4/5/24

Processes & Scheduling

Process APIs : fork, exec, low fork (performance optimization)

inherits OS resources, every syscall needs to support behavior across fork

→ other APIs

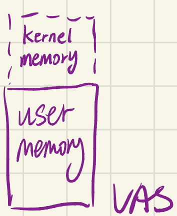
- → spawn : windows API that creates a new process running new code
- → clone : creates a new child process w/ precise control of what's inherited

Exit:

- terminate the current process
- need to clean up kstack, address space, & OS resources (open files)

should be auto closed upon exit ↑

Can these be freed by the exiting process?



① let another process (parent or next process) clean up its memory

② switch to a kernel only VAS first, then the exiting process can free its old VAS

can't be freed by the exiting process
bc it needs access to kernel code & kstack

Wait / Waitpid

(lets the process pick which child to wait for)

→ allows the calling process (parent) to wait until a child exits

→ kernel must track parent / child relationship (xk tracks this with a parent pointer)

→ a blocking system call

→ blocks until a child calls exit (or terminated due to exception)

sleep & wakeup APIs in xk.

→ parent can clean up child's memory once child exits

→ but parent doesn't have to call wait...

→ can hand its children to the init process, init will then clean up their memory

Scheduling

- policy for deciding who runs on the CPU next
  - schedules processes & threads
  - task based evaluation
- metrics
  - latency (turnaround time)
    - user perceived time for a task (starts from arrival, includes wait time)
  - throughput
    - rate of task completion
  - fairness & starvation
    - similar time on CPU, similar time waiting
    - does the policy cause any task to wait forever
  - Scheduling overhead
    - cost of doing scheduling (policy runtime & context switch times)

Scheduling Policies

→ **FIFO**

- → run each task to completion (or till it blocks) in FIFO Order ☆ NO preemption
- → no starvation, low scheduling overhead (minimal context switches)
- → latency & throughput highly dependent on arrival order

A = 10ms B = 20ms C = 100ms

[A, B, C] A's latency = 10ms, B's latency = 30ms, C's latency = 130ms — 170ms total latency

[C, B, A] C's latency = 100ms, B's latency = 120ms, A's latency = 130ms — 350ms total latency

→ **Preemptive Shortest Job First (PSJF)**

- → schedule task needing the shortest time on CPU
  - → if a new task (or unblocked task) arrives w/ a shorter CPU time, preempts the current task & runs the new task
- → minimize average latency, avoid having shorter task waiting behind longer task
- → more context switches compared to FIFO

[C, B, A] B preempts C, A preempts B and runs first

- → leads to starvation of longer task

→ Round Robin

→ FIFO w/ time quantum (10ms - 100ms)

→ preempt once time slice expires

[C, B, A]   10c | 10b | 10a | 10c | 10b | 10c ....   total latency 210ms

10ms time slice   A's latency = 30ms , B's latency = 50ms , C's latency = 130ms

☆ context switch time is neglible compared to time slice

→ no starvation, more predictable latency compared to FIFO

→ fair? a task that blocks before time quantum has to wait the same amt of time as tasks that use the entire time quantum