2 Introduction to C

C Modules

- file translated into obj. file, which gets linked by linker to other object files and std libraries
- can refer to global variables/functions of other modules via. externs

Char input/output

- `getchar(void)`
- `putchar(c)`

`printf()` string formats

- `%.2f` floating pt to 2dp
- `%p` pointer

`scanf()`

- reads from std input
- returns number of read items
- parameters must be pointers

2 Introduction to Unix

`file`

- determines type of file
- e.g. ordinary, directory, device, 'special'

Shell environment

- at login, reads from `/etc/profile`
- gets `.bash_profile`, `.profile`

Permissions

- user, group, other
- read, write, execute
- use magical numbers (r: 4, w: 2, x: 1)

Redirections

- `a.out < data >res 2>errors`
- appending: `>>`

Shell scripts

- `#!/bin/bash`
- default search path `$PATH`
- to execute a script, `./scriptname`, otherwise if the current dir is in `$PATH` it can be executed using `scriptname`

Shell

- UNIX cmd interpreter
- reads in cmds, runs appropriate programs

I/O redirection

- when prog runs, 3 std files opened
  0 std input
  1 std output
  2 std error

Shell variables

- stored in environment of the program
- setting: `VARNAME=value`
- using: `$VARNAME`
- script arguments: `$1, $2` etc.

`if` statement

-
  ```
  if <cmd>
  then
      <cmd>
  fi
  ```

`while` loop

-
  ```
  while <condition>
  do
      <cmd>
  done
  ```

`for` loop

-
  ```
  for <condition>
  do
      <cmd>
  done
  ```

`case`

-
  ```
  case $selector in
  1) <cmd> ;;
  2) <cmd> ;;
  esac
  ```

UNIX cmds

- `test`: tests a condition, exists with true/false
  `if test $1 == "blah"`
- `sort`: sorts lines of text in a file
- `cut`: cuts selected parts of lines of text in a file, and sends result to output
- `tr`: changes or removes chars from a file
- `comm`: compares files and prints lines that exist in only one or both files
- `grep`: searches text file/output, matching each line against specified regex, and prints all lines that match
- `diff` and `sdiff`: comparing files

Cmd substitution

- arg enclosed in backquotes indicates that a command is to run, and the output used as the actual argument(s)
- `prog ‘cat argfile‘`

Subshells

- run cmds in another copy of the shell
- environment copied from parent - subshell can change environ., but it will be reverted when the subshell exits
- `tar cf  mydir | (cd *loc*; tar xf -)`

Collecting output

- `(echo data; cat filename) > output`

Arithmetic

- `expr` evaluates its args as an expression
- `let` for assignment of variables
- `let count = count +1`

Read text from shell

- `read x`: reads in line from std input, and stores as x
- ”here document”

Finding files

- `find`: starts at curr dir and searches recursively
- `locate`: prints the full path names of all files that match
- `du`: prints disk usage starting at curr dir

Strange file names

- to open file named `-x`, use `nano ./-x`
- `--` to stop parsing flags
  `rm -- -x`

3 Pointers

- a memory address
- obtain address of variable with `&`
- create pointer to the address of `initial`
  `char initial = ’A’;`
  `char *initial_ptr = &initial;`
- `*` pointer to variable of specific type
- `**` unravels indirection
- `char msg[] = "message";`
  `char *string = &msg[0];` (or `msg`)
  ∴ `msg[1] == *(string+1)`
- Iterating through a string with a pointer
  `while (*str != ’\0’) { str++; }`

Dynamic data structures

- dynamically allocate memory
- `malloc`, `realloc` etc.

Indirection operator

- declaration: pointer to specified type
  `int * ptr`
- dereferencing: dereferences the pointer to mean the content/value of the variable being pointed to

Array processing

- `int array[10]`
  `array == &array[0]`
- variables can change their values, but not their addresses
- pointer’s value is the address of another variable, ∴ arithmetic ops permitted on pointer

Pointer scalars

- mathematical operations on pointers work regardless of the data type being pointed to
- ptr accesses to arrays will always move the correct number of bytes

Pass by reference

- swaps addresses of initial variables

-
  ```
  void swap (int *a, int *b) {
      int tmp = *a;
      *a = *b;
      *b = tmp;
  }
  ```

