NOTICE

- Deadlines
  - (Passed) ~~Programming assignment 1~~
  - (4/20 11:59 PM) Midterm quiz 1
  - (4/27 11:59 PM) Programming assignment 2

- Others
  - TAs and the instructor is currently grading the PA 1 submissions

CS 344: Operating Systems I

Part I – Scheduling

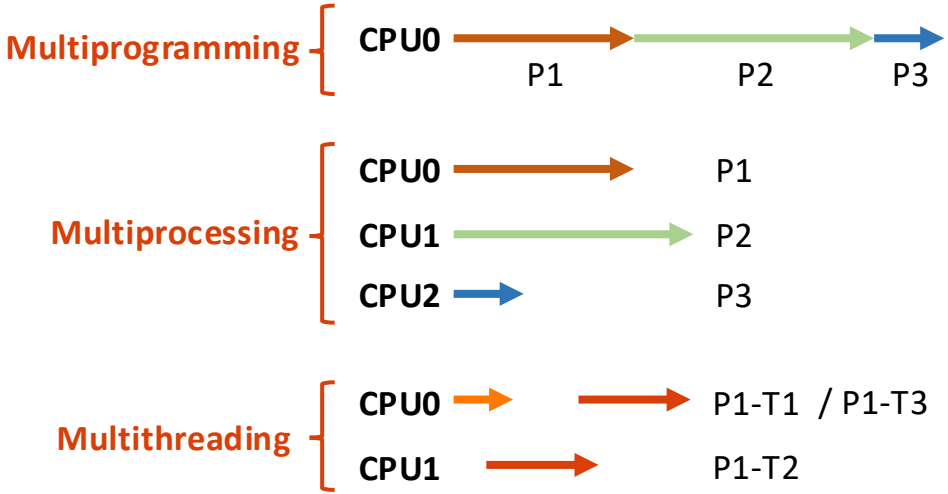
M/W 12:00 – 1:50 PM (LINC #200)

Sanghyun Hong

sanghyun.hong@oregonstate.edu

REVISIT: TERMINOLOGY

- **Definitions:**
  - Multiprogramming vs. multi-processing vs. multi-threading
    - Multi-programming: multiple jobs (or processes)
    - Multi-processing: multiple processors (CPUs)
    - Multi-threading: multiple threads

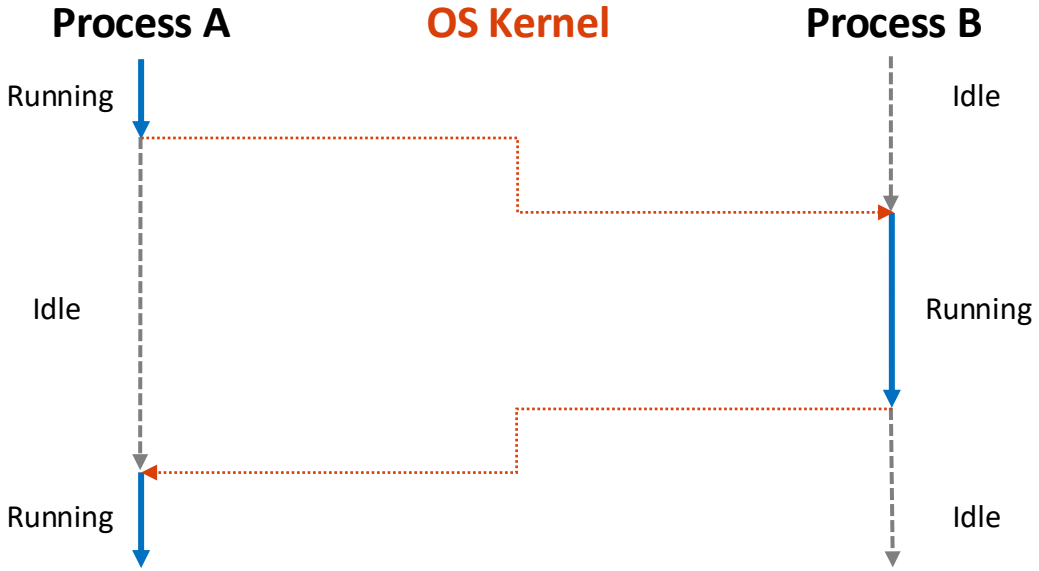


REVISIT: CONTEXT SWITCHING

- **Context switch**
  - **Definition:** OS stores the current process's status and loads the new process's one
  - **Informal:** OS takes a CPU from one process and gives it to another

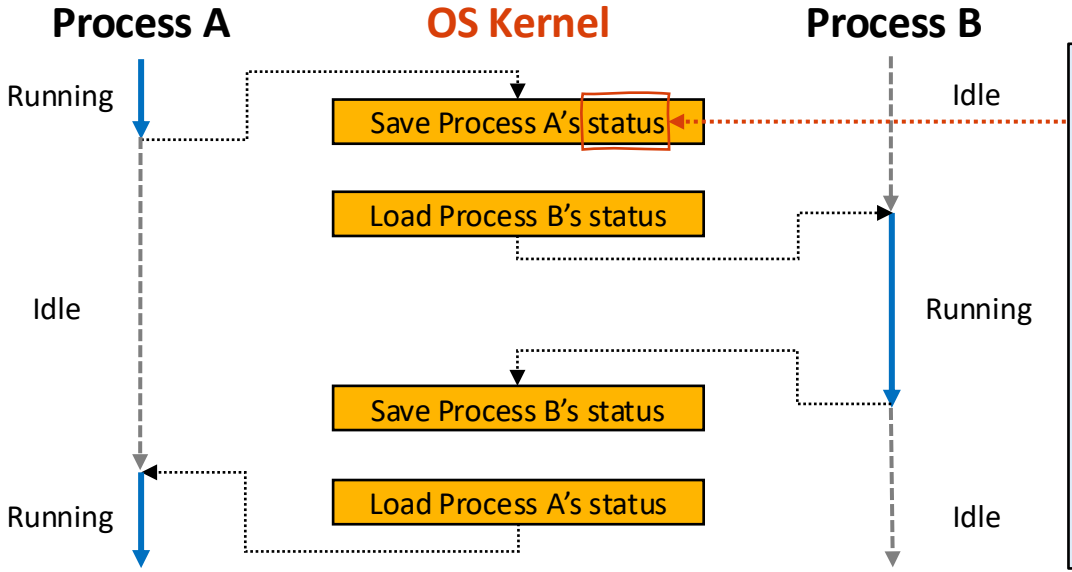
REVISIT: CONTEXT SWITCHING

- **Context switch**
  - **Definition:** OS stores the current process's status and loads the new process's one
  - **Informal:** OS takes a CPU from one process and gives it to another



