}
```

Simultaneous access by two threads?



Solution #1

- Lock the entire list any time attempts to access it
    - A call to each of the three functions protected by a mutex e.g.,

```
pthread_mutex_lock(&list_mutex);
member( value, head_p);
pthread_mutex_unlock(&list_mutex);
```

- Issues?

Solution #2

- Instead of locking the entire list, we cold try to lock individual nodes
    - A “finer-grained” approach

```
typedef struct list_node_s {
   int data;
   struct list_node_s* next;
   pthread_mutex_t mutex;
} LNode;
```

Implementation of member with one mutex per node

```
int member( int value, LNode* head_p) {
  LNode* curr_p;

  pthread_mutex_lock( &head_p_mutex );
  curr_p = head_p
  while( curr_p != NULL && curr_p->data < value ) {
    if( curr_p->next != NULL )
      pthread_mutex_lock( &(curr_p->next->mutex) );
    if( curr_p == head_p )
      pthread_mutex_unlock( &head_p_mutex );
    pthread_mutex_unlock( &(curr_p->mutex) );
    curr_p = curr_p->next;
     }
```

```
  if( curr_p == NULL || curr_p->data > data ) {
    if( curr_p == head_p )
      pthread_mutex_unlock( &head_p_mutex );
    if( curr_p != NULL )
      pthread_mutex_unlock( &(curr_p->mutex) );
    return 0;
  } else{
    if( curr_p == head_p )
      pthread_mutex_unlock( &head_p_mutex );
      pthread_mutex_unlock( &(curr_p->mutex) );
    return 1;
  }
}
```

Issues

- More complex to implement
- More memory to store the list
    - One mutex per node
- Slower
    - Each time a node is accessed, a mutex must be locked and unlocked
- Using more than on lock creates opportunities for **DEADLOCK**!

Lecture 24 - T3: POSIX Condition

Mon. Nov. 4, 2019

Mutex Review

- Mutex
  - One has to get the lock before entering critical section
  - Yields exclusive access to a resource
  - If a thread cannot get the lock, it waits …
- Example of two threads sharing data
  - Thread 1 computes a result
  - Thread 2 displays the result on UI when computation done
- Question
  - How does thread 2 know the result is ready?

Producer-Consumer with Bounded Buffer

- Classic problem
  - Producer(s) put things into a shared buffer
  - Consumer(s) take them out!

Problem Constants

- Mutual exclusion
  - Only one thread can manipulate the buffer at any time
    - Buffer is shared!
- Synchronization
  - Consumer(s) must WAITif buffer is empty
  - Producer(s) must WAITif buffer is full

Can it be done with only a lock ?

Well… Sorta… Consumer in pseudo-code

```
do {
      Lock the buffer// Buffer is shared ! May have to wait
      if buffer is not empty
            Fetch data in the buffer
```

```
      Unlock the buffer
} while (data is not fetched) // May fail, so try in a loop
Process the data
```

Consumer in C-like code

```
pthread_mutex_t buffer_lock;
// Consumer fetches data from a buffer, one each time
      fetch = 0
      do {
            pthread_mutex_lock(&buffer_lock);      // Get the lock for
the buffer
            if (nElements > 0){
                  data = fetch_from_buffer();
                  fetched = 1;
            }
            pthread_mutex_unlock(&buffer_lock);
      } while (fetched == 0);
      // Continue to process data
```

What happens if the buffer is empty?

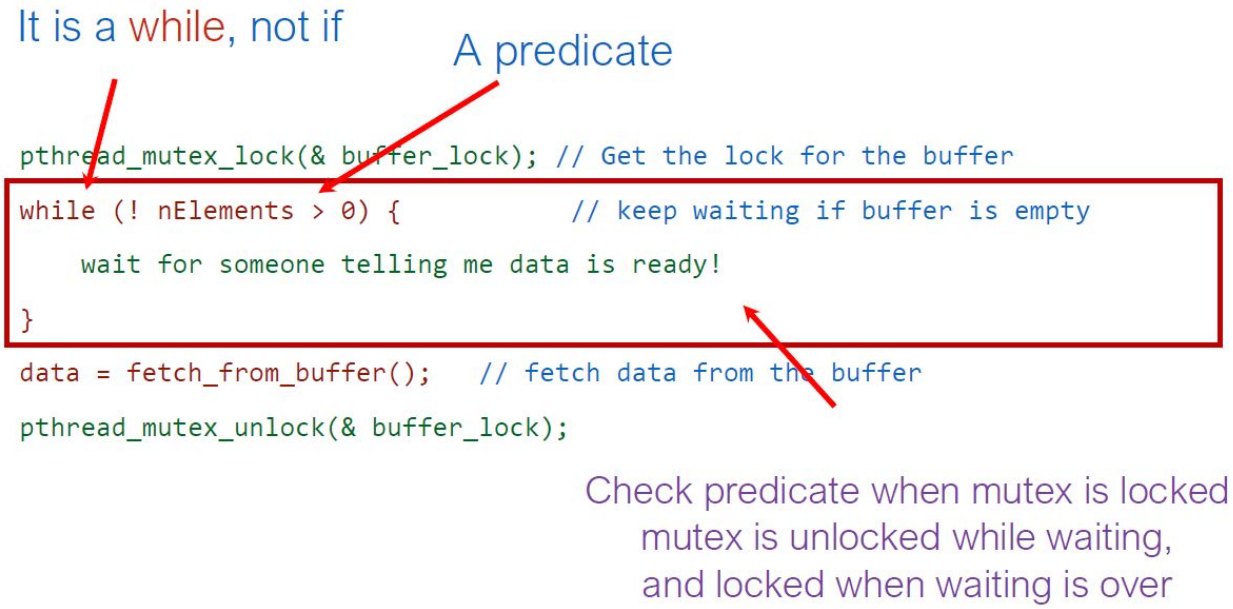
Condition variable?

- A handshake mechanism to say: “There is data for you to use”

Consumer waits on condition

```
pthread_mutex_tbuffer_lock;
// Consumer fetches data from a buffer, one each time.
pthread_mutex_lock(& buffer_lock);// Get the lock for the buffer
while (! nElements> 0) { // keep waiting if buffer is empty
      wait for someone telling me data is ready!
}
data = fetch_from_buffer(); // fetch data from the buffer
pthread_mutex_unlock(& buffer_lock);
// Continue to process data
```

Consumer waits on condition - 2



- Predicate: Logical expression describing the property of an object (or the state) the program needs
    - Examples: the buffer is full, or the buffer is not empty

POSIX condition variable

- Second and last core synchronization primitive
    - All others (barriers, read-write locks, …) can be implemented using mutexes and condition variables
- Always paired with a mutex
    - Its access needs to be **mutex** protected!
- If a predicate is not true, a thread can wait on a condition variable
    - Other threads can **signal** on the condition variable when the **predicate has changed**
        - Threads must still check if the predicate is true due to spurious wake-ups

pthreads_cond API

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

```
pthread_cond_t cond;        // Define a condition variable
int pthread_cond_init(pthread_cond_t * cond,
                      const pthread_condattr_t * attr);
int pthread_cond_destory(pthread_cond_t * cond);
int pthread_cond_wait(pthread_cond_t * cond, pthread_mutex_t *mutex);
int pthread_cond_signal
```

```
int pthread_cond_broadcast
```

Tips

- The mutex protects
    - Thes hared and the state (predicate P)
- The condition variable
    - lock the mutex before waiting
        - the mutex is unlocked automatically while waiting
        - Automatically and atomically re-lock the mutex when waiting up
- The predicate P is checked in a while loop!
    - The while loop returns to waiting when P is false
        - Why? (You will see in a moment)

Typical Structure of using mutex and condition

```
pthread_mutex_t mutex;
pthread_cond_t cond;

pthread_mutex_lock(&mutex);
while (! predicate){
      pthread_cond_wait(&cond, &mutex);
      do_something();
}
access_shared_resources();                    // safe access the share
resources
pthread_mutex_unlock(&mutex);