Pointers to pointers

- multiple indirection
- `argv[][] == *argv[] == **argv`

`void` pointers

- type signature of the pointer does not specify what it's pointing to
- can recieve/return ptrs of any type
- `void *malloc(size_t size);`

Function pointers

- refer to 12. string handling
- allows for selection of program behavior

`NULL` pointers

- pointer to something at address '0'
- invalid to read or write to that address ∴ god

3 Aggregate Data Structures

enums

- associates name to a value
- maps to an int

-
  ```
  enum day_name {
      mon, tue, wed, thur, fri, sat,
  sun
  }
  ```
  maps to ints 0..7
- can then use `sun++;`

Structures

- for a collection of data items of different types

-
  ```
  struct <tag> {
      <member-declarations>
  }
  ```
- declare: `struct <tag> <identifier-list>;`
- access: `<identifier>.<element>;`
- if a pointer to a struct is used, `->` operator is used to get an element in a struct

4 Source Code Control

Issues

- version control
- managing several versions of a program
- allows you to maintain current version whilst working on the next

Control

- checkin/checkout system
- e.g. svn, git, hg

Mercurial: hg

- distributed
  1. make copy of existing repo
  2. push changes to others
  3. pull changes from others
- `hg init`: creates repo
- `hg diff -r2 -r3`: diff b/w revision 2 and 3
- `hg revert -r2 code.c`: revert file to r2
- `hg push/pull/clone <repo>`
- repo: another dir/URL to a remote repo

4 `make`

- program can use many `.c`, `.h` files that require compiling
- time consuming to compile lots of files separately
- object file: machine language, but not yet linked with other parts of the program
- several `.c`, `.o` files can be combined to give an executable program via. linkage
- after changing one `.c` file, you need to recompile affected file and relink ∴ `make` is sexy

Rules

-
  ```
  prog.o: prog.c prog.h      dependencies
      gcc -c prog.c          action
  ```
- `<target>`: name of file to be made
- `<1+ dependencies>`: files the target file depends on
- `<action>`: shell cmd that creates target
- default rules are the bomb
- can combine rules when targets have common dependencies and actions
- can create rules without dependences
  `clean:`

```
rm *o
```

`make` variables

- assignments: `variable_name = value`
- use: `$(variable_name)`

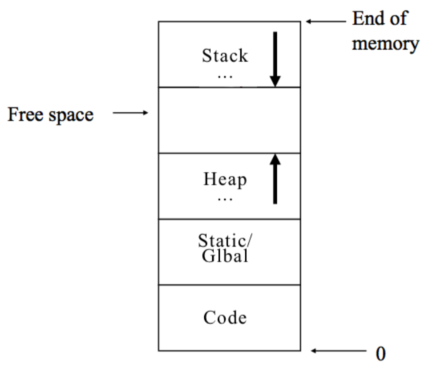
Predefined variables

- `CC`: default C compiler
- `CFLAGS`: flags passed to the C compiler

Libraries

- gives you the ability to store the object code versions of the functions in one place and have them linked into your program
- stdlib automatically searched when prog is linked
- use functions from other libraries using `-l` flag when linking
- the C compiler will search for the library in standard directories: `/lib, /usr/lib`
- create own library using `ar`, which makes library mylib.a and will contain specified .o files
  `ar c mylib.a readit.o util.o`
- can then use created library when compiling
  `gcc myprog.o mylib.a -o myprog`

5 Memory Management



Memory areas

- code: program instructions
- global/static: global/static variables
- stack: local variables, function arguments, return addresses, temporary storage
- heap: dynamically allocated memory

Stack

- all variables local to a function and function args stored on the stack
- to call func:
  1. push args to stack
  2. push return address to stack
  3. jump to function code
- inside function:
  1. increment the stack pointer to allow space for the local variables
  2. execute the code
  3. pop local variables and arguments off the stack
  4. push the return result onto the stack
  5. jump to return address

Heap

- accessed under direct control
- request allocation, if there is sufficient contiguous memory available, a pointer is returned to the address of the stage of that memory block

Memory allocation functions

- returns a pointer to void
- pointer must be cast to a specific type

`malloc`

- `void *malloc(size_t size)`
- requests number of bytes of memory

`calloc`

- `void *calloc(size_t num, size_t size)`
- requests number of blocks of memory, and the size of each block
- allocated memory is cleared i.e. set to '0'

