CSS503A Lecture #3 2014-04-04

Administrivia

Our Story So Far (with a bit of review stuff to keep you awake)

- operating systems operate at lower levels of abstraction
  - closer to the metal
  - performance is critical

- kernel operates in "supervisor" mode
  - application programs make system calls for services
  - timer interrupts processing so kernel has chance to do housekeeping tasks & schedule other processes (timeslice)
  - context switch to kernel for hardware interrupt

  Context switches

  Service hardware interrupts

- init (process #1): system starter
  - secondary function: reap zombie orphans

- kernel concerns
  - process management
  - filesystems
  - device drivers
  - networking
  - memory management
  - accounting
  - protection/security
  - error detection

OSSF (cont.)

- process: (essentially) running program
  - executable file
  - argv
  - pre-opened files

  ⟹ exit status { 0 – success/true; non-zero – fail/false }

  no intrinsic meaning to system. means whatever the receiving program wants it to mean – but: conventions

- convention:
  - 0 standard input (cin)
  - 1 standard output (cout)
  - 2 standard error (cerr)

  most programs expect these fds to be pre-opened
  languages bind fds to their formatted/buffered I/O library e.g. cin/cout/cerr

- C/C++ provide standard libraries for formatting & buffering.
  - actual I/O operations performed by read/write system calls

OSSF (cont.)

- System call/trap
  - typically: arguments stored in registers (ints & pointers)
    result in register: int
    -ve: fail
    0: ok, if no other meaningful result
    ≥0: result (e.g. fd, pid)

- global variable errno holds integer error code } raw info
  - perror: library function

OSSF (cont.)

Unix/Linux/Posix: separate process creation from program invocation

```
pid_t pid = fork()
```

-> clones process

- <0 : fail
- =0 : child process
- >0 : parent process (return result is pid of child)

- child process inherits same open files as parent

- parent collects exit status of child: wait/waitpid

=> process free (new)

- init { adopts orphans / inherits orphans

- fork is useful for distributing work
  - assignment 1

- But what if you want to run a different program?
  - stay tuned

process free

OSSF (cont.)

inter-process communication

- Pipe: original IPC mechanism
  - producer - consumer
  - producer writes data to memory buffer in kernel
  - consumer reads data from buffer
  - when buffer is full, writing is blocked
  - when buffer is empty, reading is blocked*
    - * unless non-blocking I/O flag is set

Both processes operate concurrently

- pipe is unidirectional
  - for bidirectional communication, use 2 pairs of endpoints
  - but watch out for deadlocks (new)

- pipe instance is file descriptor (small integer)
  - use same read/write system calls
  - file descriptor is object (in C++ sense)
    - but implemented in C
    - uses table of function pointers, just like C++ vtbl

new

- From a programming POV, pipe behaves just like any ordinary file
  - System call to set non-blocking I/O flag

Pipes (cont.)

- popen system call returns pair of open file descriptors for producer (writer) & consumer (reader)
  - each fd is one-way (endpoints)