// continue to perform other tasks
```

Example setup

- Assumptions
    - A buffer that can hold only one item
    - A predicate P, indicating if the data is ready for consumer
    - A POSIX condition variable
    - A POSIX mutex
- Threads
    - T1: producer T2: consumer
    - Can be many threads: 1 .. k

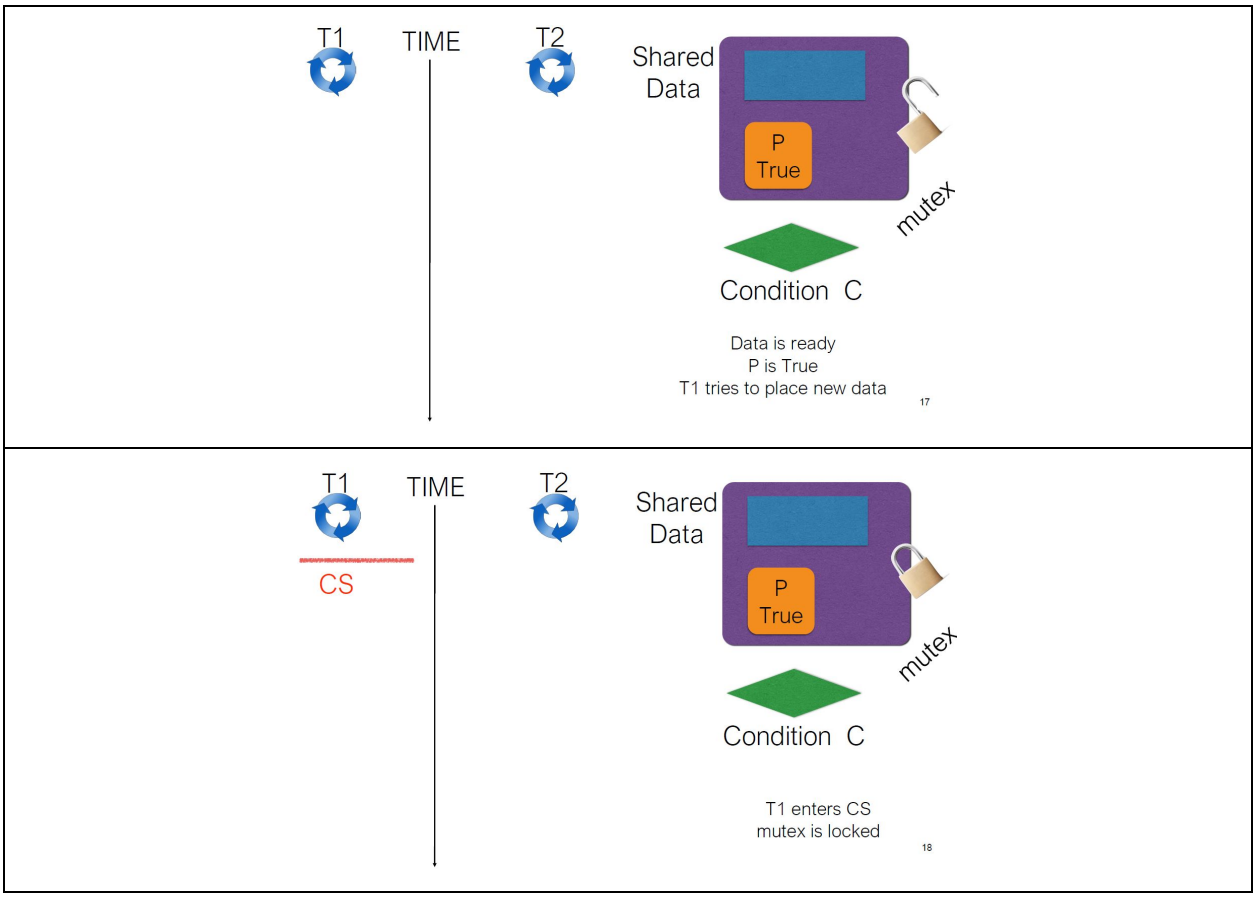
Two scenarios

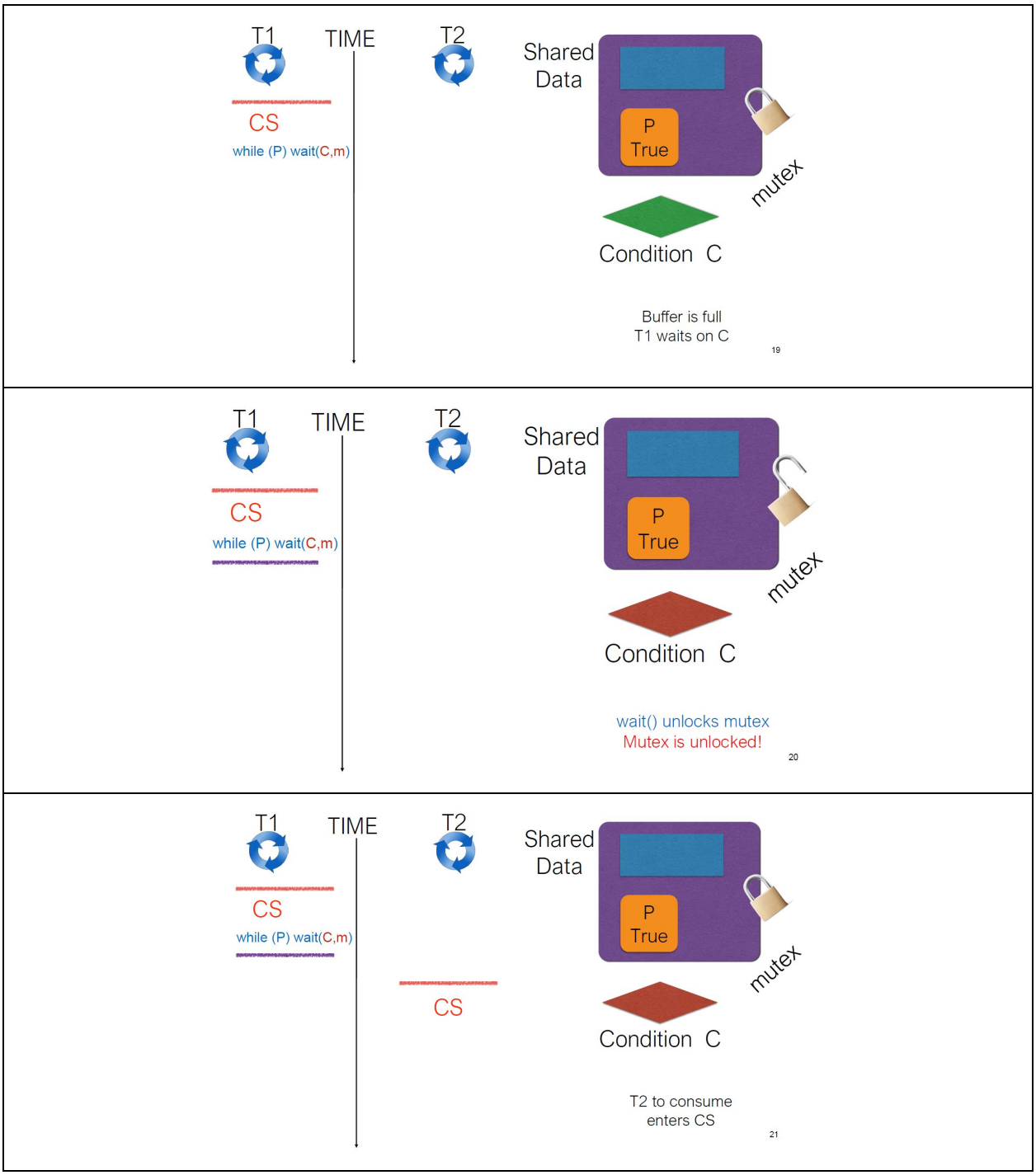
- Scenario 1
  - The producer gets to produce…
  - ...before the consumer consumes the previous one
- Scenario 2
  - The consumer is eager and tries to consume…
  - ...before the producer gets a chance to produce

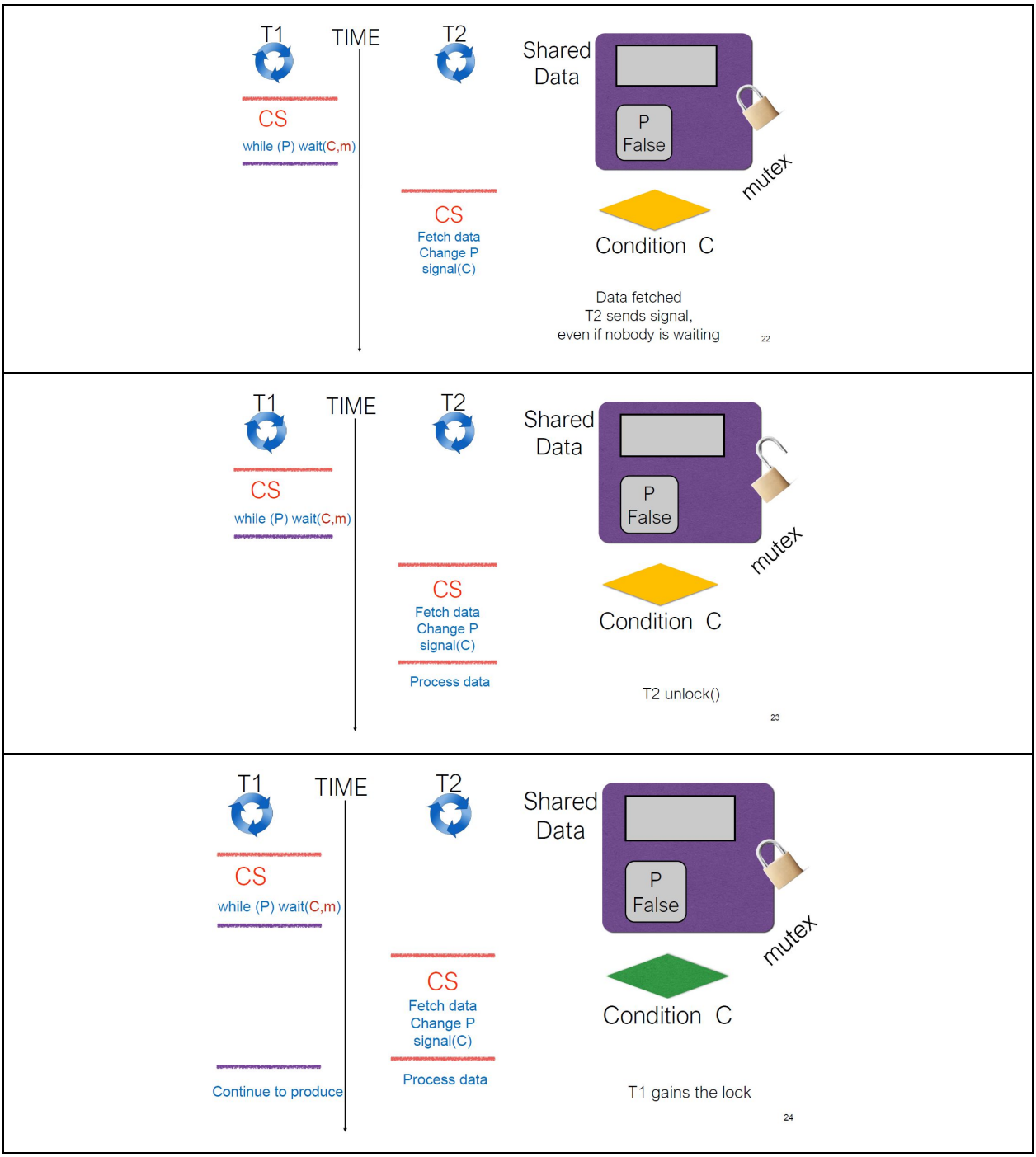
Scenario 1: eager producer

- Producer tries to produce more, but the buffer is full

Storyboard



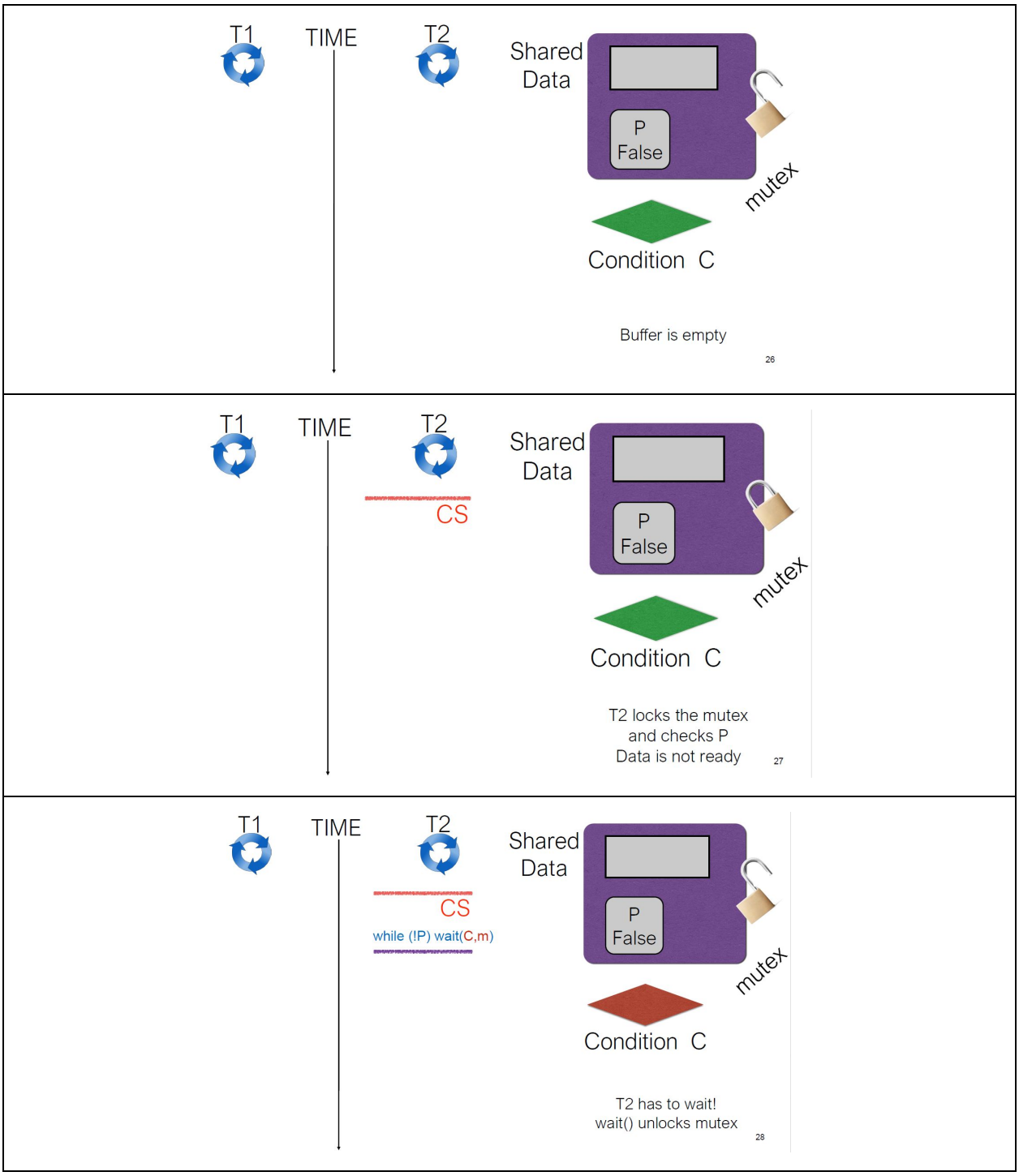

T1
TIME
T2
CS
while (P) wait(C,m)
Shared Data
P
True
mutex
Condition C
Buffer is full
T1 waits on C
19
T1
TIME
T2
CS
while (P) wait(C,m)
Shared Data
P
True
mutex
Condition C
wait() unlocks mutex
Mutex is unlocked!
20
T1
TIME
T2
CS
while (P) wait(C,m)
CS
Shared Data
P
True
mutex
Condition C
T2 to consume
enters CS
21

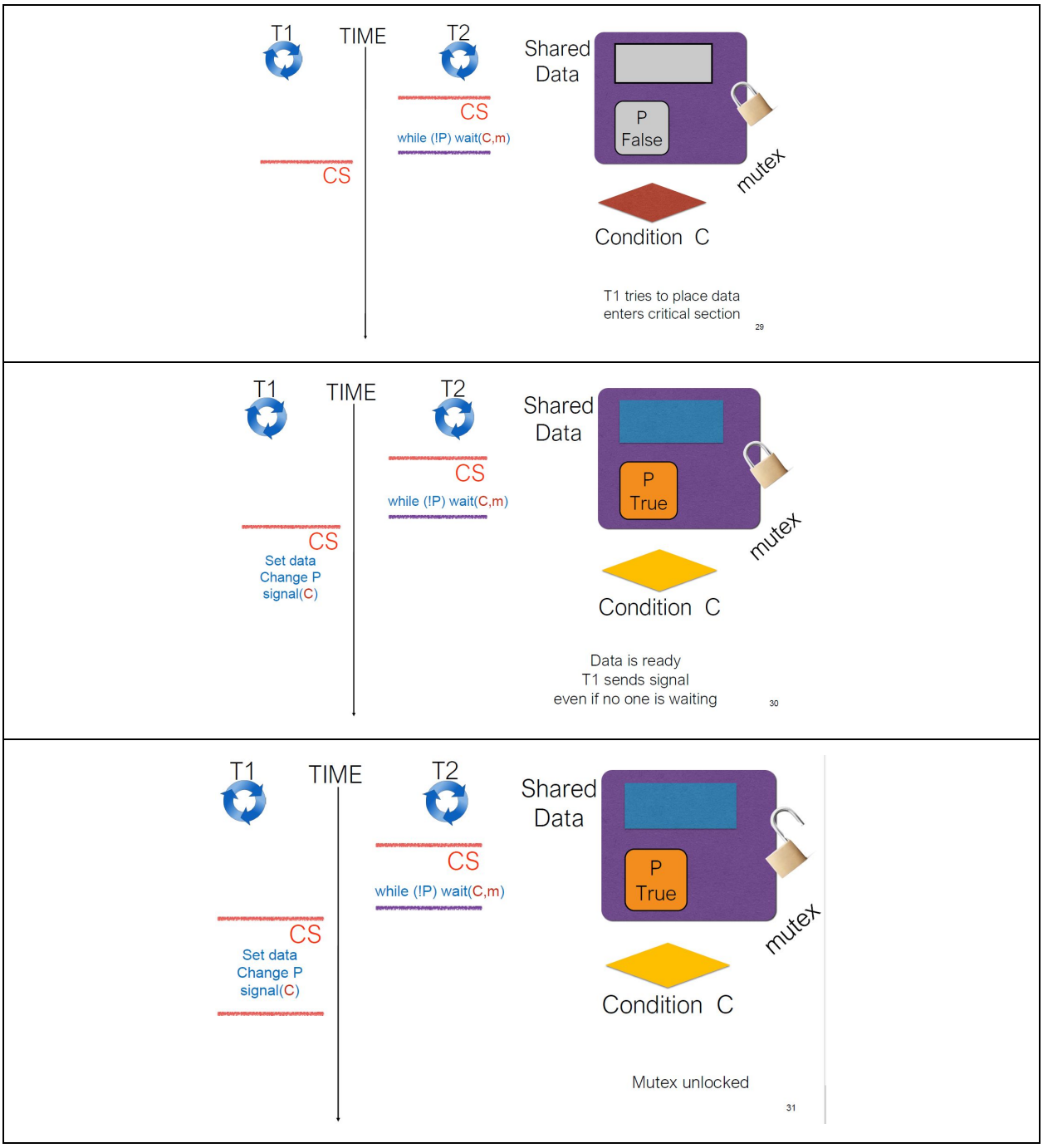


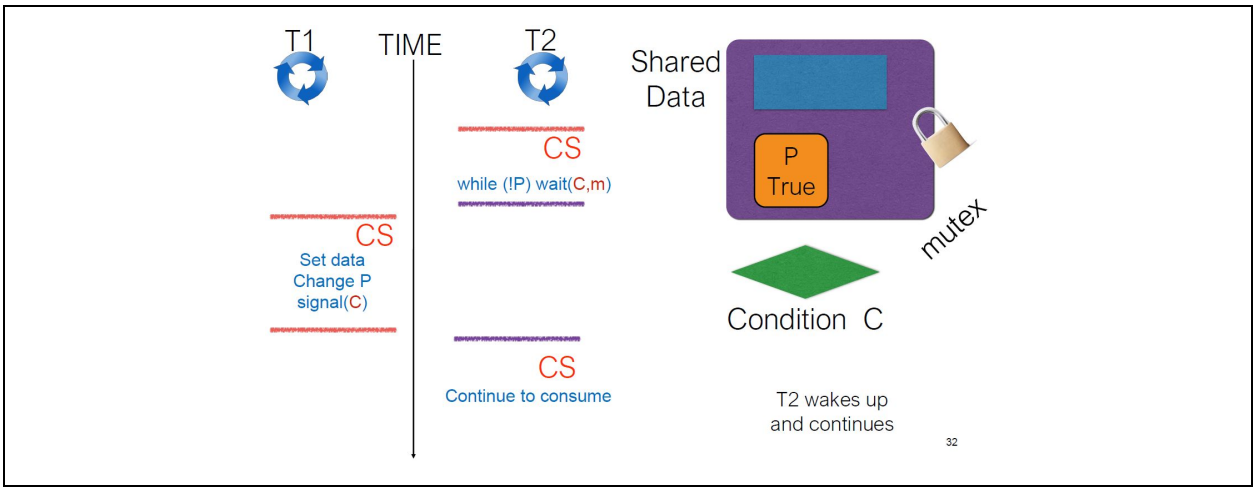
Scenario 2 (Eager consumer)

- Consumer tries to fetch data before they are produced

Storyboard


T1
TIME
T2
Shared Data
P
False
mutex
Condition C
Buffer is empty
26
T1
TIME
T2
CS
Shared Data
P
False
mutex
Condition C
T2 locks the mutex
and checks P
Data is not ready
27
T1
TIME
T2
CS
while (!P) wait(C,m)
Shared Data
P
False
mutex
Condition C
T2 has to wait!
wait() unlocks mutex
28


T1
TIME
T2
CS
while (!P) wait(C,m)
CS
Shared Data
P
False
mutex
Condition C
T1 tries to place data
enters critical section
29
T1
TIME
T2
CS
while (!P) wait(C,m)
CS
Set data
Change P
signal(C)
Shared Data
P
True
mutex
Condition C
Data is ready
T1 sends signal
even if no one is waiting
30
T1
TIME
T2
CS
while (!P) wait(C,m)
CS
Set data
Change P
signal(C)
Shared Data
P
True
mutex
Condition C
Mutex unlocked
31



Signal vs. Broadcast

`pthread_cond_signal(pthread_cond_t*cond);`

- Wakes up **one waiting thread** in the condition

`pthread_cond_broadcast(pthread_cond_t*cond);`

- Wakes up **all waiting threads** in the condition
- Yet, only one of the waiting threads can grab the mutex
    - Go back to sleep if it fails

Caveat: a thread calling wait **after** the broadcast will not wake up (even if others in the process of waking up are still stuck in the condition variable —e.g., waiting on getting the mutex lock —)

Producer in pseudo-code