REVISIT: CONTEXT SWITCHING

- **Context switch**
  - **Definition:** OS stores the current process's status and loads the new process's one
  - **Informal:** OS takes a CPU from one process and gives it to another



Recall: Process control block

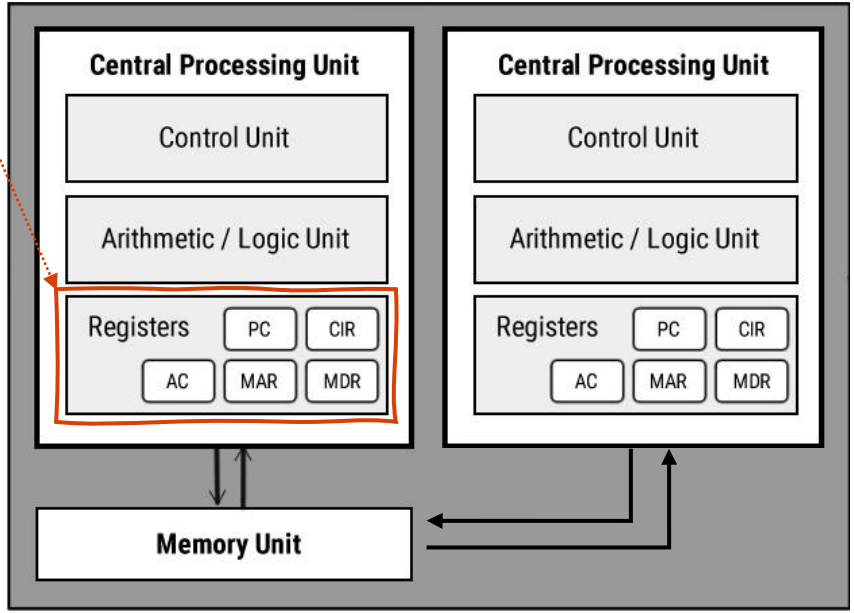
A structure in OS that contains a set of information required to run a process on a CPU. Recall that Linux has *task_struct*.

- CPU#
- Program counter
- Instruction pointer
- Heap/stack pointer
- Process state [!]
- …

REVISIT: CONTEXT SWITCHING

- (Linux) has the process context
  - **Code**
    - Program counter
    - Instruction pointer
  - **Stack and heap**
    - Stack pointer
    - Heap pointer
  - **Running context**
    - Process state (ID, …)
    - Execution flags
    - CPU # to run
    - (OS II) Scheduling policy
    - (OS II) Mem. virtualization
  - …

Process Context: A set of information that OS requires to run a process on a CPU, different from CPU vendors (ex. In Linux, it's defined as *task_struct*, Link)



REVISIT: CONTEXT SWITCHING

- (Linux) has the *process context*
  - **Code**
    - Program counter
    - Instruction pointer
  - **Stack and heap**
    - Stack pointer
    - Heap pointer
  - **Running context**
    - Process state (ID, …)
    - Execution flags
    - CPU # to run
    - (OS II) Scheduling policy
    - (OS II) Mem. virtualization
  - …

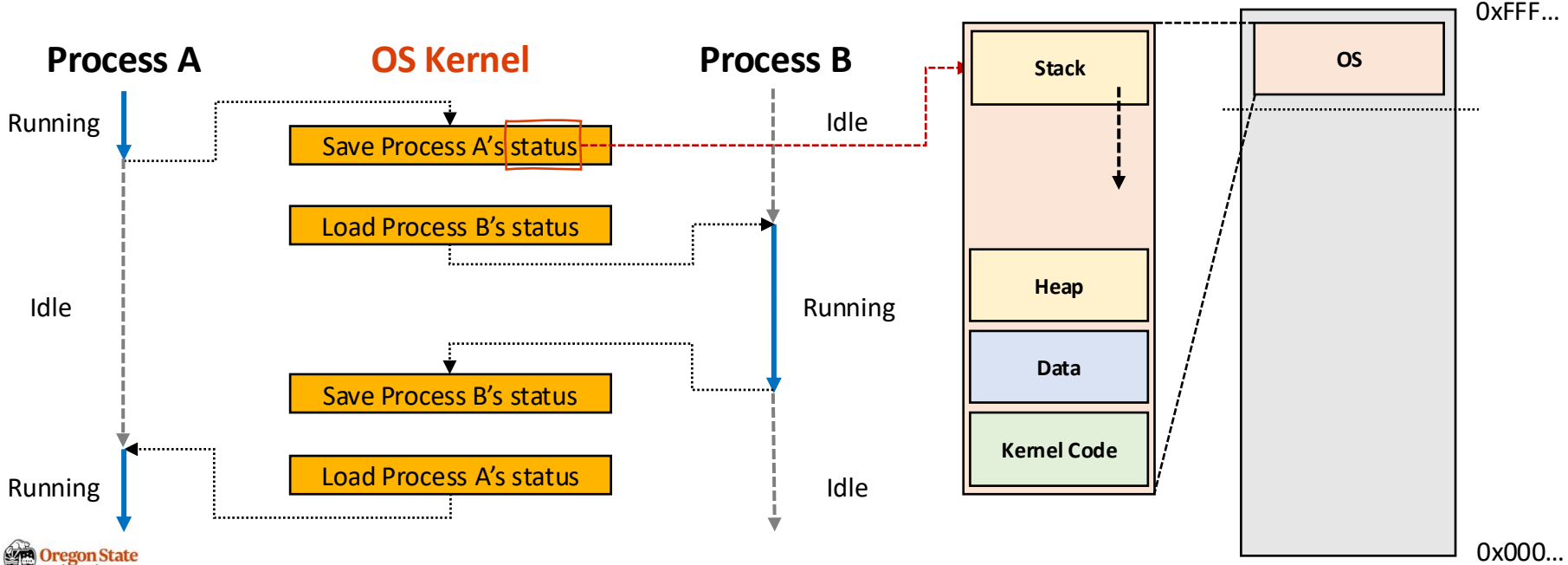
Process Context: A set of information that OS requires to run a process on a CPU, different from CPU vendors (ex. In Linux, it's defined as *task_struct*, Link)

```
728 struct task_struct {
729 #ifdef CONFIG_THREAD_INFO_IN_TASK
730         /*
731          * For reasons of header soup (see current_thread_info()), this
732          * must be the first element of task_struct.
733          */
734         struct thread_info              thread_info;
735 #endif
736         unsigned int                    __state;
737
738 #ifdef CONFIG_PREEMPT_RT
739         /* saved state for "spinlock sleepers" */
740         unsigned int                    saved_state;
741 #endif
742
743         /*
744          * This begins the randomizable portion of task_struct. Only
745          * scheduling-critical items should be added above here.
746          */
747         randomized_struct_fields_start
748
749         void                            *stack;
750         refcount_t                      usage;
751         /* Per task flags (PF_*), defined further below: */
752         unsigned int                    flags;
753         unsigned int                    ptrace;
```

```
852         struct sched_info               sched_info;
853
854         struct list_head                tasks;
855 #ifdef CONFIG_SMP
856         struct plist_node               pushable_tasks;
857         struct rb_node                  pushable_dl_tasks;
858 #endif
859
860         struct mm_struct                *mm;
861         struct mm_struct                *active_mm;
862
863         /* Per-thread vma caching: */
864         struct vmacache                 vmacache;
865
866 #ifdef SPLIT_RSS_COUNTING
867         struct task_rss_stat            rss_stat;
868 #endif
869         int                             exit_state;
870         int                             exit_code;
871         int                             exit_signal;
872         /* The signal sent when the parent dies: */
873         int                             pdeath_signal;
874         /* JOBCTL_*, siglock protected: */
875         unsigned long                   jobctl;
876
877         /* Used for emulating ABI behavior of previous Linux versions: */
878         unsigned int                    personality;
```