`realloc`

- `void *realloc(void *ptr, size_t size)`
- takes previously allocated memory, and attempts to resize it
- contents are preserved
- may require new block of memory (for contiguous-ness) ∴ new void pointer is returned

`free`

- `void free(void *ptr)`
- deallocates memory previously allocated
- `valgrind`: check for leaks

to make program happy

- check memory allocation for success (NULL pointer is returned if unsuccessful)
- don't free memory that has already been freed or was never allocated

5 Preprocessor

- `#include "decs.h"` > copying declarations into every file
- useful for externs, tyedefs, struct definitions

Defined symbols

- replace identifier with string, everywhere it appears in the program
- can be any string of characters, ∴ should bracket arithmetic expressions

Macros

- `define min(a,b) ((a) < (b) ? (a) : (b))`

Conditional inclusion

- `#ifdef`, `#ifndef`, `#undef`
- `#if`, `#elif`, `#else`, `#end`

Predefined symbols

- `__LINE__`: current line number at any point
- `__FILE__`: name of current program file

Preprocessor ninjaness

- `gcc -E` runs only preprocessor
- exploit `#define` call by name, and `#ifdef` etc. for conditional generation of hacky templates

6 Unions and Bitfields

Unions

- variables that occupy the same space
-
```
union {
    struct {
       /* struct guts */
    } type_one;
    struct {
       /* struct guts */
    } type_two;
} info;
```
- access elements: `union_name.part_name`
- don't know which variant of the union is being used, ∴ need to use a separate variable to indicate this
-
```
struct catalog x;
switch (x.holding_type) {
    case book:
       <do stuff>
       break;
    case film:
       <do stuff>
       break;
}
```

Bitfields

- can specify the size of bits for each element in a structure
- size placed after field name, with a colon
-
```
struct IOdev {
    unsigned RW: 1;
    unsigned Dirn:  8;
    unsigned mode:  3;
    unsigned pad:  4;
}
```
- may not be portable:
  - depends on compiler whether structs are padded
  - endianness: whether fields are stored MSB or LSB
- unpack bitfields from data:
  - don't need C bitfield syntax
  - shift `<<` `>>` and logical operators `&` ∼ ∧ `|`
  - `R_W = x >> 15;`
  - `Dirn = (x >>7) & 0xFF;`

6 Linked Lists

Internals

- terminated by the NULL pointer
- have pointer to the first element
- don't lose your nodes, that'd be awkward

Characteristics

- altering nth element:
  - array: O(1)
  - linked list: O(n)

7 Parallelism

Origin

- require continuous/reasonable performance improvements to support growing architecture
- portability, malleability, maintaining

Moore's Law

- number of transistors that can be placed on a circuit is doubling approximately every 2 years
- bottlenecks: power density, wire delays, DRAM access latency

Hardware/software development

- historically, transistors used to boost performance
- now, more cores per chip, ∴ more, not faster processors
- can't find solution to obtain substantial performance improvement of single core CPUs
- require new parallel architecture

Task parallelism

- processing the task in parallel
- dependencies between tasks - some must be processed before other tasks can occur
- task dependency graph
  e.g. directed acrylic graph (DAG)

$$\begin{tikzcd}
t_1 \arrow[d] & \\
t_2 \arrow[dr] & t_3 \arrow[d] \\
& t_4
\end{tikzcd}$$

Data parallelism

- parallel work on the data of a task
- i.e. performing the same task to different data items at the same time

Task

- computation that consists of a sequence of instructions
- a distinct part of programs/algorithms, as they can be decomposed into tasks

Dependencies

- execution order between 2 tasks
- no dependencies = maximum parallelism

$$\begin{tikzcd}
t_1 & & t_3 & \\
& t_2 & & t_4
\end{tikzcd}$$

- dependencies

$$\begin{tikzcd}
& t_1 \arrow[dl] \arrow[dr] & \\
t_2 \arrow[rr] & & t_3 \arrow[d] \\
& & t_4
\end{tikzcd}$$

- impose partial ordering on tasks
- dependency is transitive
  - if Ta → Tb and Tb → Tc, then Ta → Tc