```
Prepare data
do {
      Lock the buffer// May have to wait
      if buffer is not full// For bounded buffer
            Put data in the buffer
      Unlock the buffer
} while (data is not placed in the buffer)
```

Producer in C-like code

```
pthread_mutex_tbuffer_lock;
// Producer adds data in a buffer, one each time.
      …… // Prepare data here
      added = 0;
      do {
            pthread_mutex_lock(& buffer_lock);// Get the lock for the
      buffer
```

```
            if (nElements< BUF_SIZE) { // If the buffer is not full
                  add_to_buffer(data); // Add data into the buffer
                  added = 1;
            }
            pthread_mutex_unlock(& buffer_lock);
      } while (added == 0);
```

What happens if the buffer is full ?

Producer-Consumer Using Condition Variables

```
pthread_cond_t cond_queue_empty, cond_queue_full;
pthread_mutex_t task_queue_cond_lock;
int task_available;
/* other data structures here */
main() {
  /* declarations and initializations */
  task_available = 0;
  pthread_init();
  pthread_cond_init(&cond_queue_empty, NULL);
  pthread_cond_init(&cond_queue_full, NULL);
  pthread_mutex_init(&task_queue_cond_lock, NULL);
  /* create and join producer and consumer threads */
}
```

```
void *producer(void *producer_thread_data)
{
  int inserted;
  while (!done())
  {
    create_task();
    pthread_mutex_lock(&task_queue_cond_lock);
    while (task_available == 1)
      pthread_cond_wait(&cond_queue_empty,&task_queue_cond_lock);
    insert_into_queue();
    task_available = 1;
    pthread_cond_signal(&cond_queue_full);
    pthread_mutex_unlock(&task_queue_cond_lock);
  }
}
```

```
void *consumer(void *consumer_thread_data)
{
  while (!done())
  {
     pthread_mutex_lock(&task_queue_cond_lock);
     while (task_available == 0)
         pthread_cond_wait(&cond_queue_full,&task_queue_cond_lock);
    my_task = extract_from_queue();
    task_available = 0;
    pthread_cond_signal(&cond_queue_empty);
    pthread_mutex_unlock(&task_queue_cond_lock);
    process_task(my_task);
  }
}
```

Lecture 25 - T5: More Threads Synchronization

Weds. Nov. 6, 2019

More Synchronization Primitives

- So Far
  - Mutexes
  - Condition variables
- Today
  - Read-write blocks
  - Barriers
- Mutexes and cond. variables are primitive constructs
  - Read-write locks & barriers can be built using mutexes and condition variables

Deadlock

- Deadlock is a state in which no member can make progress
  - All are waiting for other members to take actions
  - A member can be a thread, a process, a computer, etc.
- Example: 2 Mutexes A & B
  - Thread 1, locked A, try to get B
  - Thread 2, locked B, try to get A

Common Solutions

- Fixed order
  - All the threads lock mutexes in the same order
  - m0, m1, and so on
- Try and back off
  - Try to lock. If it fails, release all locks & try again

Readers-Writers Program

- In many applications, a data structure is read frequently btu written infrequently
- Constraints:
  - Multiple readers can read simultaneously
    - A reader can start even if other readers are reading
  - Only one writer can write at a time
  - Read & write cannot happen at the same time
    - A reader has to wait if a writer is writing
    - A writer has to wait for other readers & writers to finish

Solutions?

- Use Pthreads **read-write blocks**
  - Like a mutex w/ **2 diff. lock funcs.** (read/write)
- Use mutex & cond. variables

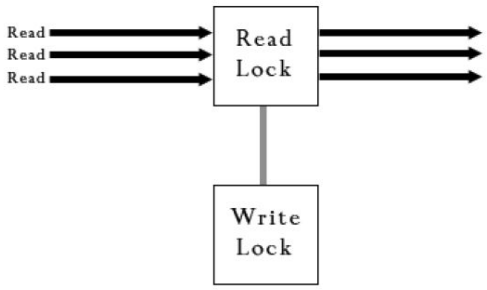
- Keep track of the number of readers and writers
  - Active & waiting
- Choose proper predicate
  - If the predicate is true, proceed to read/write
  - If the predicate not true, wait
- Notify others when a thread is done w/ reading/writing

Pthread Read-Write Lock

```
#include <pthread.h>
pthread_rwlock_t rwlock;    // define a read-write lock

//initialize and destroy the a read-write lock
intpthread_rwlock_init(pthread_rwlock_t*restrict rwlock,
     constpthread_rwlockattr_t*restrict attr);
intpthread_rwlock_destroy(pthread_rwlock_t*rwlock);

// lock functions for readers and writers, and unlock
intpthread_rwlock_rdlock(pthread_rwlock_t*rwlock);
intpthread_rwlock_wrlock(pthread_rwlock_t*rwlock);
intpthread_rwlock_unlock(pthread_rwlock_t*rwlock);
```

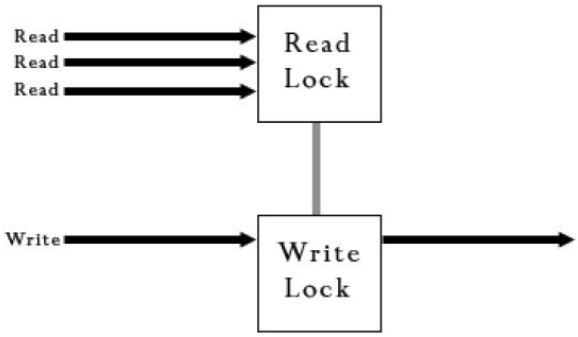
Pthread Read-Write Locks For Reading

- First lock function locks for **reading**
  - A thread gets read lock if and only if no writer has the lock and no writer is blocked on the lock
  - More than one thread can get the read lock



Pthread Read-Write Locks For Writing

- Second lock function locks for **writing**
  - A thread gets write lock if and only if no reader or writer has the lock
    - Wait for read lock, too!
  - Only one writer is allowed at any time



Read-Write Lock Issues

- Writer starvation
  - Too many readers. Writers do not have a chance to start
  - Solution?
- “write locks shall take precedence over read locks”
  - which leads to reader starvation
- Reader starvation
  - Too many writers. Readers do not have a chance
    - Another writer starts when a writer unlocks

Pthread read-write lock

- Pthread rwlock prefers writer
  - From the manual page

```
The pthread_rwlock_rdlock() function shall apply a read lock to the
read-write lock referenced by rwlock. The calling thread acquires the
read lock if a writer does not hold the lock and there are no writers
blocked on the lock.
```

Use mutex and condition to implement rwlock

- Using mutex and condition is more flexible, we can adjust our strategy
  - There are many ways
  - Ex.): alternate reader writer action, every other
- Best for us: One mutex and two condition variables (one for rd, one for wr)
  - Keep track of the numbers needed by the predicate

http://heather.cs.ucdavis.edu/~matloff/158/PLN/RWLock.c

Pseudocode for rwlock lock

```
pthread_mutex_lock(&mutex);
// increment/decrement waiting counter and the whole loop
```

```
// can be placed in an if branch: if (! predicate)
increment waiting counter
// predicate depends on the policy!
while (! predicate)                 // check if this thread can lock
      pthread_cond_wait(&cond);     // using either rd or wr condition
decrement waiting counter
increment active counter
pthread_mutex_unlock(&mutex);
// rwlock is locked. Can start to read/write
// Note that mutex is unlocked!
```

Pseudocode rwlock unlock

