10/10 Agenda: Fork/exec, wait, process

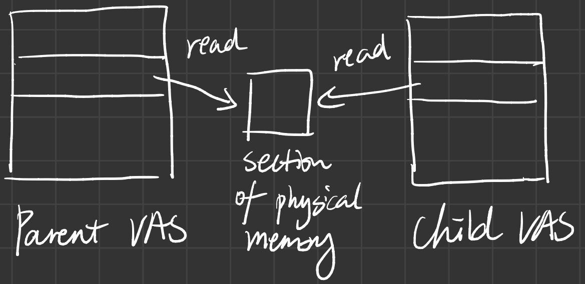
Fork / Exec

- Why?
  - → simple, fork() takes no argument, inherit environments, exec keeps the
  - → use case: shell redirection (ls > file.txt)
    - fork(), close stdout, open file.txt, exec("ls")

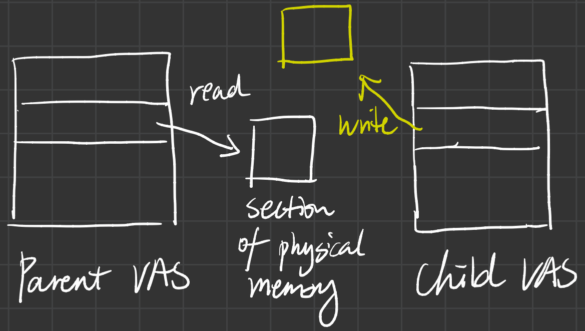
- Problems
  - copy the entire VAS is expensive!
    - ↳ need to allocate physical memory, set up the page table, and copy over all the content.

  Very Expensive!

- CoW Fork
  - Share the same physical memory for as long as possible.



☆ allocate memory, make the copy on write.



→ So how do we detect writes?
  - mark pages as read only, write will then cause an exception, make the copy when handling the exception (page fault)

? How to identify cow pages vs actual read only page.

→ Another variant: vfork()

- Create a new process and let it temporarily execute in parent's VAS, until it exits or calls exec()
- No copying whatsoever! Also no protection by the OS.

Faster than CoW Fork b/c no need to set up its own page table during fork

dangerous! child can modify parent's memory after vfork()

⇒ Additional Problem with fork

- semantic of fork is implicit inheritance
  → Simple, but difficult to add new services
  → not all services have a clear way of inheritance, not modular.

Alternatives: spawn(), clone().

Wait : wait for a child to exit

↳ waitpid : -1, any child
pid, specific child.

☆ kernel needs to track parent child relationship.

- Implementation

① In exit(), child needs to indicate its exiting status

② Child needs to free resources like its VAS, file descriptors

→ How about PCB & kernel stack?

↙ can't free, otherwise parent doesn't know child's status/state

↳ can't be freed by child, cause child is using it to execute in the kernel.

③ parent waits & reclaims the rest of child's resources.

→ Does the parent have to call wait?

shell : foreground (waits on), background & jobs (doesn't wait!)

→ Who reclaims resources for unwaited children?

☆ init process adopts all orphaned children!

(parent exited w/out waiting for children)

↳ the first process, started by the kernel during boot, creates many more processes according to some config files (starts ssh, shell ...)

Process Communication

- Interprocess Communication (IPC)
  → signals
- Pipes
  → pipe() returns 2 fds : read end fd & write end fd.
  → implemented as a kernel buffer.
  → shell usecase : ls | grep "a"
  pipe
- Files
- Shared memory
- Sockets

IPC

→ OS defines a set of signals (integers) for processes to send & recv

→ send via

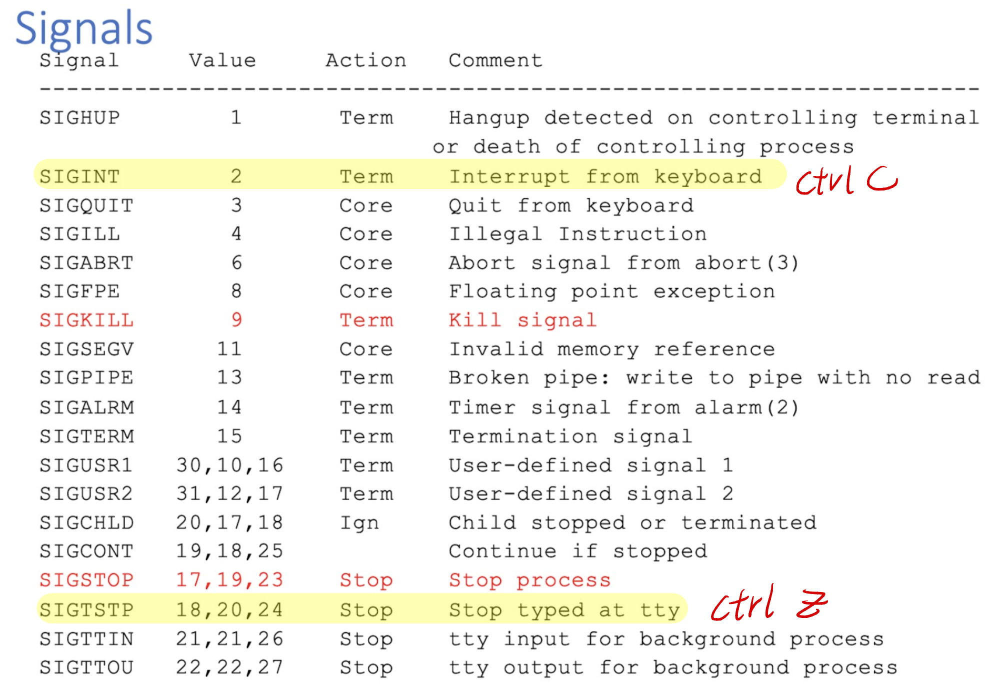
kill (pid, sig)

→ recv via

① default handlers from the OS

② custom installed handlers

eg. shell's sigint handler forwards the signal to foreground process



Signals

Signal	Value	Action	Comment	
SIGHUP	1	Term	Hangup detected on controlling terminal or death of controlling process	
SIGINT	2	Term	Interrupt from keyboard	ctrl C
SIGQUIT	3	Core	Quit from keyboard	
SIGILL	4	Core	Illegal Instruction	
SIGABRT	6	Core	Abort signal from abort(3)	
SIGFPE	8	Core	Floating point exception	
SIGKILL	9	Term	Kill signal	
SIGSEGV	11	Core	Invalid memory reference	
SIGPIPE	13	Term	Broken pipe: write to pipe with no read	
SIGALRM	14	Term	Timer signal from alarm(2)	
SIGTERM	15	Term	Termination signal	
SIGUSR1	30,10,16	Term	User-defined signal 1	
SIGUSR2	31,12,17	Term	User-defined signal 2	
SIGCHLD	20,17,18	Ign	Child stopped or terminated	
SIGCONT	19,18,25		Continue if stopped	
SIGSTOP	17,19,23	Stop	Stop process	
SIGTSTP	18,20,24	Stop	Stop typed at tty	ctrl Z
SIGTTIN	21,21,26	Stop	tty input for background process	
SIGTTOU	22,22,27	Stop	tty output for background process	