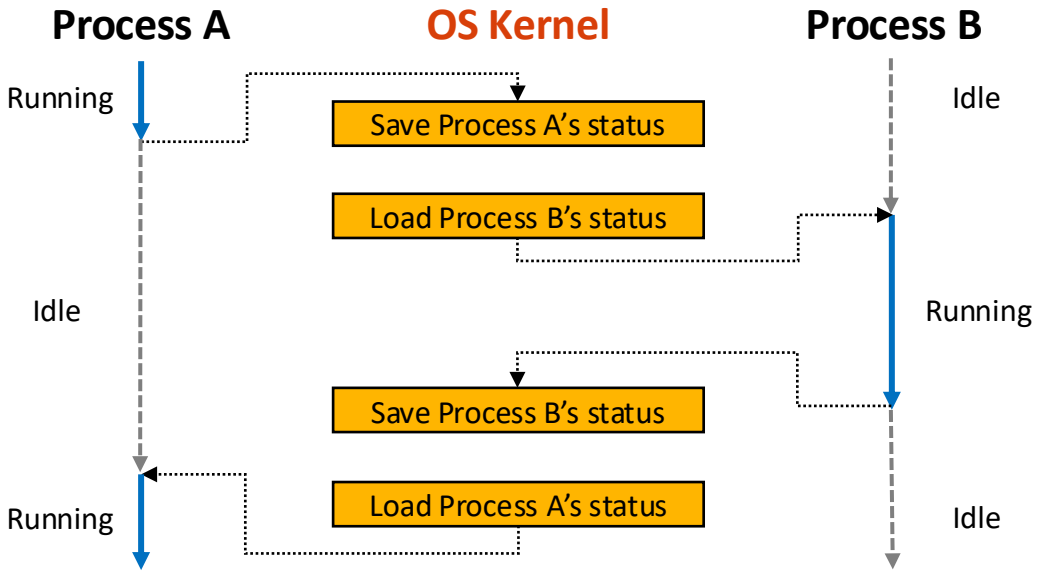
REVISIT: CONTEXT SWITCHING

- **OS kernel**
  - **Definition:** a program that has a complete control over the system
  - OS kernel stores the context in its own *stack (not in the user stack)*



REVISIT: CONTEXT SWITCHING

- **Context switch**
  - **Definition:** OS stores the current process's status and loads the new process's one
  - **Informal:** OS takes a CPU from one process and gives it to another



REVISIT: CONTEXT SWITCHING

- **Context switch**
  - **Definition:** OS stores the current process's status and loads the new process's one
  - **Informal:** OS takes a CPU from one process and gives it to another

- No free lunch
  - Context switching takes $\sim$5 μs on average
  - OS typically runs 100+ processes
  - Too many context switching makes a system unable to respond…

TOPICS FOR TODAY

- Part I: Scheduling
  - Provide abstraction
    - What is scheduling?
    - What does OS achieve by scheduling?
  - Manage resources
    - What happens during scheduling?
    - How does OS perform scheduling?
    - How does OS implement this scheduling?
  - Offer standard libraries
    - What interfaces does the OS provide for scheduling?
    - (Note) We will talk about this more in the “synchronization” lecture

OS SCHEDULING

- Scheduling
  - OS mechanism that decides which process gets the CPU next
  - OS creates the illusion that multiple processes run at the same time
  - The *scheduler* is the kernel component responsible for this decision

```
while True:
    pause()                 # wait for a timer interrupt
    p = pick_next()         # select the next process to run
    context_switch(p)       # save current state, restore p's state
```

  - The scheduler process
    - CPU idles until the hardware timer fires (ex. ~1ms)
    - OS wakes up and picks the next process

OS SCHEDULING

- Scheduling
  - OS mechanism that decides which process gets the CPU next
  - OS creates the illusion that multiple processes run at the same time
  - The *scheduler* is the kernel component responsible for this decision
  - The scheduler process
    - CPU idles until the hardware timer fires (ex. ~1ms)
    - OS wakes up and picks the next process
  - Two scheduling methods
    - Non-preemptive: the process voluntarily gives up the CPU (Windows 3.1)

```
# Cooperative — process decides when to yield
while True:
    run(current)              # runs until the process calls yield() itself
    p = pick_next()
    context_switch(p)
```

OS SCHEDULING

- Scheduling
  - OS mechanism that decides which process gets the CPU next
  - OS creates the illusion that multiple processes run at the same time
  - The *scheduler* is the kernel component responsible for this decision
  - The scheduler process
    - CPU idles until the hardware timer fires (ex. ~1ms)
    - OS wakes up and picks the next process
  - Two scheduling methods
    - Non-preemptive: the process voluntarily gives up the CPU (Windows 3.1)
    - Preemptive : the OS forcibly takes the CPU (Linux)

```
# Cooperative — process decides when to yield
while True:
    run(current)              # runs until the process calls yield() itself
    p = pick_next()
    context_switch(p)
```

```
# Preemptive — OS decides when to stop a process
while True:
    set_timer(1ms)            # hardware timer will fire after 1ms
    run(current)              # process runs...
                              # ...timer fires → CPU runs OS interrupt handler
    p = pick_next()           # OS takes control and picks next process
    context_switch(p)
```

OS SCHEDULING

- I/O bound vs. compute-bound
  - **I/O-bound** **:** a process spends the most time waiting for I/O (e.g., terminal, shell)
  - **Compute-bound:** a process spends the most time computing (e.g., ML training)
  - An example scheduler behavior:
    - Shell and a process for ML training have the same scheduling *priority*
    - Shell spends most of its time blocked – the total time it runs barely grows
    - ML spends most of its time for GEMM – the total time it runs grows a lot
    - On a keypress, the scheduler sees that the shell has run much less
    - The scheduler immediately preempts the ML training process
    - Result:
      - Shell responds immediately upon a keypress
      - ML training process gets more CPU time than its share