```
Perform read/write operations    // Depending on the lock type
// unlock rwlock when the operation is done
pthread_mutex_lock(&mutex);
decrement active counter
// inform waiting threads
// policy decides checking writer or reader first
if there is a writer waiting
      signal write cond
if there is a reader waiting     // only needed for write unlock
      broadcast read cond
pthread_mutex_unlock(&mutex);
```

Example: rwlock_writeunlock()

Does this lock prefer reader or writer?

- **A: reader**
- **B: writer**
- **C: don't know**

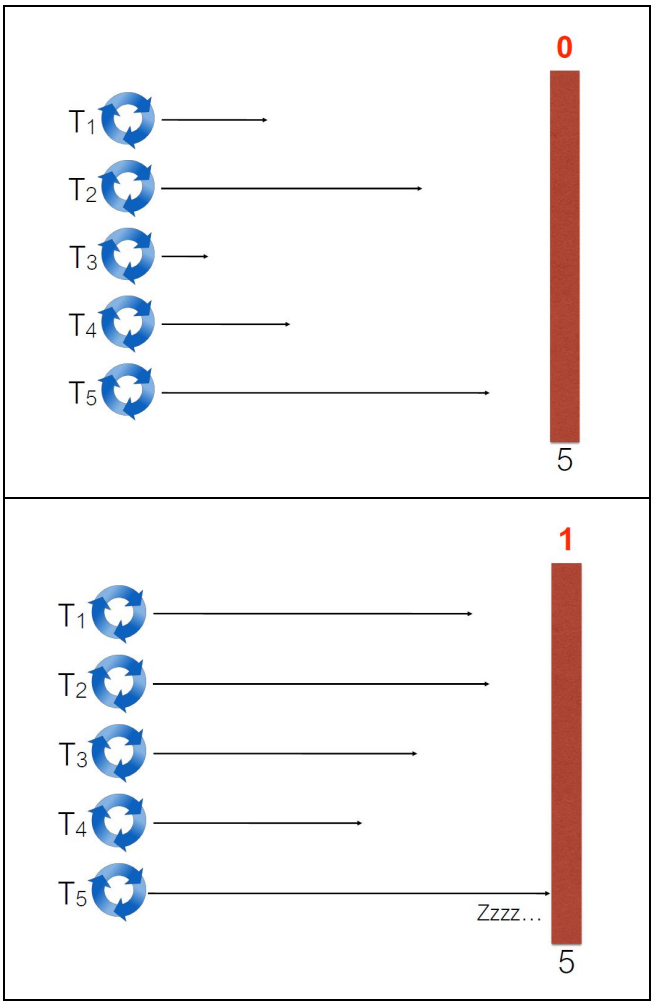
```
status = pthread_mutex_lock(&rwl->mutex);
rwl->w_active = 0
if (rwl->r_wait > 0){
      status = pthread_cond_broadcast (&rwl->read);
      if(status != 0){
            pthread_mutex_unlock (&rwl->mutex);
            return status;
      }
}
else if (rwl->w_wait > 0){
      status = pthread_cond_signal (&rwl->write
```

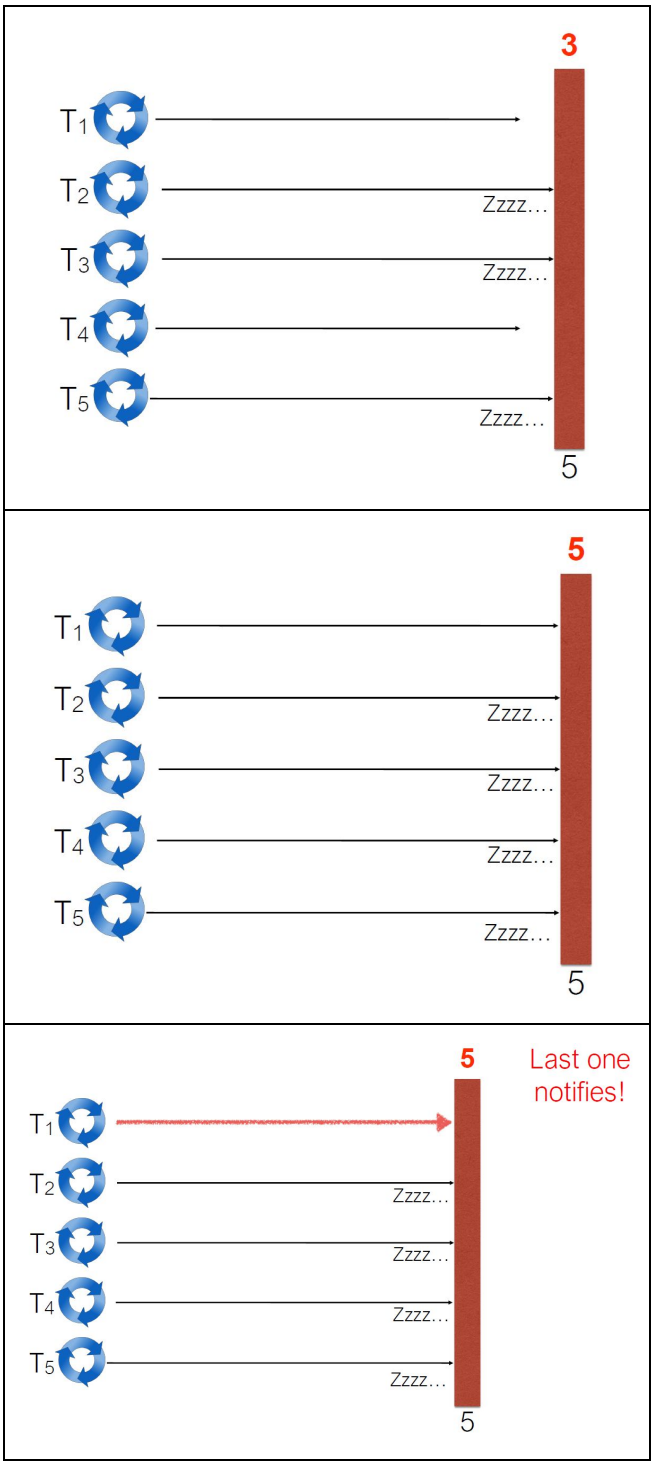
```
      if (status != 0){
            pthread_mutex_unlock (&rwl->mutex);
            return status;
      }
}
pthread_mutex_unlock(&rwl-mutex);
```

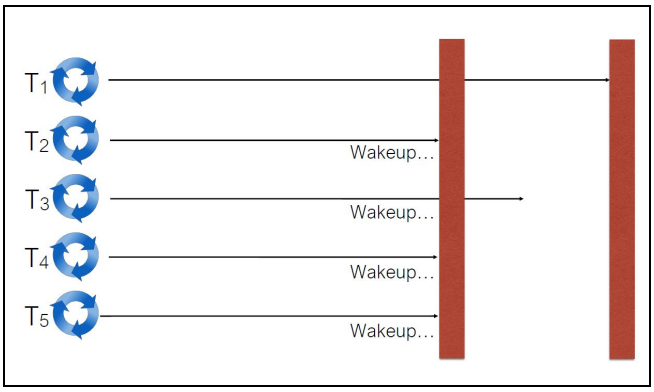
Barriers

- Purpose
  - For applications where work is done in “phases”
  - Must have “worker” threads wait for the entire “group” to be done before proceeding to next phase
  - Number of workers known a priority

Visually




3
T1
T2
Zzzz…
T3
Zzzz…
T4
T5
Zzzz…
5
5
T1
T2
Zzzz…
T3
Zzzz…
T4
Zzzz…
T5
Zzzz…
5
5
Last one notifies!
T1
T2
Zzzz…
T3
Zzzz…
T4
Zzzz…
T5
Zzzz…
5



POSIX Barriers Support

```
#include <pthread.h>
// create barrier. note the count argument
int pthread_barrier_init(pthread_barrier_t* restrict barrier,
      const pthread_barrierattr_t* restrict attr, unsigned count);

// destroy a barrier
int pthread_barrier_destroy(pthread_barrier_t* barrier);

