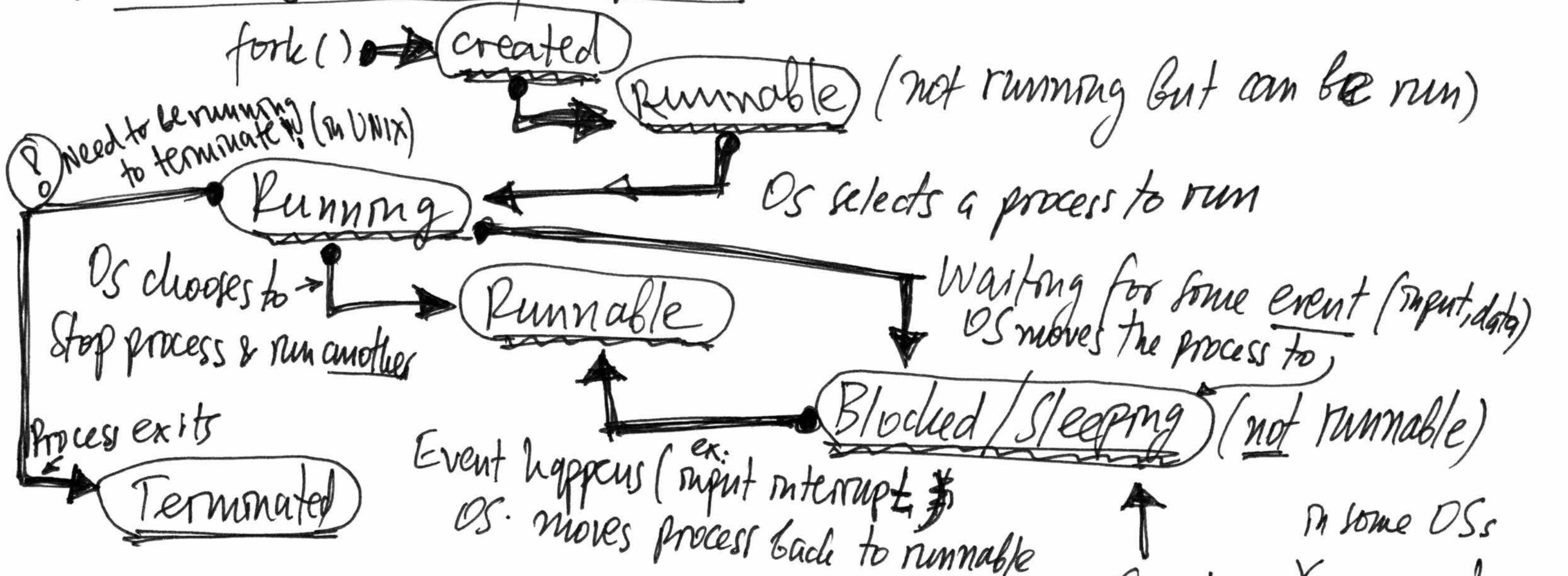
COMSW4118-7-1 9/26/2017

- **Process** = program in execution; Process Control Block (PCB), address space associated w/ process
  - mm_struct in Linux (address space)
  - task_struct in Linux (PCB) that has identifier, current running state, children, sibling.

▶ Running state of a process:



▶ In Linux, Blocked state is broken in:
- TASK_INTERRUPTIBLE
- TASK_UNINTERRUPTIBLE

Depending if task can be interrupted from sleeping or not (ex. waiting on a device is usually not inter.)

OR Sometimes (in some OSs) you may have various not runnable states:
Ex: Stopped state in Linux (CTRL-Z) — Also

▶ Terminating a process:

CTRL-C on Linux will terminate from stdin.

A process has to be running in order to terminate (In UNIX-like OSs)
- This is to facilitate cleaning up, to allow process to clean up (it's easier to delegate to the process, since processes are independent)
- If you can't run, you can't exit.

CTRL-C goes from keyboard, through the interrupt handler (part of OS), OS interprets this and converts it to a signal and sends a signal flag. then returns to interrupted proc.

▶ Signalling - type of IPC (like pipes) - the signal will set a signal flag on the recipient process (to indicate the signal, ex. kill signal).
- Some programs set signal handlers as hooks to execute for specific signals.
- Some signals are reserved and can't be handled by the program (ex. KILL).

The interrupt handler, on the way out (back to user-level code) also checks other things: ex. scheduling a new process, pending signals (including the ones that ~~were~~ may have been set from the interrupt) and processing any unhandled signals (ex. call exit() for kill).

In the OS, when an interrupt is handled (OS ~~an~~ interrupts the current process), the OS checks for not only the interrupt but also other things like signals and process them.