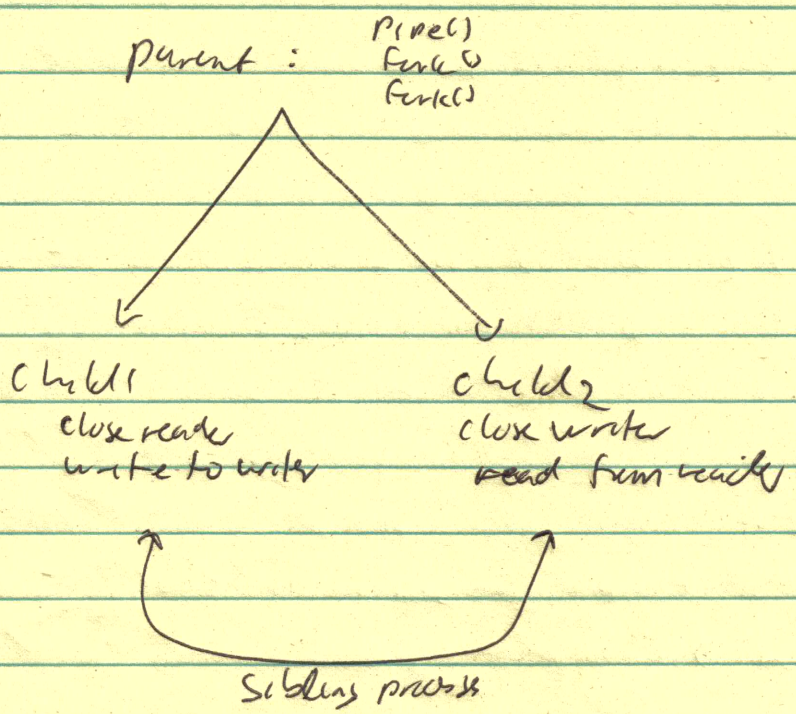
- single process can't really use pipe descriptors
  - read empty pipe would block, then you're stuck (similarly, write until full, then blocks)
  - besides, why would you? Why would you do such a thing

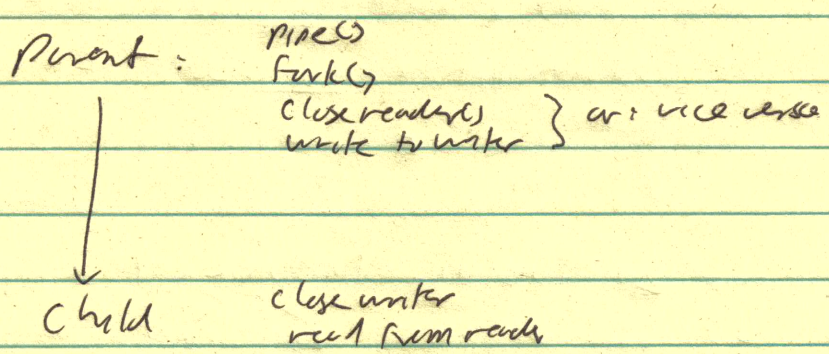
- file descriptor is index into kernel data structure
  - table is per process (each process has fd 0, 1, 2, ...)
  ⇒ so how do you get a pipe across two processes?

- files stay open across fork
  - two processes that share a pipe must have a common ancestor that created the pipe

- typically, one process closes the reader & keeps the writer while the other process does the opposite

Pipes (cont.)


parent :
pipe()
fork()
fork()
child1
close reader
write to writer
child2
close writer
read from reader
Sibling processes


parent :
pipe()
fork()
close reader()
write to writer
} or : vice versa
child
close writer
read from reader

Pipes (cont.)

- Shell syntax: do-this | do-that
  - pipe symbol (vertical bar)
  - also used for OR operator

- Shell only supports pipelines with standard input / standard output (FD 0/1)
  (can redirect stdout to stdin - shell magic)
  - pipe system call allows more complex usage patterns (not stdin/stdout)
    ⇒ subject of assignment #1

"Everything Is a File"

- basic file ops: read byte(s), write byte(s)
  => that's a good abstraction for a lot of stuff
  - ordinary file
  - tty (keyboard/screen) - dumb terminal
  - pipe (original IPC mechanism)
  - network connections
  - speakers, microphone, camera
  - blinking lights
  - pseudo devices (/dev/null /dev/zero /dev/random)
  - other devices

- stdin ≠ not keyboard
  stdout ≠ not screen
  } default for interactive programs

- shell syntax for redirecting input/output
  - design principle: policy vs mechanism
  - > >> < |

Process Management (cont.)