Stream programs

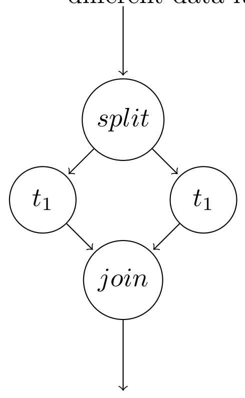
- some programs work on streams of data (audio, video etc.)
- used for regular, repeating computations
- pipeline parallelism:
  - pipeline: sequence of actors
  - reads input from upstream data channel, outputs to downstream channel
  - producer-consumer relationship (have dependencies)
  - stateful actors can be parallelised

$$\begin{tikzcd}
{} \arrow[r] & t_1 \arrow[r] & t_2 \arrow[r] & t_3 \arrow[r] & {}
\end{tikzcd}$$

- task parallelism:
  - between actors without producer-consumer relationship (no dependencies)
  - parallel execution of different tasks

$$\begin{tikzcd}
& {} \arrow[d] & \\
& \mathit{split} \arrow[dl] \arrow[dr] & \\
t_1 \arrow[dr] & & t_2 \arrow[dl] \\
& \mathit{join} \arrow[d] & \\
& {} &
\end{tikzcd}$$

- stream parallelism:
  - for stateless actors (don't remember information between executions)
  - parallel execution of the same task on different data items



Implicit parallelism

- automatic, done by compiler
- speed up is limited

Explicit parallelism

- done by programmer, but they need to understand the program
- decompose it into tasks
- understand hardware to get decomposition that will fit it
- take care of communication/coordination between tasks

Notes

- loops executing in parallel need their own loop index
- SIMD vectorisation (SSE)
- scalar register - a register of a conventional CPU that can only hold one data item at a time
- vector processors have large registers that can hold multiple values of the same data type
- SSE can apply an operation to all elements of a vector at once

7 POSIX Pthreads

- `<pthread.h>`
- compile: `-lpthread`
- `pthread_t thread_a`
- `pthread_join()`: waits for created threads to terminate, main function is blocked
- `pthread_create()`

Execution indetermination

- no assumptions about execution order can be made about threads executing in parallel
- enforce order via. synchronisation

Thread-safe routine

- function/lib routine/system call is thread safe if it can be called from several threads simultaneously and produce correct results
- `printf()` is safe as output happens atomically
- serialises output

OS schedules threads

- uniprocessor: threads share single processor that will interleave execution of threads
  - illusion that threads are executing in parallel
- multicore processor: true parallelism
  - if num threads ¿ num cores, OS will multiplex between threads
  - at any one time, more than one thread is being executed

Shared address space

- a global passed to thread, is visible in both

Thread creation arguments

- thread ID: variable used to refer back to created thread
- thread attributes: NULL is default
- function: function that thread will execute once created
- args: get passed to function at run-time
- threads are peers, may create other threads, no assumption on execution order

Thread termination arguments

- thread ID: for thread parent is waiting for
- completion status: gets copied unless NULL, where it is ignored
- once joined, thread ID invalid, as thread no longer exists

Passing arguments to thread

- same thread function used for each thread between the different threads

- 1+ arguments: create struct, then pass a pointer to struct the struct variable
- `pthread_create(&thread_ID, NULL, &func, (void*)&multi_arg_struct)`

Thread termination

- Ways thread can be terminated:
  - thread returns from start routine
  - thread calls `pthread_exit()`
  - thread cancelled by another thread
- when thread destroyed, resources unavailable
- then you gotsta `free()`
- close files opened by threads
- when `main()` terminates, so do all threads
- `pthread_exit()` can be used to pass on exit status to any thread that will join the existing thread
- value of exit status will be (void *)

---

8 Synchronisation

Mutual exclusion

- serialises access: only one thread must be in the critical section at any time
- race conditions due to interleaved execution of process access to critical section would result in incorrect results

Race conditions

- multiple threads read and write a shared data item, and the final result depends on the relative timing of their execution
- shared resource
- critical section: 2+ code parts that access and manipulate shared data
- prevents mutual exclusion

Mutexes

- `pthread_mutex_t lock = PTHREAD_MUTEX_INITIALISER`
- synchronise access to shared global variables
- entering critical zone: lock mutex `pthread_mutex_lock(&lock);`
- leaving critical: unlock mutex
- initial state: mutex free/unlocked
- calling `lock()` means thread acquires lock
- while mutex is locked, other threads calling `lock()` are blocked
- blocked threads sit in a queue
- `trylock()` doesn't block threads in a queue if lock isn't free, it'll return immediately
- use return value of `trylock()` distinguishes between the 2 locks