// wait on a barrier. When the function returns,
// one thread gets PTHREAD_BARRIER_SERIAL_THREAD, others get 0
int pthread_barrier_wait(pthread_barrier_t* barrier);
```

Sync mechanisms in this course

- We can use mutex, condition, read/write lock, barriers, and only call blocking functions!
- The following mechanisms are not allowed for synchronization!
    - Busy waiting, like spin lock
    - Sleep functions

*** important for HWs and exams

Lecture 26 - S1 - Intro to Internet Sockets

Mon. Nov. 11, 2019

Sockets

- Allows both inter-machine and inter-process communications
    - Simple APIs for IPC, including communications over network

    - Communication channels created without forking • Greater interoperability
- Plan
    - Introduction to networking and sockets
    - Socket-based communications
        - Client and Server
    - Protocol

Network Sockets

- We will focus on network sockets
    - Networking service is provided by OS
    - Two processes on same/different machines can talk to each other
- TCP/IP provides end-to-end communication layer in the Internet
    - IP: addressing and routing
    - TCP: other services such as error correction and retransmission

Network Protocol Layers

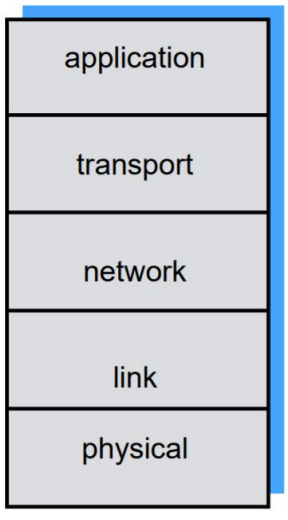
- Network are complex, with many “pieces”:
    - hosts
    - routers
    - links to various media
    - applications
    - protocols
    - hardware, software

Question: is there any hope of organizing structure of network? … or at least our discussion of networks?

Why layering?

Dealing with complex systems!

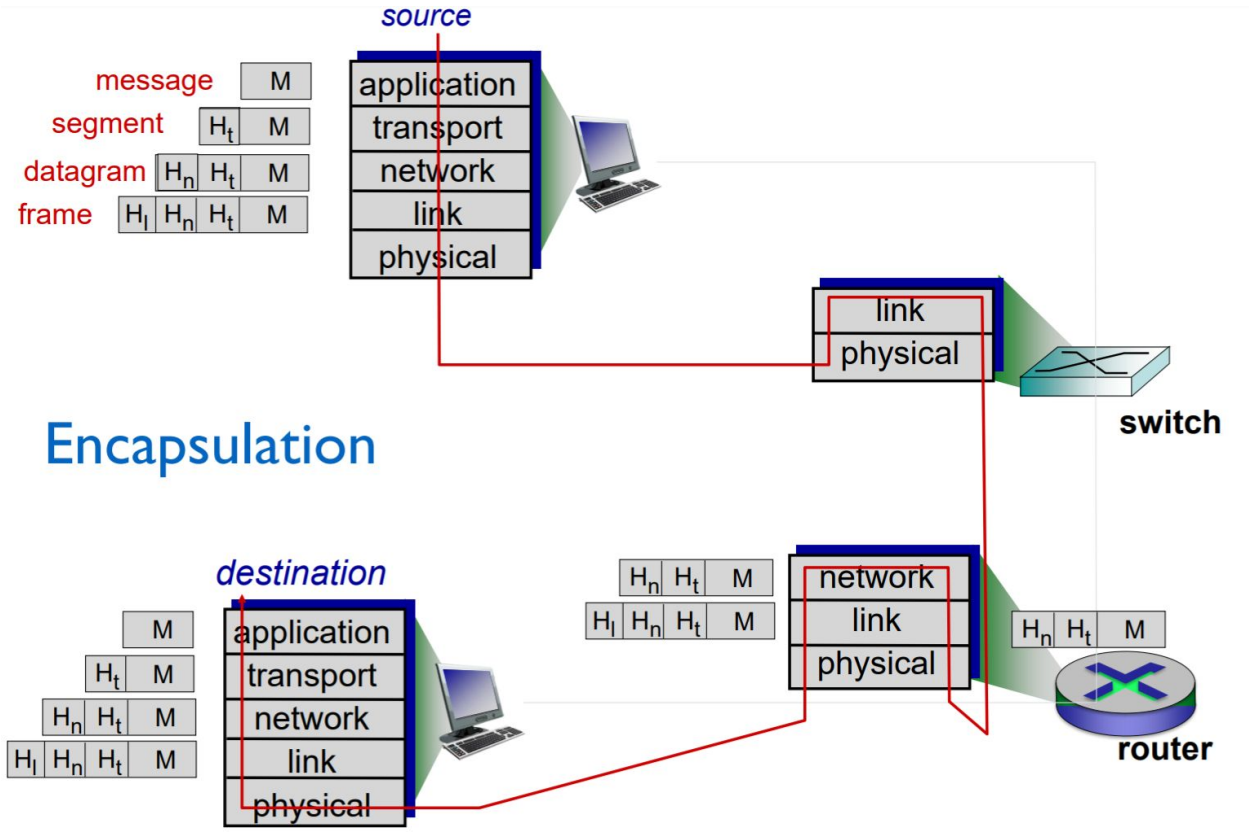
- Explicit structure allows identification, relationship of complex system’s pieces
    - layered reference model for discussion
- Modularization eases maintenance, updating of system
    - change of implementation of layer’s service transparent to rest of system
- layering considered harmful?



Internal protocol stack

- application: supporting network applications
    - FTP, SMTP, HTTP
        - transport: process-process data transfer
    - TCP, UDP

- network: routing of datagrams from source to destination
    - IP, routing protocols
- link: data transfer between neighboring network elements
    - Ethernet, 802.111 (WiFi), PPP
- physical: bits “on the wire”



IP (Internal Protocol)

- A datagram service at the network layer
    - Data are divided into packets
    - Every packet has a destination “label”: the IP address!
    - Packets are handled independently
    - And delivered with best-effort
        - However, you can see loss, out of order, duplication,..

IP Address

- IPv4: 4 bytes
    - Each device normally gets one (but can have more)
        - Each interface(boundary between host and physical link) has one IP address
- IPv6: 16 bytes

- 128 bit addresses
  - Make it feasible to be very wasteful with address allocations
- Lots of other new features
  - Built-in auto configuration, security options, …
- Not compatible with IPv4

IPv4 examples

A Google server:
67.218.93.15

A gateway at UConn:
137.99.10.1

Private addresses:
10.0.0.0 - 10.255.255.255
172.16.0.0 - 172.31.255.255
192.168.0.0 - 192.168.255.255

Domain name for dummies

- A domain name is an identification string
  - Formed by rules and procedures in DNS (Domain name system)
  - Hierarchical
- Fully qualified domain names (FQDNs) is a complete domain name for a specific computer on the Internet
- Domain name servers (also a DNS) translate domain names to IP address
  - Phonebook for the Internet

Example:
www.uconn.edu is a FQDN.
www.google.com is a FQDN. Its IP address is 172.217.7.228

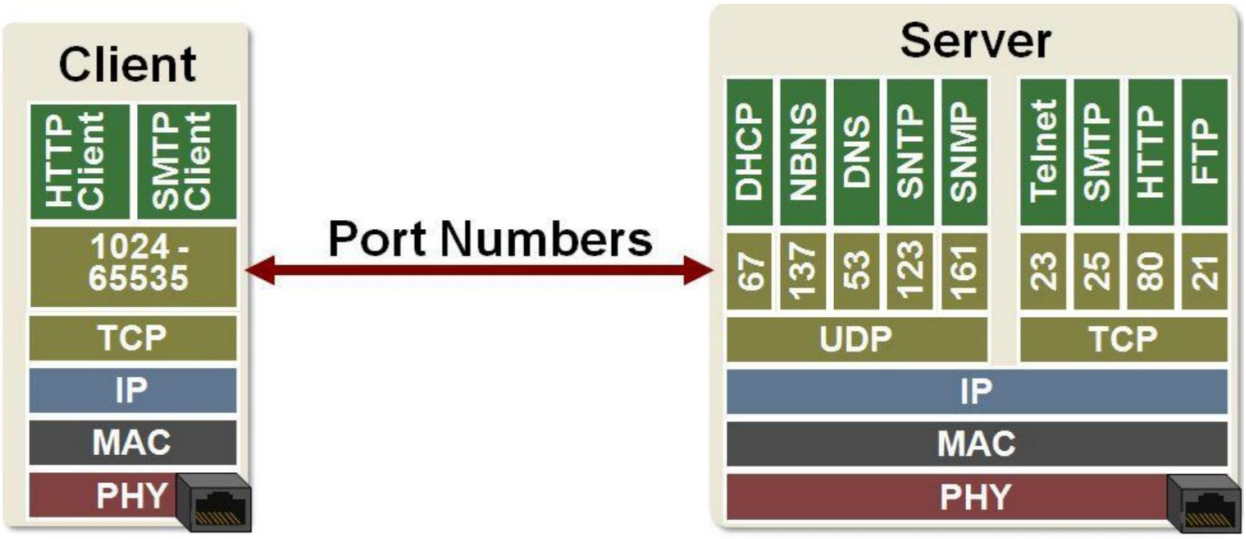
The Internet Transport Layer

Two protocols above IP

- TCP (Transmission Control Protocol) TCP/IP
  - **Connection-oriented** like POTS (plain old telephone service)
  - Flow-control and bi-directional
    - Reliable
- UDP (User Datagram Protocol) UDP/IP
  - **Connection-less** like (unregistered) snail mail
  - No acknowledgments or retransmissions
  - Packets may be delivered out of order and have duplicates
    - Retransmission can be handled in applications

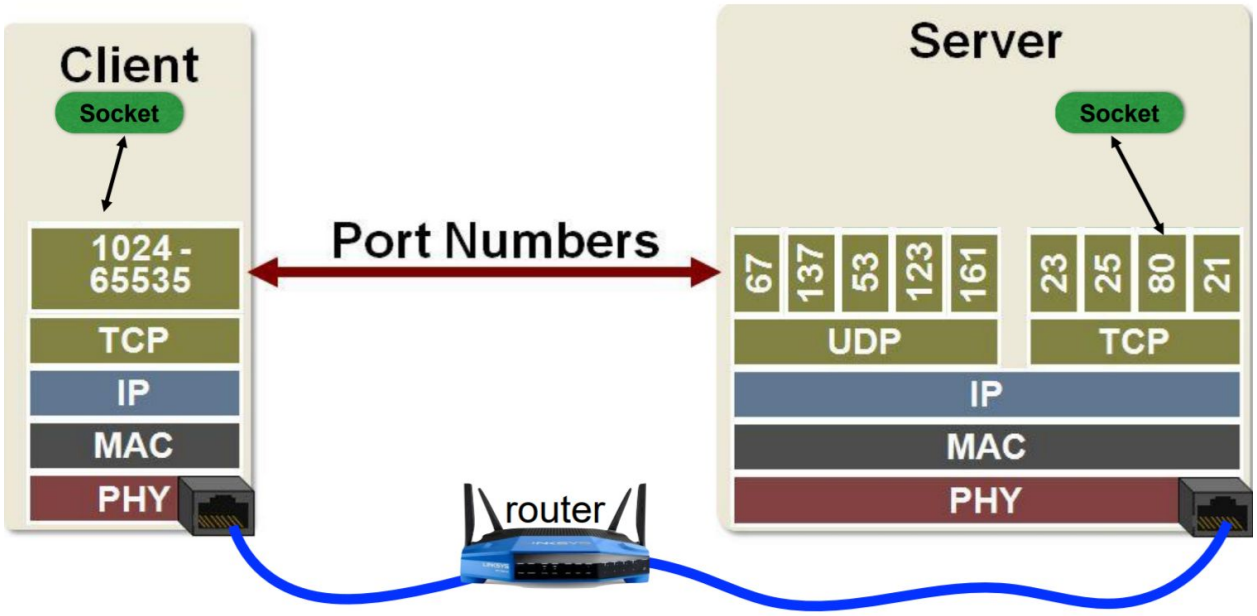
Ports

- A server can provide many services
- Ports Identify services and applications for both TCP and UDP



Sockets

- The programmatic abstraction to access ports



Commonly used TCP port numbers

Port number	Service/ Protocol
21	FTP
22	SSH
25	SMTP
80	HTTP
137-139	NetBios
443	HTTPS

Sockets API

- API to create, close, read, and write
  - Sockets are like files!
  - Can use the normal I/O functions
- Two types of sockets
  - Stream sockets for TCP
  - Datagram sockets for UDP
    - Single payload oriented
- Need to associate a socket with a port (coming up next time)
  - An Internet address (IPV4 or IPV6)
  - A port number

Create a socket

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
// Use close() to close a socket
```

Returns a file descriptor for the given domain, type, and protocol.
Returns -1 on error.

domain: AF_INET, AF_INET6, etc.
type: SOCK_STREAM, SOCK_DGRAM, etc.
protocol: 0.

Send and recv for TCP

```
#include <sys/types.h>
#include <sys/socket.h>
ssize_t send(int sfd,const void *buf,size_t len,int flags);
```

```
ssize_t recv(int sfd, void *buf, size_t len, int flags);
```

- Return the number of bytes that have been sent/received
  - It can be less than the number of bytes requested!
  - sent does not mean delivered!
- Both returns -1 on errors
  - Similar to read() and write()
    - You can use read and write if you do not need flags
  - Working with TCP (not UDP)

Example: send

After a connection is established!

```
// To send n bytes stored in buf
sent = 0;
while(sent < n){
      int r = send(sid, buf +sent, n - sent, 0);
      if (r < 0) report_error();
      sent += r;
}
```

Example: recv

After a connection is established!

```
received = 0;
while(received < n):|{
      int r = recv(sid, buf + received, n - received, 0);
      if (r < 0) report_error();
      received += r;
}
```

Study this slide yourself

Useful commands. Read the manual for details

```
#scan ports
nmap localhost

#list FDs for the Internet Need sudo for system files.
lsof - i

#show information about networking subsystem
netstat -l
```

```
netstat -l -4
```

Lecture 27 - S2: Sockets Programming: Client & Server

Weds. Nov. 13, 2019

Review

- TCP/IP
    - Name/IP and port number
- Socket
    - We get a file descriptor and we can send and recv

A Client-Server World

Establish a connection between a server and client

- Server
    - Listens on a port for inbound “let’s talk” requests
- Client
    - Initiates the comm. w/ the server
    - Must know
        - Server name (which is used to find the IP address)
        - Protocol to use (TCP/UDP)
        - Port number

Server side (TCP)

- After creating a socket
    - bind() Server binds (reservers) a port for chatting
    - listen() Server listens on port for “Let’s talk requests”
    - accept() When a request comes in …
        - Server allocates a separate socket to accept private comm. w/ the client (same host, diff. port)
            - The original socket remains available for future “Let’s talk”
            - The new socket is bound to an ephemeral port
        - Comm. goes on on separate sockets with
            - send() / recv()
        - close() Conversation ends via a close

Analogy Between Sockets and Telephone

	Sockets	**Telephone**

socket()	Create a socket	Buy a telephone
bind()	Bind to a port	Plug in / Insert a SIM
listen()	Get ready to respond	Get ready for ringtone
accept()	Accept requests	Pick up the phone
send() / recv()	Communication	Listen and talk
close(0	Close the communication	Hang up

Client side (TCP)

- After creating a socket
    - connect() Client initiates a session
        - Need server name, type, and port
    - Then, comm. goes on with send() / recv()
    - close() Client closes a session

Note that no bind() is needed on the client side OS allocates a port

Getting back to server…

```
#include <sys/types.h>
#include <sys/socket.h>

int bind(int sockfd,
     const struct sockaddr *addr,
     socklen_t addrlen);
```

- sockfd is the socket file descriptor. It does not have an address!
- What is struct sockaddr?
    - It holds address information for many types of socket

Socket address types

```
// generic socket address
struct sockaddr{
     unsigned short sa_family; // address family, AF_xxx
     char sa_data[14];         // 14 bytes of protocol address
};
// we are using Internet socket address
struct sockaddr_insa; // socket address IPv4
struct sockaddr_in6 sa6; // socket address IPv6
// struct sockaddr_in* can be cast to struct sockaddr*
```

```
// Do I really need to learn how to set the fields in saand sa6?
```

addrinfo structure

```
// A recent invention for preparing the socket address
struct addrinfo {
      int   ai_flags;             // AI_PASSIVE, AI_CANONNAME, etc.
      int   ai_family;            // AF_INET, AF_INET6, AF_UNSPEC
      int   ai_socktype           // SOCK_STREAM, SOCK_DGRAM
      int   ai_protocol           // use 0 for "any"
      size_t     ai_addrlen       // size of ai_addr in bytes
      struct     sockaddr * ai_addr;   // struct sockaddr_in or _in6
      char *ai_canonname;  // full canonical hostname
      struct     addrinfo *ai_next;    // linked list, next node
}
```

getaddrinfo()

```
int getaddrinfo(const char *node // "www.example.com" or IP
            const char *service,  // "http" or port #
            const struct addrinfo *hints,
            struct addrinfo **res
```

- this function finds addrinfo for you!
    - Provide you a linked-list, res. A list of results!
- ***node*** is the host name to connect to, or an IP address
- ***service*** can be a port number, like "80", or the name of a particular service like "HTTP"
- ***hints*** points to a struct addrinfo that you've already filled out with relevant information.

Usage of getaddrinfo() for server

```
struct addrinfo hints, *servinfo;           // res will point to the
results

memset(&hints, 0, sizeof(hints));           // make sure the struct
is empty
hints.ai_family = AF_UNSPEC;           // don't care IPv4 or IPv6
hints.ai_socktype - SOCK_STREAM        // TCP stream sockets
hints_ai_flags = AI_PASSIVE;           // fill in my IP for me
if (getaddrinfo(NULL, "3490", &hints, &servinfo) != 0)
      die("Error");
```

```
// servinfo now points to a linked list of 1 or more struct addrinfos
// continue on the next page
```

Now, let us create socket and then ibnd!

```
int sockfd, yes = 1;
struct addrinfo *p    ;
for (p = servinfo; p != NULL; p = p->ai_next){
//socket(); setsockopt(); bind();
// should check return value of errosr!
sockfd = socket(p->ai_family, p->ai_socktype, p->ai_protocol);
setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int));
// bind it to the port we passed in to getaddrinfo();
bind(sockfd, p->ai_addr, p->ai_addrlen);
break;
}
freeaddrinfo(servinfo);          // free the linked-list
```

***** should understand it at least once, afterwards, just copy, paste, and use**

After bind()

```
// listen on that socket for "Let's talk" message.
status = listen(sockfd,10);
checkError(status);
// client's address information, if you are interested in
struct sockaddr_storage client_addr;
while(1) {
   socklen_t clientSize = sizeof(client_addr);
   int chatSocket = accept(sockfd,
                           (struct sockaddr*)&client_addr,
                           &clientSize);
   checkError(chatSocket);
   printf("We accepted a socket: %d\n",chatSocket);
   close(chatSocket);
}
```

Usage of getaddrinfo() for client

```
struct addrinfohints, *servinfo; // will point to the results

memset(&hints, 0, sizeofhints); // make sure the struct is empty
hints.ai_family= AF_UNSPEC; // don't care IPv4 or IPv6
hints.ai_socktype= SOCK_STREAM; // TCP stream sockets
// get ready to connect
if (getaddrinfo("www.uconn.edu", "3490", &hints, &servinfo) != 0)
      die("Error");
// servinfonow points to a linked list of 1 or more struct addrinfos
// continue on the next page
```

But, how can i do bind()?