- fork(2) allows a program to divide work across multiple processes, each running the same program
  - pipe is just one IPC mechanism
    - we'll look at others later in the quarter

* subject of assignment 1

- but what if you want to run a different program?
  ⇒ execve (plus several standard library variants that let you marshal arguments differently — all eventually call execve)

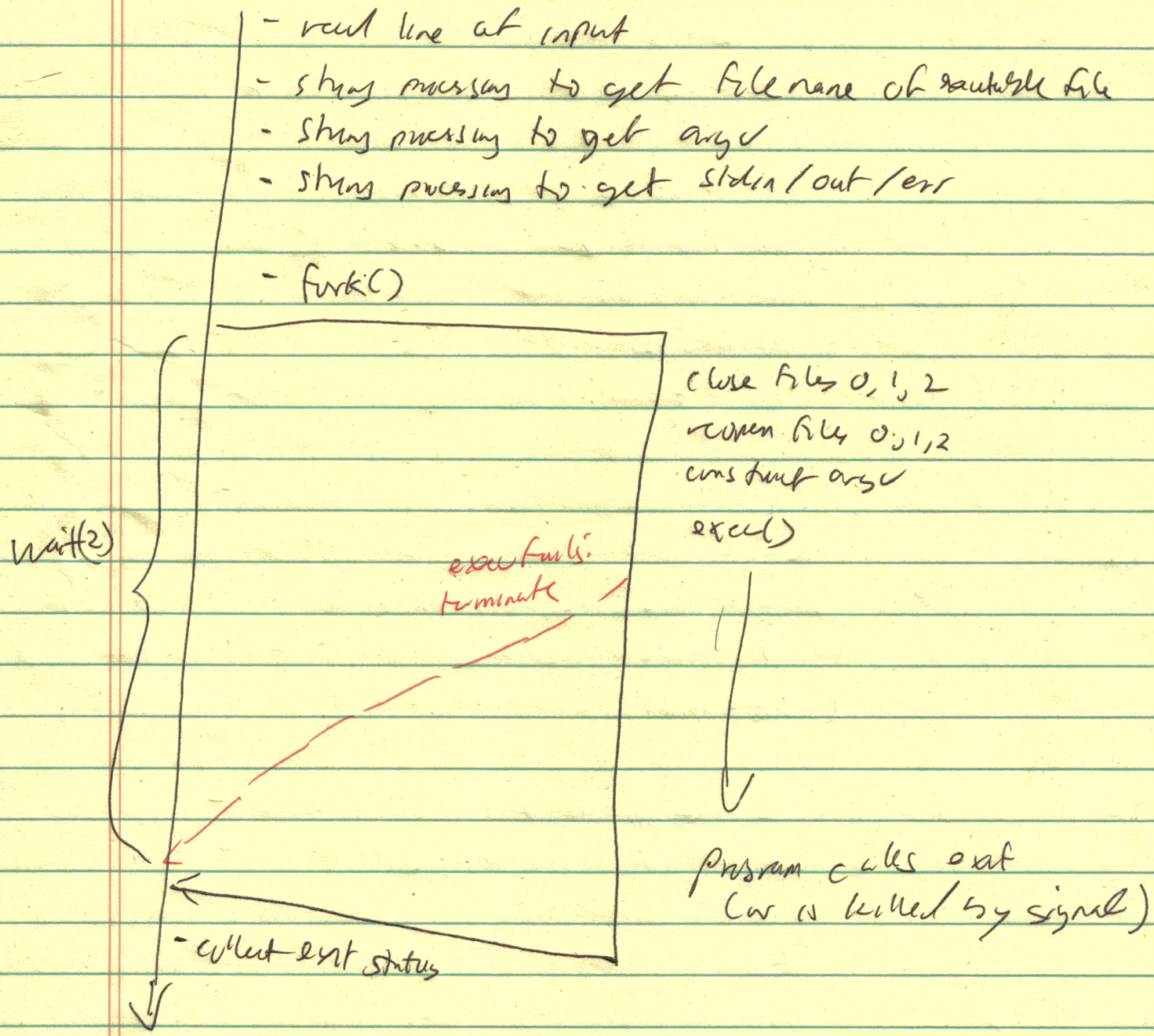
- exec* replaces current running program in process with new program, new argv
  * same process, new program
  - if call to exec is successful, it does not return (you can't go home again)