TOPICS FOR TODAY

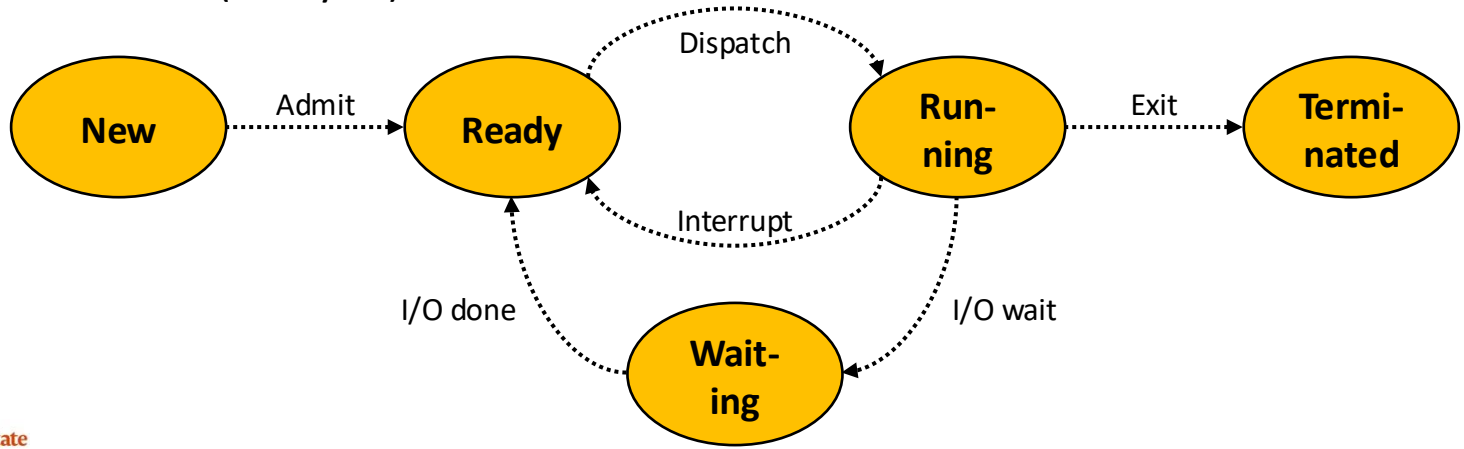
- Part I: Scheduling
  - Provide abstraction
    - What is scheduling?
    - What does OS achieve by scheduling?
  - Manage resources
    - What happens during scheduling?
    - How does OS perform scheduling?
    - How does OS implement this scheduling?
  - Offer standard libraries
    - What interfaces does the OS provide for scheduling?
    - (Note) We will talk about this more in the “synchronization” lecture

WHAT HAPPENS DURING SCHEDULING?

- **Process states:**
  - **New:** a process (or thread) is being created (by fork())
  - **Ready:** the process is waiting to run
  - **Running:** the process is running on a CPU(or CPUs)
  - **Waiting:** the process is waiting for some events to occur (*e.g.*, a data loaded from storage)
  - **Terminated:** the process has finished execution; waiting for removal

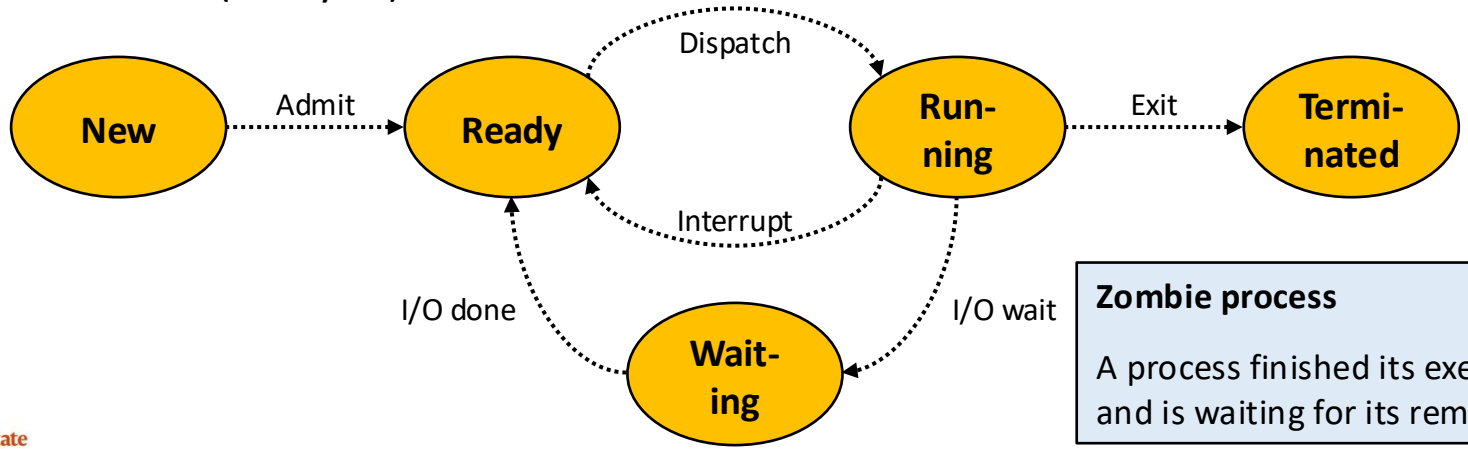
WHAT HAPPENS DURING SCHEDULING?

- A process can have **five states**:
  - **New:** a process (or thread) is being created (by fork())
  - **Ready:** the process is waiting to run
  - **Running:** the process is running on a CPU(or CPUs)
  - **Waiting:** the process is waiting for some events to occur (*e.g.*, a data loaded from storage)
  - **Terminated:** the process has finished execution; waiting for removal
- State transition (life cycle):



WHAT HAPPENS DURING SCHEDULING?