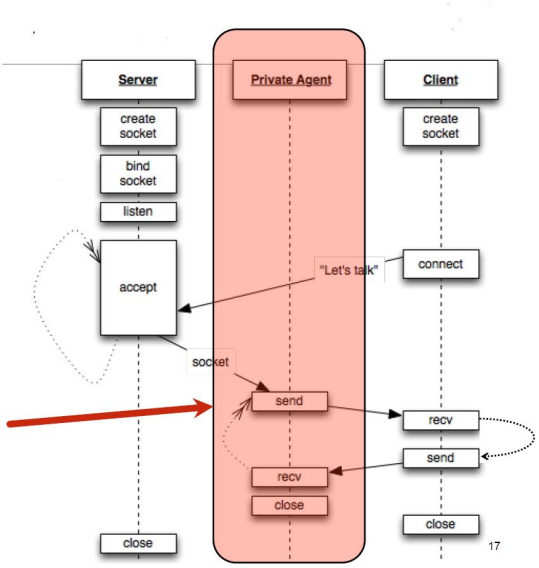
```
int sockfd, yes = 1;
struct ddrinfo *p;
for (p = servinfo; p != NULL; p = p->ai_next);
// socket();
// should check return value for errors!
sockfd = socket(p->ai_family, p->ai_socktype, p->ai_protocol);
// connect
connect(sockfd, p->ai_addr, p->ai_addrlen);
break;
}
…
freeaddrinfo(servinfo);          // free the linked-list
```

Blocking and Talking to Many Clients

- On the server side
  - Accept is blocking (until a “let’s talk” request comes in)
  - If serving the client directly…
    - In bound “Let’s talk” are queue up (up to maximum)
  - If the server wishes to get back to accepting: fork a process! or create a

    Could be a cloned process (fork!) … or a thread

    thread!



        - All real comm. happens on the accepted socket
- On client side
    - Connect is blocking (until the server responds or refuses)

Server Dealing with Many Clients

- The server cannot talk to many clients at the same time
    - Let others to deal with clients
    - It can wait for new clients

Lecture 28 - S3: Sockets Programming: Protocol

Mon. Nov. 18, 2019

Network Protocols

- Rules for communications between two parties
    - Need protocols at every layer
    - TCP and IP are protocols!
- After establishing a TCP connection, two processes talk at application layer
    - Who should talk first?
    - How much can each talk?
    - How are the messages encoded into bytes?

Byte stream in sockets

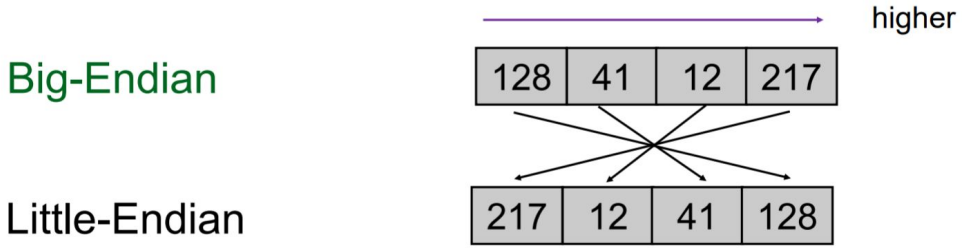
- Applications see byte streams into and out of sockets
- It is up to applications to interpret bytes
- Two common options (both parties have to agree !)
    - Everything is in plain text. Need conversion and parsing
        - Example: XML | JSON
    - Raw data. Issues: endianness and padding

Example: int v = 128. Send v.
Option 1: three ASCII characters with a delimiter: '1', '2', '8', '\n'
Option 2: four bytes: 0x00, 0x00, 0x00, 0x80 (or 0x80, 0x00, 0x00, 0x00?)
Is the size of v really 4?

Endianness

- Two ways to use bytes if a data item has more than 1 byte
    - Some hosts use big-endian while some use little-endian
- Both parties need an agreement to interpret the bytes

Integer: 2150173913 or 0x80 29 0C D9
Bytes: 128(0x80), 41(0x29), 12(0x0C), 217(0xD9)



Change byte ordering for 16-bit or 32-bit integers

- Two orderings may not be the same
  - Host Byte Ordering — natural for platform
  - Network Byte Ordering — always big-endian
- Convert to network byte ordering before send
- Convert to host byte ordering after receive
  - htonl and ntohl for 32-bit. htons and ntohs for 16-bit

```
#include <arpa/inet.h>

uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

Sending a string

- We are fine in this course because we assume ASCII strings
  - Each character has only one byte
- In real word, modern applications assume Unicode internally
  - Especially important for web servers
    - Your customers/users speak many kinds of languages!

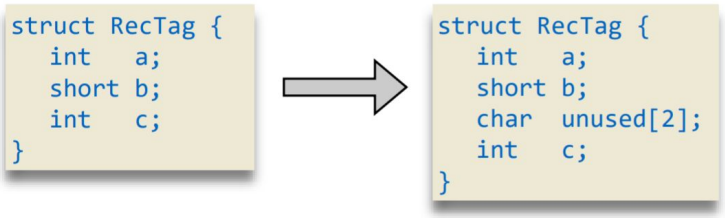
Example:
Python 3 uses Unicode strings by default

```
"♡♬".encode('utf-8') # encode a string to byte array using UTF-8
data.decode('utf-8') # decode a byte array using UTF-8 to a string
```

Issues with structure: alignment and padding

- Compilation can introduce padding inside structures
  - Affect the structure size and the offset of the fields
  - Important when the two endpoints have different architectures

    - e.g. 32 bit vs. 64 bit even if both are Intel
- How to deal with the issue?
    - Use explicit padding! (or reorder fields)
    - Do not use any language/machine dependent data/structure



Remember to change byte order for a, b, and c !

How about arrays?

- Convert each element to bytes individually

- How about arrays of structures that have array members?
    - You can see it is becoming very tedious

JSON: JavaScript Object Notation is an open-standard file format that uses human-readable text to transmit data objects consisting of attribute– value pairs and array data types (from Wikipedia)

Example: SMTP(Simple Mail Transfer Protocol)

S: 220 smtp.example.com ESMTP Postfix
C: HELO relay.example.com
S: 250 smtp.example.com, I am glad to meet you
C: MAIL FROM:<bob@example.com>
S: 250 Ok
C: RCPT TO:<alice@example.com>
S: 250 Ok
C: DATA
S: 354 End data with <CR><LF>.<CR><LF>
C: From: "Bob Example" <bob@example.com>
C: To: Alice Example <alice@example.com>
C: Date: Tue, 15 Jan 2008 16:02:43 -0500

Each line ends with , "\r\n"
Server may respond with many lines, for example, to EHLO command
Based on an example on Wikipedia

Example: SMTP - 2

C: Subject: Test message

C:

C: Hello, Alice,

C: This is a test message.

C: Bob

C: .

S: 250 Ok: queued as 12345

C: QUIT

S: 221 Bye

{The server closes the connection}

Example: TFTP-style applications

Trivial File Transfer Protocol (TFTP)

- Server waits for commands and performs operations accordingly
- Client sends a command to the server and waits for the response

- Three commands are supported
  - LS List files in the directory on the server side
  - GET fn Download a file named “fn” from the server
  - EXIT End the session

Commands

Client sends a command
Server responds

Commands	Response
LS	A list of file names
Get fn	A file
EXIT	nothing

How to encode the command, messages, and files?

Command Options: Text

- A command is an ASCII string ending with a delimiter, e.g., '\n'
  - Both parties have to agree on the delimiter

Example:

```
GET(fn)
      send_str("get hello.txt\n"); // no need to send NUL
```

Pro	Cons
Easy for humans to read	Require parsing
Flexible and compatible	Variable length. Check of the delimiter
	Not compact for long commands

Command Options: Binary, fixed size

- A command is represented by a structure
    - All commands have the same size (good or bad?)
- Example:

```
#define LS      0
#define GET     1
#define PUT     2
#define EXIT    3

struct Command {
   int code;
   char fileName[252];
} cmd;

cmd.code = htonl(GET);
send_all(sid, &cmd, sizeof(cmd));
```

Pro	Cons
Nothing to parse	Hard to read and maintain
Fixed size	Fixed size
	Endianness
	Padding

Commands Options: Dynamic Size

- LS and EXIT need only 4 bytes (or we can use only one byte)
- GET needs to send additional (filename length + 4) bytes

```
#define LS      0
#define GET     1
#define PUT     2
#define EXIT    3

int code = htonl(GET);
int fnlen = strlen(fname);
int nlen = htonl(fnlen);

send_all(fd,&code,sizeof(int));
send_all(fd,&nlen,sizeof(int));
send_all(fd,fname,fnlen);
```

Pro	Cons
Nothing to parse	Hard to read and maintain
Dynamic sizing	Endianness
	More brittle

Sending the response

```
typedef struct ResponseHeader_tag {
            int code;
            int length;
} ResponseHeader;
```

- code: type of the payload: text message or file
- length: the number of bytes in the text message or file

A payload is followed by length bytes of message or file contents
Remember the byte ordering of code and length!

Recapping

- Check return value
- Validate input
- Understand strings in C
    - Always think about if you have enough space for strings
        - strcpy(), strcat(), etc.

Don'ts	**Do's**
scanf("%s", buf);	scanf("%10s", &buf);
gets(buf);	fgets(buf, BUF_SIZE, stdin);
sprint(buf, "5s", s);	snprintf(buf, BUF_SIZE, "%s", s);

Lecture 29 - T6: Threads - Some Loose Ends

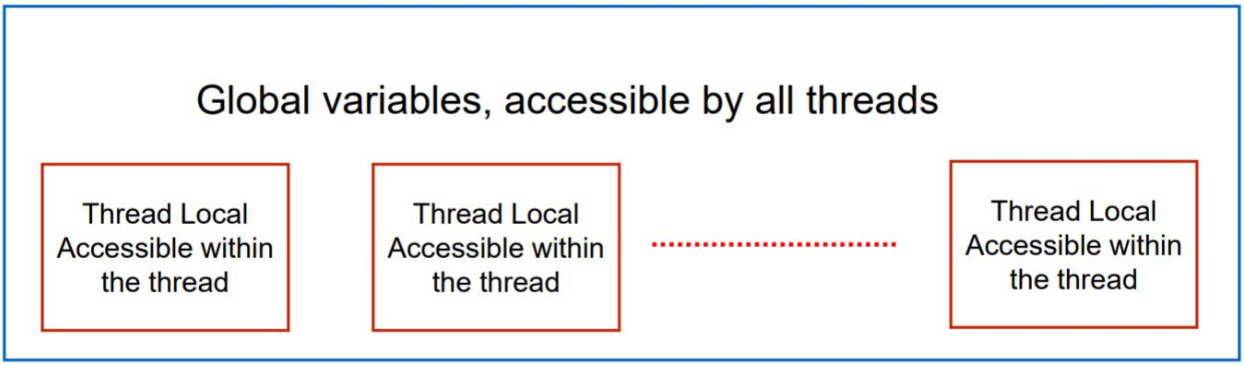
Weds. Nov. 20, 2019

Overview

- Thread Local Storage
- Thread Cancellation
    - Purpose
    - APIs
- Priority Inversion

Thread Local Storage (TLS)

- Purpose
    - Have “Globals” that are “Local” to each thread!
    - Can be accessed by all the functions in a thread



TLS in C

- Quite easy to do with modern C compilers
- In C
    - MSVC, gcc and clang have a new storage keyword: **`__thread`**
    - C11 has **`_Thread_local`**

```
__thread int x = 0;
```

- “x” is a global to each thread.
- Each thread regardless of when it is created gets its own copy of “x” !!!

Cancellation use case

- Imagine that you have several threads to find a solution to a problem
    - You do not know which could find the answer the fastest

    - Example: search a value in a large data set
- What do you do when one thread finishes?
    - It would be better to ask other threads to stop

Our need

- Ability to ask from thread A to terminate thread B
- Requirements
    - Thread A must know B's identity
    - But B should terminate "safely"

What do we mean by "Safely"

- The thread receiving the request may be…
    - In the middle of doing some IO (blocking in the kernel)
    - In the middle of updating some data-structure
    - In the middle of a critical section
        - Therefore holding locks!
    - Asleep in a POSIX condition (so on wake up he would hold locks!)
- Stopping "brutally" in any such scenario is a recipe for disaster

Cancellation is "civilized termination"

- One thread does not "kill" another
- One thread asks another thread to…

"please stop at your earliest convenience"

- The recipient (cancellation target) can
    - Ignore it
    - Honor it immediately
    - Defer to a "safe point"
        - After a critical section
        - IO done
        - Locks released, etc.…

Two sides to the story

- The thread asking another to cancel itself
    - Quite easy, just call pthread_cancel()
- The thread receiving the request to self-terminate
    - A bit more involved to handle safety

Sender: call pthread_cancel