- open files (file descriptors) stay open across exec*
  *unless you set the file's close-on-exec flag

Shell

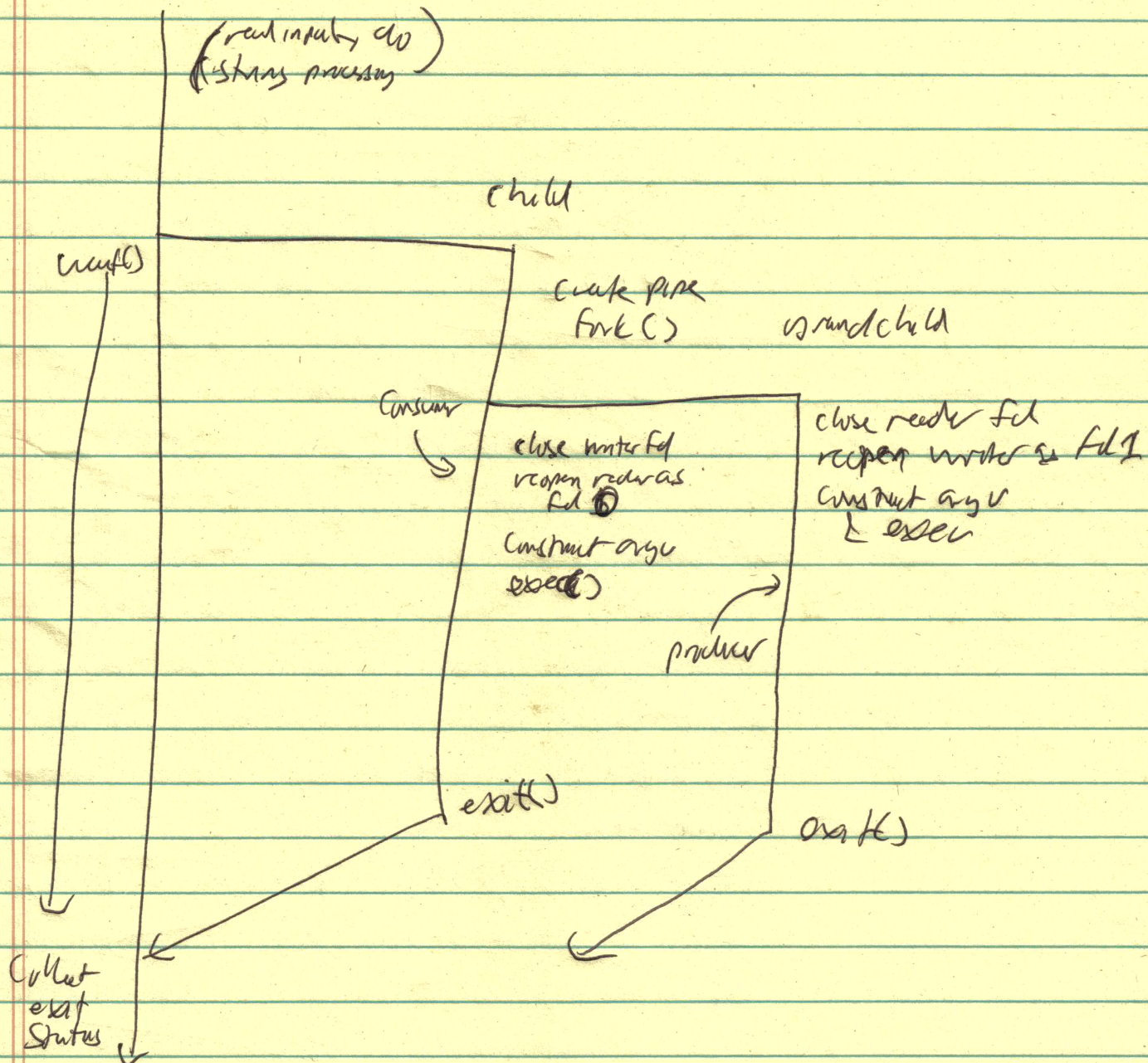
— Why separate calls for fork & exec?
⇒ allows shell & other programs to do some housekeeping after fork, before exec