- A process can have **five states**:
  - **New:** a process (or thread) is being created (by fork())
  - **Ready:** the process is waiting to run
  - **Running:** the process is running on a CPU(or CPUs)
  - **Waiting:** the process is waiting for some events to occur (*e.g.*, a data loaded from storage)
  - **Terminated:** the process has finished execution; waiting for removal
- State transition (life cycle):



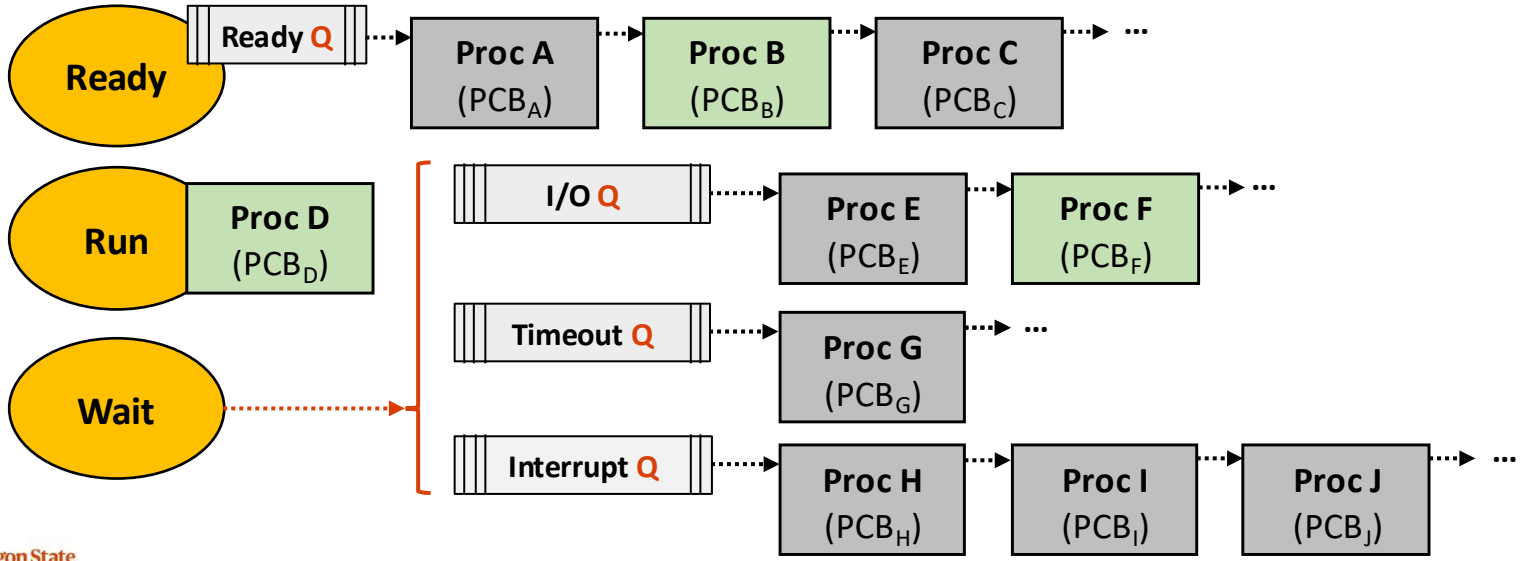
Zombie process

A process finished its execution and is waiting for its removal

HOW DOES OS PERFORMS SCHEDULING?

- **Scheduling**
  - **Definition:** an OS activity that schedules processes in different states
  - **Note:** OS implements queues to hold multiple processes in the same state

- **Illustration (single CPU)**

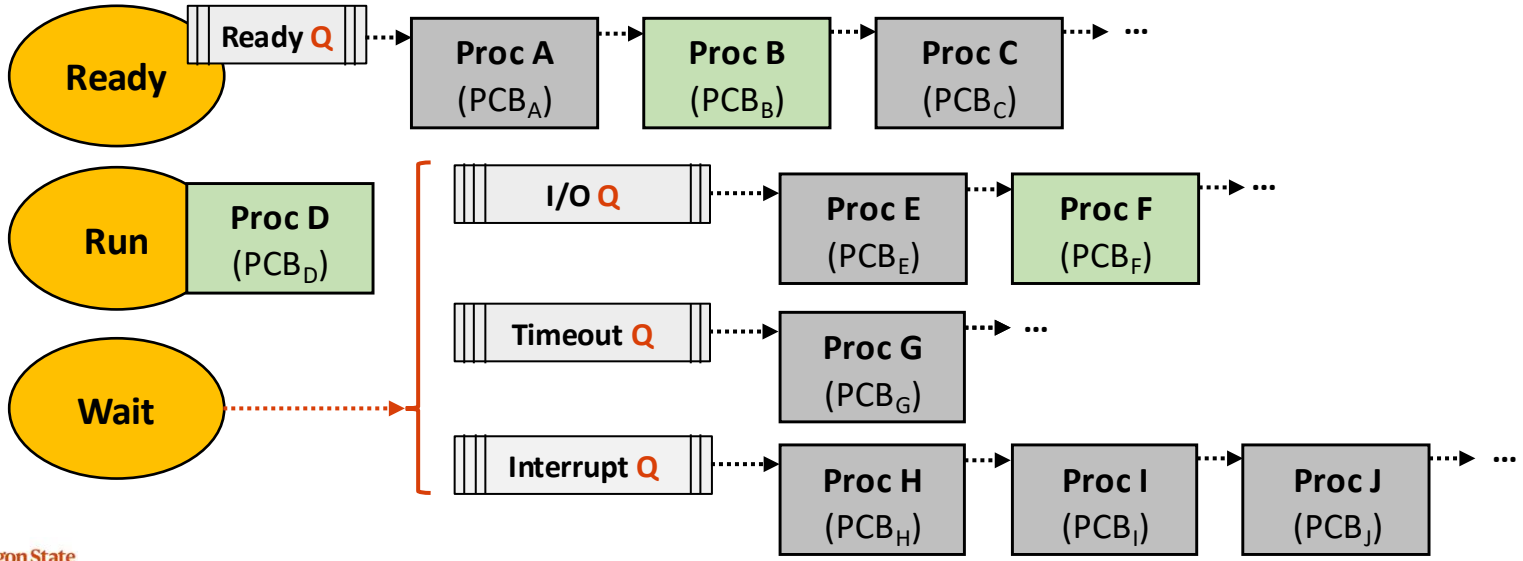


How does OS performs scheduling?