Dynamic creation of mutex

- create mutex at runtime
- `pthread_mutex_init()`: initialise mutex
- `pthread_mutex_destroy()`: atomic bomb
- `free(lock)`

Monitors

- encapsulates access to shared data structures
- hides counter variable from direct access
- only way to access counter is via. function, which already has counter in a mutex
- access via function call is less efficient

Serialisation

- threads execute critical section one after the other
- place computations that don't access shared data outside of critical

Deadlocks

- involves 1+ threads and 1+ resources, such that each thread is waiting for one of the resources, but all resources already held
- self-deadlock: thread attempts to acquire a mutex it already possesses
- ABBA deadlock: A→B, B→A 2 threads attempt to acquire each other's mutexes
- prevent by obtaining multiple mutexes in the same order, unlock operation is immaterial
- conditions necessary for deadlock:
  1. mutual exclusion: a resource can be assigned to at most one thread
  2. hold and wait: threads both hold resources and request other resources
  3. no preemption: a resource can only be released by the thread that holds it
  4. circular wait: a cycle exists in which each thread waits for a resource that is assigned to another thread

Lock contention

- lock in use, but another thread tries to use it
- highly contended lock limits parallelism
- bottleneck due to serialised activity
- limits scalability (measure of how well a system can be expanded)

Lock granularity

- coarse, medium, fine
- describes amount of data that a lock protects
- coarse → fine, if lock contention becomes a problem
- locking/unlocking adds overhead

Barriers

- synchronising point at which a certain number of threads must arrive
- `pthread_barrier_t`
- participating threads call `pthread_barrier_wait`: all threads block until specified number of threads have called it
- `pthread_barrier_init(&barrier, NULL, 3)`

Semaphores

- non-negative integer synchronisation variables
- works for mutual exclusion and thread synchronisation
- `#include <semaphore.h>`
- P(s): thread must wait until s > 0, then `s--` and allowed to continue execution
- V(s): increment s
- binary semaphores:
  - `sem_init(&s, 0, 1)`: 0 = unlocked
  - initialised at 1 initially
  - behaves like mutex
    `sem_wait(&s) <critical> sem_post(&s)`
- counting semaphore:
  - initialised at N > 1
  - allows N threads into critical

Condition variables

- "spinning": consumes CPU cycles while it circles through loop waiting for something to happen
- waiting: `pthread_cond_wait(condition, mutex)`
  - blocks thread until signal recieved
  - must be called while mutex locked
  - releases mutex while it waits, once condition signalled, the mutex is locked again
- signalling: `pthread_cond_signal(condition)`
  - wake up at least 1 thread
  - must be called after mutex locked, and unlock mutex aftwards
  - `pthread_cond_broadcast(condition)`: wakes all threads blocked at condition (bottleneck if there is contention for resources)

Critique of lock synchronisation

- locks don't compose
- tedious for large systems
- while holding locks, problematic to make calls to unknown code, as called code might acquire locks also

Monitors vs. (semaphores/mutexes)

- semaphore/mutex use voluntary, ∴ can forget to use the s/m associated with a shared data item and introduce a race condition
- deadlock: violation of locking hierarchy
- monitor = shared data + mutual exclusion/sync. mechanism
- safer and more flexible
- assumed behaviour: only one thread can access shared data at any one time
- slightly more inefficient

Problems with thread synchronisation

- deadlocks
- livelocks: two or more threads are busy synchronising and don't make progress
- starvation: one thread never allowed into critical section, ∴ require fairness property to ensure every thread can make progress

9 Performance

Limits to performance scalability

- programs have parallel and sequential parts
- sequential: have data dependencies

Amdahl's law

- potential speedup defined by the fraction of the code that can be parallelised
- $speedup = \frac{oldrunningtime}{newrunningtime} = \frac{1}{(1-p)+\frac{p}{n}}$
- sequential part limits scalability
- p: fraction of work that can be parallelised
- n: num threads
- if n → ∞, speedup $= \frac{1}{1-p}$
- p = 0, embarrassingly sequential
- p = 1, embarrassingly parallel

Performance scalability

- linear speedup

- not always achievable due to overhead etc.
- $efficiency = \frac{speedup}{n}$
- e = 1, linear speedup
- e > 1, super linear speedup (possible due to registers and cache)