- we now have enough information to construct a rudimentary shell



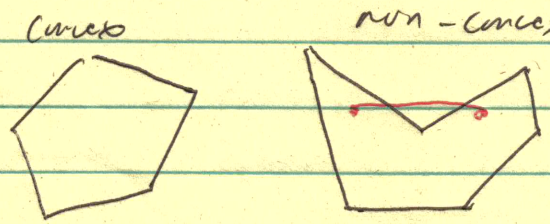
Shell (cont.)

- producer-consumer pipeline (simplified)



Assignment #1 : Convex Hull

- Convex Polygon

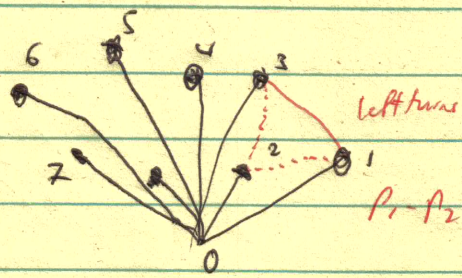


- convex hull
  - important (famous) problem in computational geometry
  - much-studied, many algorithms

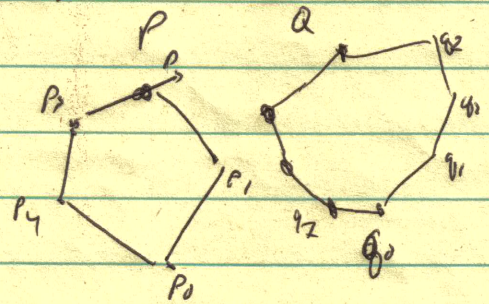
- graham scan algorithm
  - Sort points by angle wrt lowest point
  - tricks to avoid trig (OK, but expensive), divide-by-zero
  - graham scan : build stack of points, removing points that make right turns (only works in the plane - 2D)

- find min : O(N)
- sort : O(N log N)
- graham scan : O(N)

Assignment #1: Convex Hull (cont.)

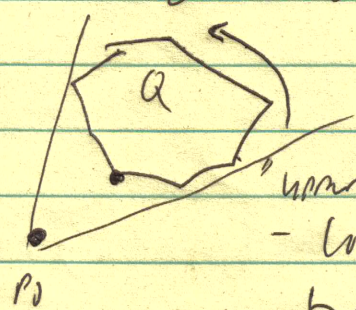


- Divide & conquer
  - given 2 Convex Hulls, find C.H of union



- Points in P are ordered around p_0
  Points in Q q are ordered around q_0

- assume p_0 is lower



upper & lower chains
- lower chain cannot be in merged hull
- upper chain is sorted around p_0

Assignment 1: Convex Hull (cont.)

- d & c algorithm
  - divide points into 2 sets
  - compute hulls of both sets
  - find upper chain of upper hull ($O(n)$)
  - merge with lower hull ($O(n)$)
  - graham scan of merged sets ($O(n)$)

- recursion (divide in half): $O(\log N)$
  → total running time $O(N \log N)$

- but the assignment is to fork()
  compute both halves in separate process
  - concurrent & independent operation
  - multicore architecture: 2 cores running at the same time
- child process must communicate results back to parent (pipe)

- demo program does everything except multiprocess part (Skeleton)
  → don't need to worry about the algorithm, just the systems programming part