- **Scheduling**
  - **Definition:** an OS activity that schedules processes in different st
  - **Note:** OS implements queues to hold multiple processes in the sa
- **Illustration (single CPU)**

Illustrated Example

1. Kicks out Proc D (timeout)
2. Runs Proc B
3. Puts Proc F in the ready Q (I/O has done, in this case)



SCHEDULING EXAMPLE: HIGH-LEVEL VIEW

- 3 Processes in Chrome:
  - **P1:** Download movies
  - **P2:** Open Canvas
  - **P3:** Search StackOverflow
- Example
  - **New** :
  - **Ready**:
  - **Run** :
  - **Wait** :
  - **Term..**:

  - **Scenario:** open a website for downloading movies

SCHEDULING EXAMPLE: HIGH-LEVEL VIEW

- 3 Processes in Chrome:
  - **P1:** Download movies
  - **P2:** Open Canvas
  - **P3:** Search StackOverflow
- Example
  - **New** :
  - **Ready**:
  - **Run** :
  - **Wait** :
  - **Term..**:

  - **Scenario:** the website opened and open two other websites

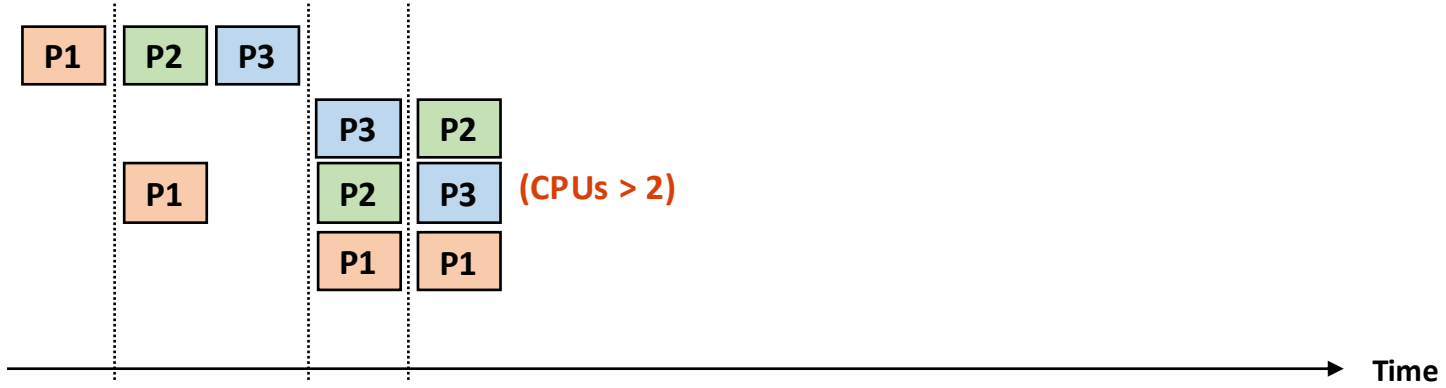
SCHEDULING EXAMPLE: HIGH-LEVEL VIEW

- 3 Processes in Chrome:
  - **P1:** Download movies
  - **P2:** Open Canvas
  - **P3:** Search StackOverflow
- Example
  - **New** :
  - **Ready**:
  - **Run** :
  - **Wait** :
  - **Term.**.:

  - **Scenario:** downloads started, and you focus on Canvas

SCHEDULING EXAMPLE: HIGH-LEVEL VIEW

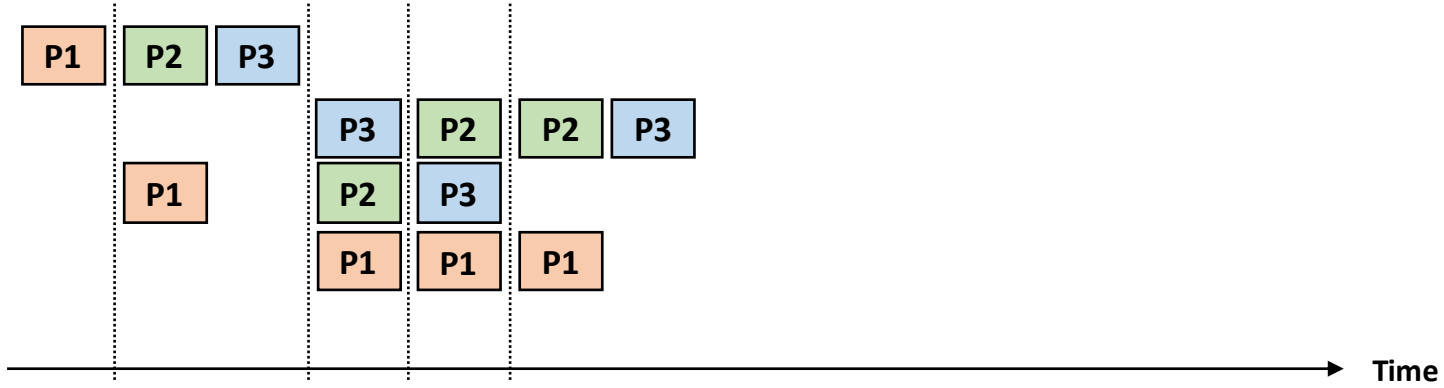
- 3 Processes in Chrome:
  - **P1:** Download movies
  - **P2:** Open Canvas
  - **P3:** Search StackOverflow
- Example
  - **New** :
  - **Ready**:
  - **Run** :
  - **Wait** :
  - **Term..**:



  - **Scenario:** while downloading, you start searching StackOverflow

SCHEDULING EXAMPLE: HIGH-LEVEL VIEW

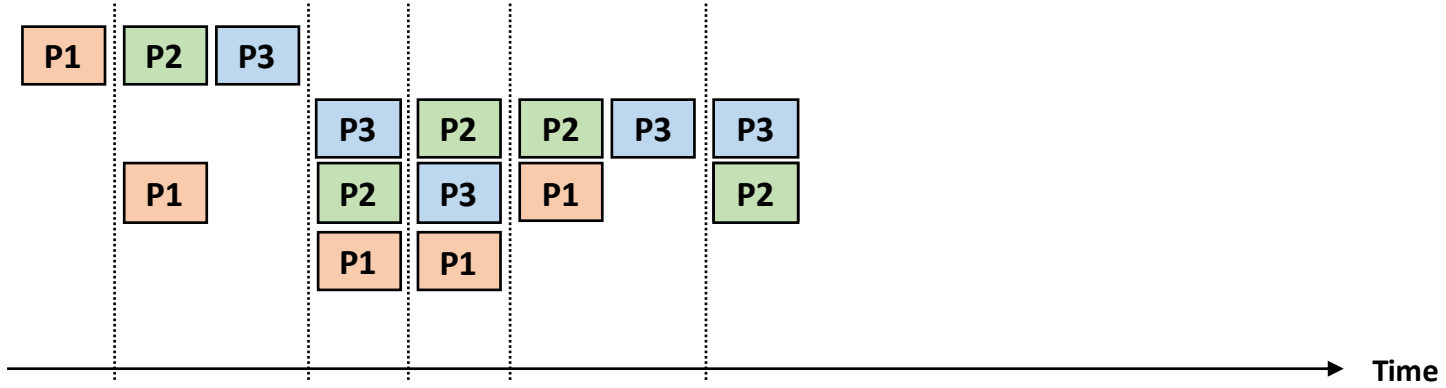
- 3 Processes in Chrome:
  - **P1:** Download movies
  - **P2:** Open Canvas
  - **P3:** Search StackOverflow
- Example
  - **New** :
  - **Ready**:
  - **Run** :
  - **Wait** :
  - **Term..**:



  - **Scenario:** downloading movies are done

SCHEDULING EXAMPLE: HIGH-LEVEL VIEW

- 3 Processes in Chrome:
  - **P1:** Download movies
  - **P2:** Open Canvas
  - **P3:** Search StackOverflow
- Example
  - **New** :
  - **Ready**:
  - **Run** :
  - **Wait** :
  - **Term..**:



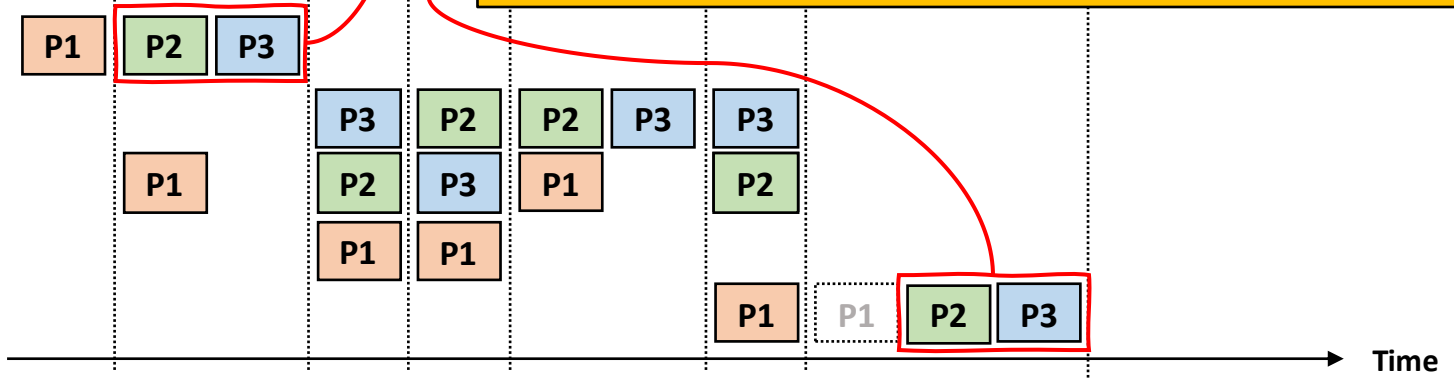
  - **Scenario:** close the download tab, and keep looking at Canvas

SCHEDULING EXAMPLE: HIGH-LEVEL VIEW

- 3 Processes in Chrome:
  - **P1:** Download movies
  - **P2:** Open Canvas
  - **P3:** Search StackOverflow
- Example
  - **New** :
  - **Ready**:
  - **Run** :
  - **Wait** :
  - **Term..**:
  - **Scenario:** close the other two tabs to go to bed

Linked list: OS keeps a list of processes for each state by using the queue structure. Based on a policy, OS searches the queue and find the process with the highest priority

Zombie process(es): A list of processes whose job has already terminated (exited) but has not removed from the process list. One case is that a parent process wait for a child's exit status.



OS SCHEDULING ALGORITHMS

- **First Come, First Served (FCFS):**
  - **Pro :** easy to implement
  - **Con:** short processes could wait behind a single long process(es) – Convoy Effect

- **Shortest Job First (SJF):**
  - **Pro :** minimize the wait time (on average)
  - **Con:** needs knowledge of future burst time – only theoretically feasible

- **Round Robin (RR):**
  - **Pro :** able to run each process for one time quantum, then rotates
  - **Con:** OS behavior depends on how to set the time quantum
    - Large: it is the same as FCFS
    - Small: OS spends the most of its time for context-switching

OS SCHEDULING ALGORITHMS

- **Round Robin (RR):**
  - **Pro :** able to run each process for one time quantum, then rotates
  - **Con:** OS behavior depends on how to set the time quantum
    - Large: it is the same as FCFS
    - Small: OS spends the most of its time for context-switching

```
# Round Robin simulation in Python
queue = [P1(burst=10),
         P2(burst=6),
         P3(burst=4)]              # you can add more

quantum = 2

while queue:
   p = queue.pop(0)
   run(p, min(quantum, p.remaining))
   if p.remaining > 0:
      queue.append(p)              # still has work → go to the back
```

OS SCHEDULING ALGORITHMS

- **Priority Scheduling:**
  - **Pro :** able to run processes based on its priority
  - **Con:** lower priority processes may never run – Starving
  - **Solution:** gradually raise priority the longer a process waits – Aging

- Nice value and real-time priority:
  - **Nice value:**
    - Used by *normal processes* (high -20 – 19 low | default 0)
    - Set to -20: schedule more often, but gets smaller share of CPU
    - Note that this is not the absolute time
  - **Real-time priority:**
    - Set to a *critical process* (low 0 – 99 high)
    - Runs always before any normal process
    - Occupies a completely separate priority space from nice values

OS SCHEDULING ALGORITHMS

- **Priority Scheduling:**
  - **Pro :** able to run processes based on its pric
  - **Con:** lower priority processes may never run -
  - **Solution:** gradually raise priority the longer a p

- Nice value and real-time priority:
  - **Nice value:**
    - *Normal processes* (high -20 – 19 low | def
    - Set to -20: schedule more often, but gets
    - Note that this is not the absolute time
  - **Real-time priority:**
    - *Critical process* (low 0 – 99 high)
    - Runs always before any normal process
    - Occupies a completely separate priority s

```
long burn(int seconds) {
   long count = 0;
   time_t start = time(NULL);
   while (time(NULL) - start < seconds)
      count++;
   return count;
}

int main() {
   pid_t pid = fork();

   if (pid == 0) {
      // child — low priority (nice +19, no root needed)
      setpriority(PRIO_PROCESS, 0, 19);
      long n = burn(3);
      printf("[child  | nice +19] iterations: %ld\n", n);
   } else {
      // parent — default priority (nice 0)
      long n = burn(3);
      printf("[parent | nice   0] iterations: %ld\n", n);
      wait(NULL);
   }
   return 0;
}
```

OS SCHEDULING ALGORITHMS

- **Priority Scheduling:**
  - **Pro :** able to run processes based on its pric
  - **Con:** lower priority processes may never run -
  - **Solution:** gradually raise priority the longer a p

- Nice value and real-time priority:
  - **Nice value:**
    - *Normal processes* (high -20 – 19 low | def
    - Set to -20: schedule more often, but gets
    - Note that this is not the absolute time
  - **Real-time priority (This requires root – DIY):**
    - *Critical process* (low 0 – 99 high)
    - Runs always before any normal process
    - Occupies a completely separate priority s

```
long burn(int seconds) {
  long count = 0;
  time_t start = time(NULL);
  while (time(NULL) - start < seconds)
    count++;
  return count;
}

int main() {
  pid_t pid = fork();

  if (pid == 0) {
    // child — normal process (SCHED_NORMAL, default 0)
    setpriority(PRIO_PROCESS, 0, 19);
    long n = burn(3);
    printf("[child  | SCHED_NORMAL] iterations: %ld\n", n);
  } else {
    // parent — real-time process (SCHED_FIFO, priority 50)
    struct sched_param param = { .sched_priority = 50 };
    sched_setscheduler(0, SCHED_FIFO, &param);
    long n = burn(3);
    printf("[parent | SCHED_FIFO    ] iterations: %ld\n", n);
    wait(NULL);
  }
  return 0;
}
```