```
int pthread_cancel(pthread_t thread);
```

- It requires the thread identifier of the “victim”
- It returns as soon as the request is logged

- Recall that it is merely a request
- Victim may keep on executing for a while

Recipient’s perspective

- Recipient can be in different “moods”
  - Mood is based on State and Type

	Type **Deferred**	Type **Asynchronous**
State **disabled**	cancellations remain pending	cancellations remain pending
State **enabled**	cancellations occur at next cancellation point	cancellations may be processes at any time

State and Type

- A thread can
  - Change its cancellation **state** at any time

```
int pthread_setcancelstate(int state, int *oldstate);
```

  - • Change its handling **type** at any time

```
int pthread_setcanceltype(int type, int *oldtype);
```

Previous values are returned through the second argument

Request Deferred until a Cancellation Point

- A cancellation point is a function in the code where a cancellation request can be safely honored.
- Most of the cancellation points are
  - System calls
  - C library functions

- Thread related
- IO related

Remember cancellation request is honored only if the thread is in the right "mood"

List of Cancellation Point

- Predefined cancellation point
  man 7 pthreads,
  or http://man7.org/linux/man-pages/man7/pthreads.7.html

accept, aio_suspend, close, connect, creat, fcntl, fsync, lockf, msgrcv, msgsnd, msync, nanosleep, open, pause, poll, pread, pselect, pthread_cond_timedwait, pthread_cond_wait, pthread_join, pthread_testcancel, pwrite, read, readv, recv, recvfrom, recvmsg,select, sem_wait, send, sendmsg, sendto, sigpause, sigsuspend, sigwait, sleep, system, tcdrain, usleep, wait, waitpid, write, writev

- Most are system calls

pthread_testcancel

- What if your code does not call any of the functions?

Add a cancellation point:

```
pthread_testcancel();
```

- It does nothing if there is not cancellation request or cancelability is not enabled
- It will handle the cancellation request otherwise

Clean up

- The cancelled thread must use the cancellation point to "cleanup"
  - Release locks
  - Close resources
  - Free memory
  - …

Leave data in a coherent state

- How to do this?
  - With cleanup handlers

Cleanup Handlers

- Functions that must be executed at cancellation time
  - A handler is one function
  - You get access to the function via pointers
  - Invoked handler receives a void* to an arbitrary data structure
- Multiple handlers?
  - You have a stack of "scheduled" cleanup handlers
  - You push a handler /data pair on the stack if a cleanup is needed

- You pop a handler / data pair if the cleanup is no longer needed

Demo!

Asynchronous Cancellation?

STAY AWAY FROM IT

- No need for victim to check with pthread_testcancel
- Victim killed without any chances to recover

- Rarely needed
- Difficult to use correctly
  - Safe to use when computation bound with no locks / resources

Doing something only ONCE

- Make sure that an operation is done only once, even if called from several threads

```
#include <pthread.h>

int pthread_once(pthread_once_t *once_control,
                 void (*init_routine)(void));
```

- Use case examples
  - Initialize a mutex once
  - Set the seed of a random number generator once

Pthread scheduling

- You can change
  - Thread scheduling policies: FIFO, round-robin, …
  - Thread scheduling priorities: min. 32 priority levels

See pthread_setschedparam()

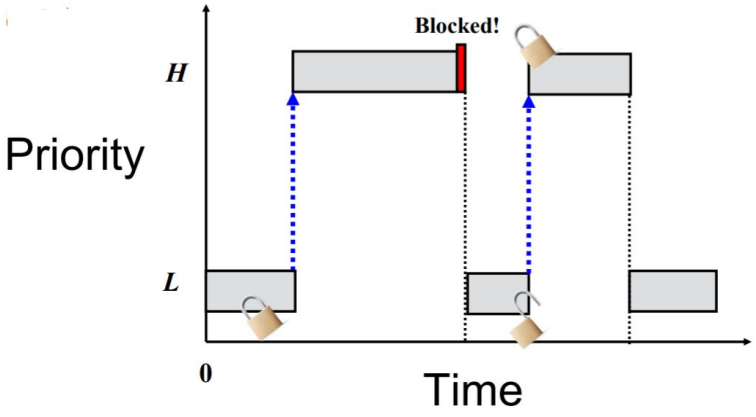
Priority inversion

- High priority threads cannot run while low priority threads can
- The result of a conflict between
  - Resource locking
  - Priority scheduling

Priority inversion example

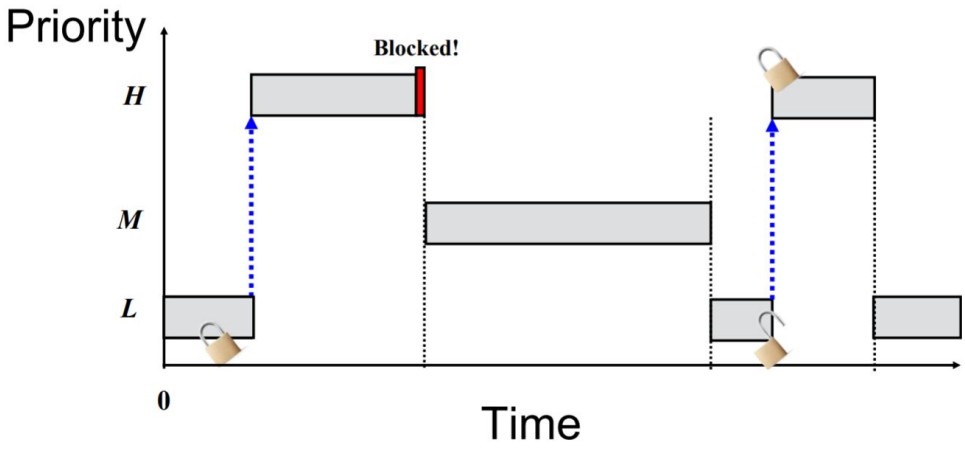
- Mutex is locked by a low priority thread L
- A high-priority thread H trying to grab a lock…

- H goes to sleep because the mutex is locked
- So the low priority thread gets to execute before the high-priority thread!



Can be worse!

- Thread M has a priority level between H and L
- M does not need the mutex. It runs before L, and before H

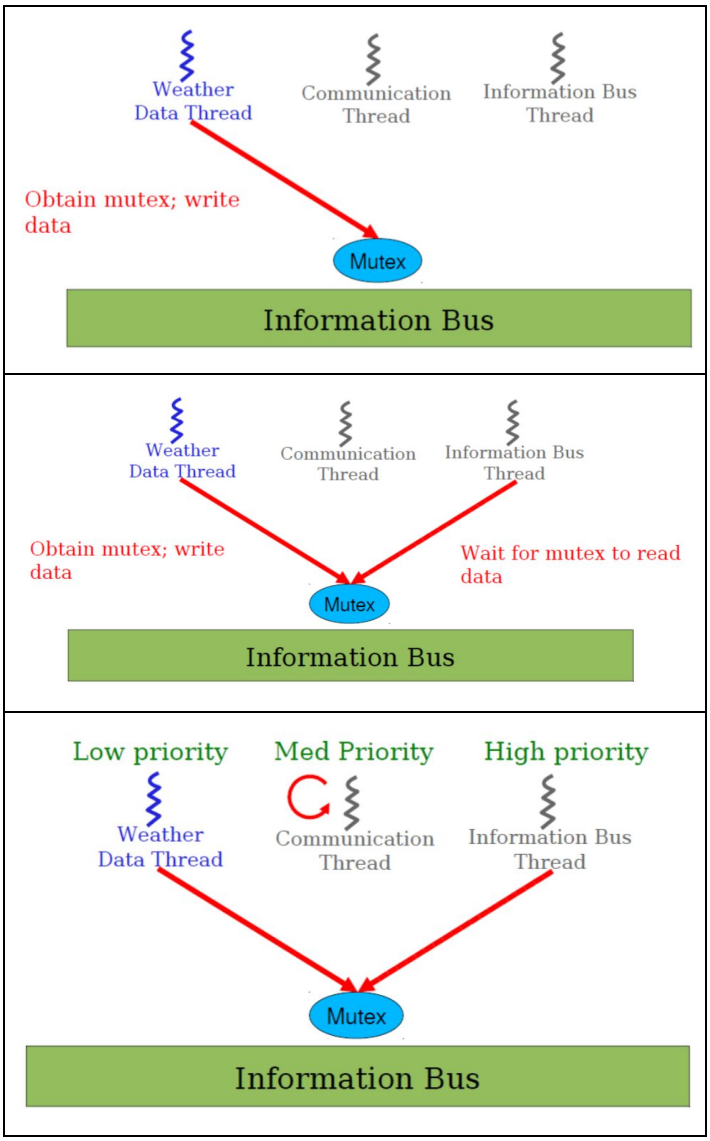


Solution? Priority boosting

- Trouble in the example
  - Low priority thread holding a lock preventing high priority thread to run
- Solution
  - Temporarily boost the priority of the thread holding the lock to match the priority of the blocked high-priority thread
  - It will release the mutex sooner and let the real high-priority thread run
- In practice
  - Handled by a priority inheritance attribute associated to the mutex

Priority inversion on Mars

- Priority inversion caused Mars Pathfinder rover dozens of unexpected system resets



Lecture 30 - Practice Exam 3

Mon. Dec. 2, 2019

Rock, paper, and scissors

This is a game everyone has played. There are two players in the game. Each player makes a choice of a hand shape, rock, paper, or scissors and compares their choices.

In this problem, we will use two threads to simulate two players playing the game for `n` rounds. A third thread serves as a referee, a trusted party that compares players' choices. In each round, plays make a decision and inform the referee their choice. The referee compares the choices and annouces the result to both players.

`thread_player()` and `thread_referee()` are the main function of two types of thread. Compelete the functions so two players can play the game.

Structure `shared_int_t` is used for players to send their choice to referee. The referee shares `choice1` with player 1 and `choice2` with player 2. `mutex` and `cond` are used for synchronization.

Sturcture `result_t` is for referee to announce the outcome to players. The referee shares `result` with both players. `barrier` is used for synchronization.

The program takes several arguments. For example, `-n` option specifies the number of rounds. Read the code for details.

Here is the sample output of the program.

```
$ ./rock-paper -q
Player 1 won   2 times, lost   5 times, and tied   3 times.
Player 2 won   5 times, lost   2 times, and tied   3 times.

$ ./rock-paper -n100 -q
Player 1 won  29 times, lost  38 times, and tied  33 times.
Player 2 won  38 times, lost  29 times, and tied  33 times.

