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 with precise control of what's inherited

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

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

Kernel memory
User memory
VAS

Can't be freed by the exiting process
Because 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 in FIFO Order
  - no starvation, low scheduling overhead (minimal context switches)
  - latency & throughput highly dependent on arrival order

\[ A = 10\text{ms} \quad B = 20\text{ms} \quad C = 100\text{ms} \]

\[[A, B, C] \quad A's\ latency = 10\text{ms}, \ B's\ latency = 30\text{ms}, \ C's\ latency = 130\text{ms} \]

\[[C, B, A] \quad C's\ latency = 100\text{ms}, \ B's\ latency = 120\text{ms}, \ A's\ latency = 130\text{ms} \]

\[ 170\text{ms} \quad \text{total latency} \]

\[ 350\text{ms} \quad \text{total latency} \]
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
- More context switches compared to FIFO longer task

[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] 1OC | 10b | 10a | 10c | 10b | 10c .... Total latency 20ms

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

3 context switch time is negligible 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 amount of time as tasks that use the entire time quantum