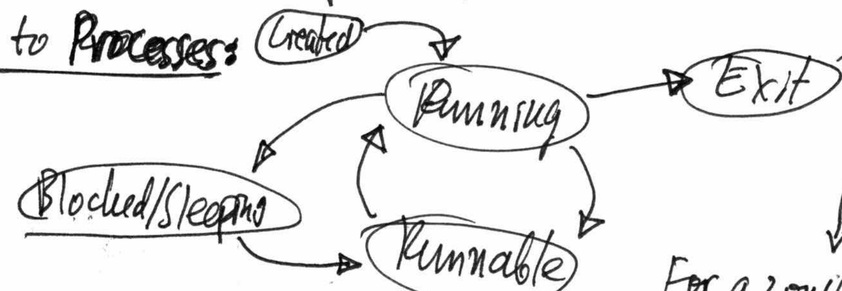
- The signal flag is part of task_struct
- If a handled signal is found, the OS will execute the ~~signals~~ signal handler, before returning from the interrupt.

▸ An interrupt is not a process (can't be scheduled, etc.) - it "borrows" the process context.

- Sending ~~kill~~ kill with "kill -9 <pid>" is similar to CTRL-C from keyboard, but:
  - ▸ kill is a system call, so it gets executed via software interrupt
  - ▸ The system call sets the flag on the target process and returns
  - ▸ Since the target process is likely not running, it will not exit till it becomes running again - just before scheduling it to run, the OS will check for any signals pending, find the KILL and execute it.

Signals are an OS inter-process-communication mechanism, they don't have anything to do w/ interrupts or hardware (although some may be delivered via those, ex. CTRL-C for kill).

- **Back to Processes:**



Process first becomes a zombie ← can ~~never~~ never be revived
In UNIX
then it becomes Dead

For a zombie, the PCB is retained (state etc.) but you don't have an address state.

- This is to allow wait() to get information about child termination from PCB.
- Also, OS can collect statistics after process termination.
- wait() finally releases zombies and discards their PCB.
- orphans (where parent finished w/o calling wait()) are reparented to <init>, which periodically calls wait() to release them.

- **Linux source examples**
  - ▸ PCB ⇒ include/linux/sched.h ⇒ task-struct
    - state = running state of process (Linux does not distinguish b/w running/runnable)
    - mm-struct = pointer to address space
    - pid, children, sibling
    - SYSCALL macro → fork → do-fork → copy-process has:
      1. "dup-task-struct (current)" ← macro to get the currently running process
         allocate new, copy from template, etc.
      2. Then copy file handles, file system, signal handlers, signals, address space, etc.

task-struct is manipulated by a lot of kernel code!

COMSW4118-7-3 9/26/2017

Threads

while processes are heavyweight and independent (no sharing by default), the threads share the state (memory / address space) and thus communication is easier.

$\underbrace{t_1\ t_2\ t_3\ t_4}_{}$ → all in same group share the same address space ← $\underbrace{t_5\ t_6\ t_7}_{}$

thread group w/ its own address space

another thread group w/ own address space

▶ Exact relation b/n threads & processes depends on the particular OS implementation

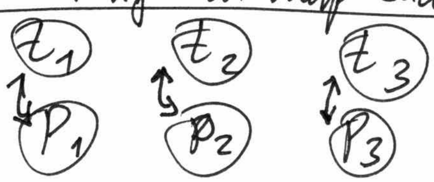
- Originally all were user threads: → User Library → $t_1\ t_2\ t_3$ (M) creates threads ↔ Kernel Process (1)

OS does not know about user threads, the user library manages creation, scheduling, etc.

But all of those map to a single process in the OS

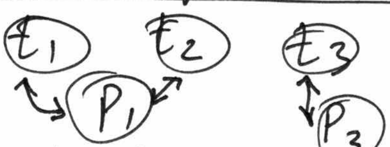
Plus is you don't have to go to OS to create threads. Bad thing is OS does not know about threads – OS decides what to run where and here it only sees one process so no threads will truly run in parallel on multiple cores, here the threads are just an illusion, the OS still schedules one process. User thread Libraries can intercept threads that are about to wait and switch to another thread – thus, getting some concurrency but only for things for which the user library can intercept – not in the case of blocking system calls (like read() for example which blocks in kernel) – only for non-blocking system calls (ex. User Lib. can check if non-blocking read has data and switch to another process if that is the case).

- A user Library can map each thread to a separate process:



then the OS knows about the threads, but the address spaces are separate and it takes effort to make it look like they are shared.

- A user Library can also map many-to-many:



But that probably has problems from both $n:1$ and $1:1$.

- The OS itself can support threads (kernel threads) and map them to user threads – thus, sharing the address space is trivial, but the OS is more complex since it has to support threads.

Linux uses task_struct to support both a process and a thread, the difference only in what is shared w/ other task_structs (ex. mm_struct for threads)

COMSW4118-7-4 9/26/2017

For example, see **kernel/fork.c → copy-mm**
it copies mm-struct when one process forks another, but just sets it to the same reference for threads.

- A <u>process</u> is a <u>thread group</u> with only one thread (which is the <u>thread group leader</u> for the group).
- <u>clone()</u> is the sys call on Linux to create threads (<u>fork()</u> calls <u>clone()</u> w/ flags to copy everything for creating a new process).