$ ./rock-paper -n1000 -q
Player 1 won 318 times, lost 333 times, and tied 349 times.
Player 2 won 333 times, lost 318 times, and tied 349 times.
```

Additonal challenges

Try different synchronization mechanisms.

Use mutex and cond in `result_t`

Sturcture `result_t` does not have to use barrier for synchronization. It can rely on mutex and cond.

Use barrier in `shared_int_t`

In this problem, we can use barrier in `shared_int_t` for synchronization.

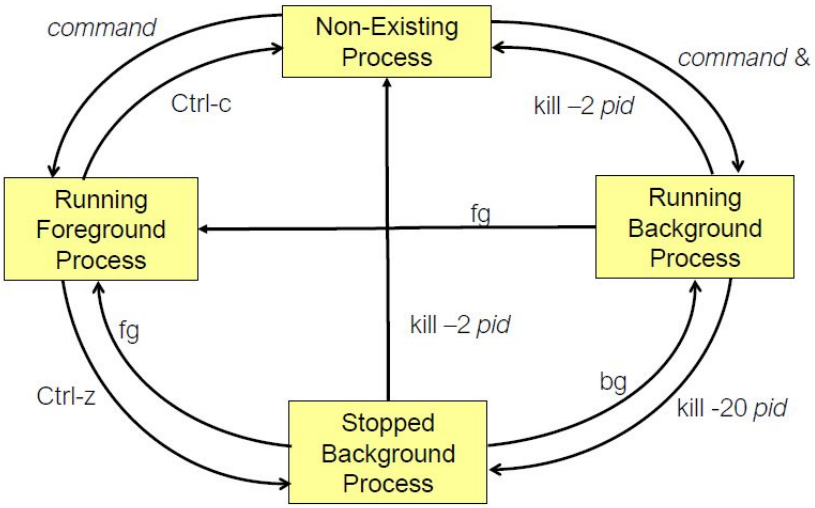
Lecture 31 - X1: Misc Topics: Signals

Weds. Dec. 4, 2019

Process Control & Signals

- Unix process control
- Signals (ABC 12.4)
  - From keyboard
  - Sent via function calls
  - Signal handlers
- Overview of signals
  - man -7 signals

Unix process states



Signals

- A signal is an ***asynchronous*** event that is delivered to a process
  - Asynchronous means that the event can occur at any time
  - May be unrelated to the execution of the process, e.g., user types ctrl+C
- What happens when an event is triggered?
  - Event gains attention of the OS

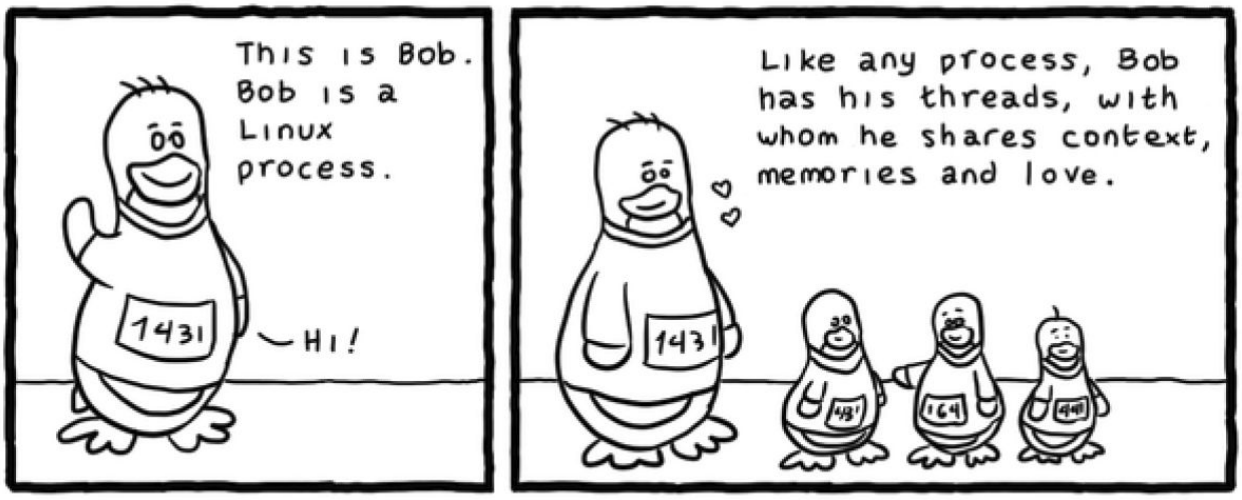
- OS stops the application process immediately
- **Signal handler** executes to completion
- Application process resumes where it left off (unless signal is to terminate it)

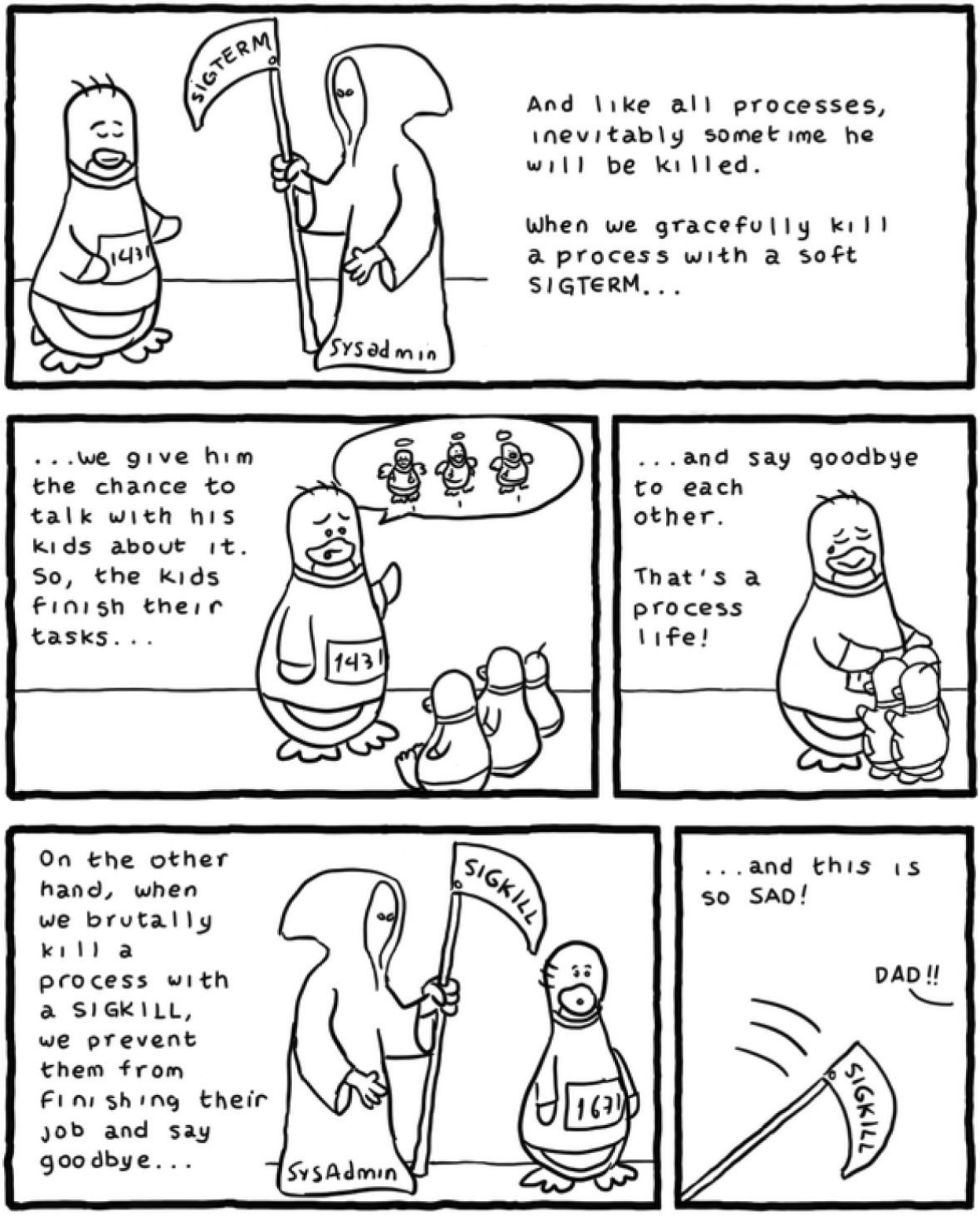
- Some example signals. Run “kill -l” for full list

Name	Description	Default Action
SIGINT	Interrupt character (Ctrl-C)	terminate process
SIGTERM	Terminate process	terminate process
SIGKILL	kill -9	terminate process
SIGFPE	Floating-point exception	create core image
SIGSEGV	Invalid memory reference	create core image
SIGPIPE	Write to a(no reader) pipe	terminate process
SIGALRM	alarm() clock ‘rings”	terminate process
SIGSTOP	Suspend the process	suspend process

- SIGTERM vs. SIGKILL ???
  - Graceful vs. forced termination

Don’t SIGKILL (unless you have to)




SIGTERM
1431
SYSadmin
And like all processes, inevitably sometime he will be killed.
When we gracefully kill a process with a soft SIGTERM...
...we give him the chance to talk with his kids about it. So, the kids finish their tasks...
1431
...and say goodbye to each other.
That's a process life!
On the other hand, when we brutally kill a process with a SIGKILL, we prevent them from finishing their job and say goodbye...
SIGKILL
1671
SysAdmin
...and this is so SAD!
DAD!!
SIGKILL

Sending Signals via Commands

- "kill command
  - **`kill -signal pid`**
    - Send a signal of type signal to the process with id pid
    - Can specify either signal type name (-SIGINT) or number (-2)
  - No signal type name or number specified => sends 15/SIGTERM signal
    - Default 15/SIGTERM handler exits process
  - Editorial comment: Better command name would be sendsig
- Examples
  kill -2 1234 or kill -SIGINT 1234
  - Same as pressing Ctrl-C if process

Sending Signals via Function Calls

```
int kill(pid_t pid, int sig);
int raise(int sig);   // send to itself
```

- Return 0 on success
- Send a signal sig to process pid of itself
- raise(sig) is equivalent to kill(getpid(), sig) for single-thread apps
- Editorial comment: Better functio nname would be sendsig()

Example

```
// Process sends itself a SIGINT signal (commits suicide?)
kill(getpid(), SIGINT);
raise(SIGINT);
```

Sending Signals via Keystrokes

- Three signals can be sent from keyboard:
    - Ctrl-C → 2/SIGINT signal
        - Default handler exits process
    - Ctrl-Z → 20/SIGINT signal
        - Default handler suspends process
    - Ctrl-\ → 3/SIGQUIT signal
        - Default handler exits process
        - Core dump may be saved

Process Control Implementation

Exactly what happens when you:

- Type Ctrl-C?
    - Keyboard sends hardware interrupt
    - Hardware interrupt is handled by OS
    - OS sends a 2/SIGINT signal
    - Signal handler for 2/SIGINT signal executes to completion
        - Default signal handler for 2/SIGINT signal exits process
- Type Ctrl-Z?
    - Keyboard sends hardware interrupt
    - Hardware interrupt is handled by OS
    - OS sends a 20/SIGTSTP signal
    - Signal handler for 20/SIGTSTP signal executes to completion
        - Default signal handler for 20/SIGTSTP signal suspends process
- Issue a “fg” or “bg” command?
    - OS sends an 18/SIGCONT signal (and does some other things too!)
    - OS sends an 19/SIGCONT signal

Signals generated by abnormal events

- Example: process makes illegal memory reference
    - Event gains attention of OS
    - OS stops application process immediately, sending it a 11/SIGSEGV signal
    - Signal handler for 11/SIGSEGV executes to completion
        - Default signal handler for 11/SIGSEGV signal prints “segmentation fault” and exits process

Responding to Signals

- A process can:
    - Carry out the default action for that signal

    - Ignore the signal by setting a mask (sigprocmask())
    - Catch a signal with signal handlers (signal() or sigaction())
        - And then possibly resume execution or terminate
- A process cannot ignore and/or catch SIGKILL and SIGSTOP

sigaction

```
#include <sys/time.h>
int sigaction(int signum, const struct sigaction*act,
                  struct sigaction*oldact);
```

- signum: specifies the signal to change
- act: new action for signal
- oldact: the existing action

```
struct sigaction{
      // the signal handler is either sa_handleror sa_sigaction
      void (*sa_handler)(int);// only takes one parameter
      void (*sa_sigaction)(int, siginfo_t*, void *);// 3 parameters
      sigset_tsa_mask;// mask of signals to be blocked in sig.    handler
      int sa_flags; // Flag SA_SIGINFO selects sa_sigaction
      void (*sa_restorer)(void);
};
```

Block signals in signal handler

- During the execution of signal handler, the same signal is blocked
- Additional signals can be blocked by setting mask sa_mask in struct sigaction

Example:

```
sigemptyset(&sa.sa_mask);// clear the set
sigaddset(&sa.sa_mask, SIGINT);// block SIGINT
sigaddset(&sa.sa_mask, SIGQUIT);// block SIGQUIT
```

Note: Many functions cannot be called safely in a signal handler

"Man signal-safety" for a list of safe functions

Example Problem

- What if the user types Ctrl-C?
    - OS sends a 2/SIGINT signal to the process
    - Default handler of 2/SIGINT exits the process
- Problem: temporary files not deleted
    - Process dies before remove("tmp.txt") is executed
- Challenge: Ctrl-C could happen at any time
    - Which line of code will be interrupted???
- Solution: Install a signal handler

- Define a “clean up” function to delete the file
- Install the function as a signal handler for 2/SIGINT

Examples

Catch a signal (in demo code x1.sigaction)

- Set a timer (in demo code x1.timer)

```
alarm()
setitimer()
timer_create()
```

Alarms

```
#include <unistd.h>
unsigned intalarm(unsigned intseconds);
```

- Sends 14/SIGALRM signal after `seconds` seconds
- Cancels pending alarm if `seconds` is 0
- Uses real time, a.k.a. **wall-clock time**
  - Time spent executing other processes counts
  - Time spent waiting for user input counts
- Used to implement time-outs
- Better to use setitimer() or POSIX timers

Interval Timers

```
#include <sys/time.h>
intsetitimer(intwhich, conststructitimerval*new_value,
                       structitimerval*old_value);
intgetitimer(intwhich, structitimerval*curr_value);
```

- What gets counted is specified by which
  - **ITIMER_REAL**, decrements in real time (wall time), delivers SIGALRM
  - **ITIMER_VIRTUAL**, decrements only when process is executing (CPU time), delivers **SIGVTALRM**
  - **ITIMER_PROF**, decrements when process is executing or OS is executing on behalf of the process, delivers **SIGPROF**

POSIX timer

```
#include <signal.h>
#include <time.h>
// link with -lrt
inttimer_create(clock_tclockid, struct sigevent * sevp,
                                timer_t * timerid);

inttimer_settime(timer_ttimerid, intflags,
                const struct itimerspec * new_value,
                const struct itimerspec * old_value);
```

- Create a timer first and then settime
- clockid can be CLOCK_REALTIME, CLOCK_MONOTONIC, etc.
- When timer expires, deliver a signal, or create a thread
    - Or check manually