Load balancing

- granularity of parallelism = frequency of interaction between threads
- fine: high communication/synchronisation overhead $\therefore$ less opportunity for performance improvement
- coarse: low communication/synchronisation overheard, harder to load balance effectively
- threads that finish early have to wait for the thread with the largest amount of work to complete
- idle times lower process utilisation
- load imbalance: work unevenly assignment to cores, underutilises parallelism
- assignment of work, not data
- static assignment: more prone to imbalance
- dynamic: quantum of work must be large enough to amortise overhead, allows for solving of load imbalance

Measuring performance

- wallclock time
- `time`: real (elasped), user (time executed in user mode), sys (time executed in kernel mode)
- `gettimeofday()`: measure wallclock time for specific parts of the program

False sharing

- private sum variable ≠ private cache line
- force into different cache lines using padded variables

Profiling

- program instrumentation and execution
- CPU provides performance counters to measure various elements
- Heisenberg effect: measuring can perturb a program's execution time

Source of performance loss

- overhead: communication, synchronisation, computation, memory
- non-parallelisable computation
- idle times: lack of work, waiting for external event, load imbalance, memory bound computations
- contention for resources

Performance trade-offs

- communication vs. computation: often possible to reduce communication overhead by performing additional computations
- memory vs. parallelism: increase parallelism can often be increased at cost of increased memory usage (privatisation, padding)
- overhead vs. parallelism: parallelisation overhead, load balance vs. overhead, granularity trade-offs

9 Scalable algorithms

Recursion

- procedure calls itself to solve a simpler version of problem, terminates at base case

Divide and conquer

- break problem down into subsections, solve recursively, and combine solutions to create solution to the original problem
- mergesort

Reduction

- Schwartz algorithm: scalable
- reduces a collection of data items to a single, by repeatedly combining the data items pairwise with a binary operator
- op usually associative and commutative
- reduction ops: + *
- reduction of shallow tree contains less computations, not much faster than sequential
- deeper trees = more speed-ups

Fixed parallelisation

- generate until hard coded number of threads, then continues recursively
- more efficient than unlimited, doesn't scale

Scalable parallelisation

- switch to sequential depending on number of cores
- most efficient
- scales to higher number of cores
- algorithms harder to program for

Unlimited parallelisation

- generate threads with each dividing step
- highest amount of logical parallelism
- bad nearer base case

10 Pipes and signals

Process communication

- `exec`: start process
- `fork` → `exec`: processes operating in parallel
- pipes and signals: communication

File descriptors

- small integers
- low level I/O to operate on these
- 0: std in
- 1: std out
- 2: std error

Pipe

- `int pipe(int filedes[2])`
  - 2 elem array of integers
  - 0: writing
  - 1: reading
- pipe returns 0 (success), -1 (failure)
- parent to child communication by creating a pipe before fork
- small buffer: when filled the writer is suspended until more data has been read

Signals

- interprocess communication
- form of software interrupt, can be generated by OS
- exec interrupted, function call made to user specified function; exec resumes when function returns

Kill

- send signal to process from cmd line
- some signals can be caught and handled, but some can't (`SIGKILL`)
- `kill -9 12345`

Catching signals

- catch by specifying function to be called once signal is recieved
- `signal(signame, ptr to func)`
- returns a ptr to function that previously caught the signal

10 Processes

Process initiation

- functions that involve system calls
- set of these functions let you initiate and manage the running of other processes
- shell uses these to start programs corresponding to cmd given
- starting another process: `execl`, `execv` etc.
- after process is started, the main function is called
- `exec`: switches program execution to another
  - program terminated and main function of other program is called
  - no return: exec successful
  - -ve return: program not found
  - 0+ return: exec function failed

Parallel execution

- `exec` is like a GOTO: jumps to other program and doesn't return
- forking allows for execution of another program while still continuing execution of current
- `fork`: creates child process that is a copy of the memory image of a parent
- return value of fork function is different for parent and child, by checking this value, running program can determine its identity
  - 0: child
  - PID of child: parent
  - -1: in parent process if fork fails
- parent can ignore or wait for child to exit

`wait`

- parent can wait until child exists and get the exact value
- waits for a child process to exit
  `int s = 0;`
  `wait(&s);`
- returns the PID of child process
- exit value can be extracted from status value s, other info in value indicates if child failed or was terminated
- `waitpid()`: waits for specific process to exit