OS SCHEDULING ALGORITHMS

- Problems in traditional UNIX scheduling
  - **Nice to timeslice mapping:** OS behavior depends on the chosen default nice value

    nice 0 → 100ms? 50ms? 10ms? — there is no obviously correct answer
    nice +19 → 5ms? 1ms? — whatever you pick causes edge cases

    - If default is too long, then poor responsiveness
    - If default is too short, too many ctx
    - The *right value* depends on the system load

OS SCHEDULING ALGORITHMS

- Problems in traditional UNIX scheduling
  - **Nice to timeslice mapping:** OS behavior depends on the chosen default nice value
  - **Non-linear nice effect:** a step from nice 0-1 has a different impact than nice 18-19

    nice 0 → 100ms
    nice +1 → 95ms (difference: 5ms, ~5% change)
    nice +18 → 10ms
    nice +19 → 5ms (difference: 5ms, ~50% change)

    - The same +1 step has a completely different impact depending on the starting value
    - Nice values unpredictable — users and developers cannot reason about them intuitively

OS SCHEDULING ALGORITHMS

- Problems in traditional UNIX scheduling
  - **Nice to timeslice mapping:** OS behavior depends on the chosen default nice value
  - **Non-linear nice effect:** a step from nice 0-1 has a different impact than nice 18-19
  - **Timer-tick alignment:** absolute timelices must be multiple of the tick

    HZ = 100 → tick = 10ms → minimum timeslice = 10ms
    HZ = 250 → tick = 4ms → minimum timeslice = 4ms
    HZ = 1000 → tick = 1ms → minimum timeslice = 1ms

    - The same nice value produces different actual timeslices on systems with different HZ
    - Behavior changes depending on the kernel configurations – not portable

  - **Unfair wakeup bonus:** a waken process can receive a disproportionate share of CPU time

OS SCHEDULING ALGORITHMS

- Problems in traditional UNIX scheduling
  - **Nice to timeslice mapping:** OS behavior depends on the chosen default nice value
  - **Non-linear nice effect:** a step from nice 0-1 has a different impact than nice 18-19
  - **Timer-tick alignment:** absolute timelices must be multiple of the tick
  - **Unfair wakeup bonus:** a waken process can receive a disproportionate share of CPU time

    Scenario:
    P1 (ML training) — running, has used most of its timeslice
    P2 (text editor) — just woke up from waiting for a keypress

    Traditional scheduler gives P2 a wakeup bonus →
    P2 runs immediately (good)

    But if P2 keeps waking and sleeping rapidly:
    P2 accumulates bonus after bonus →
    P2 ends up with far more CPU than its fair share (bad)

OS SCHEDULING ALGORITHMS

- Completely fair scheduler (CFS)
  - Run the processes that has consumed the least CPU so far
    - Always pick the process with the smallest *vruntime*
  - **vruntime:** actual runtime weighted by the process's nice value
    - vruntime = actual_runtime × (NICE_0_LOAD / process_weight)
    - Low nice → large weight → vruntime grows slowly → chosen more often
    - High nice → small weight → vruntime grows quickly → chosen less often

```
# process 1 — nice 0
nice -n 0 yes > /dev/null &
PID1=$!

# process 1 — nice +19
nice -n 19 yes > /dev/null &
PID2=$!
```

```
# vruntime runtime comparison
watch -n 0.5 "echo '[nice 0 ]' && cat /proc/$PID1/sched | grep vruntime
              echo '[nice+19]' && cat /proc/$PID2/sched | grep vruntime"

# terminate all
kill $PID1 $PID2
```

TOPICS FOR TODAY

- Part I: Scheduling
  - Provide abstraction
    - What is scheduling?
    - What does OS achieve by scheduling?
  - Manage resources
    - What happens during scheduling?
    - How does OS perform scheduling?
    - How does OS implement this scheduling?
  - Offer standard libraries
    - What interfaces does the OS provide for scheduling?
    - (Note) We will talk about this more in the “synchronization” lecture

REAL-TIME (RT) SCHEDULING

- SCHED_FIFO vs. SCHED_RR
  - RT processes *always* run before any normal process
  - RT priorities are *static* – a lower-priority RT processes cannot preempt a higher one
  - Linux also offers *soft* RT: it tries to meet deadlines but does not guarantee them

- SCHED_FIFO
  - P1 (SCHED_FIFO, priority 50) — running
    P2 (SCHED_NORMAL) — waiting

    → P2 never runs until P1 voluntarily yields or blocks
    → if P1 enters an infinite loop, P2 starves forever

REAL-TIME (RT) SCHEDULING

- SCHED_FIFO
  - P1 (SCHED_FIFO, priority 50) — running
    P2 (SCHED_NORMAL) — waiting

    → P2 never runs until P1 voluntarily yields or blocks
    → if P1 enters an infinite loop, P2 starves forever

- SCHED_RR
  - P1 (SCHED_RR, priority 50) — running
    P2 (SCHED_RR, priority 50) — waiting
    P3 (SCHED_NORMAL) — waiting

    → P1 timeslice expires → P2 runs → P1 runs → ... (round-robin between P1 and P2)
    → P3 still cannot run until both P1 and P2 block

REAL-TIME (RT) SCHEDULING

- Static priority
  - P1 (SCHED_FIFO, priority 80) — blocked (waiting for I/O)
    P2 (SCHED_FIFO, priority 50) — running

    → P1 I/O completes → immediately preempts P2
    → priority 80 always beats 50, no dynamic adjustment

OS SCHEDULING INTERFACE

- System calls
  - Query
    - sched_getscheduler() — what policy am I running under?
    - sched_getparam() — what is my RT priority?
  - Control
    - nice() / setpriority() — adjust CPU share (no root needed)
    - sched_setscheduler() — change policy — FIFO, RR (root needed)
    - sched_setaffinity() — pin to specific CPU cores (root needed)
  - Yield
    - sched_yield() — voluntarily give up the CPU

TOPICS FOR TODAY

- Part I: Scheduling
  - Provide abstraction
    - What is scheduling?
    - What does OS achieve by scheduling?
  - Manage resources
    - What happens during scheduling?
    - How does OS perform scheduling?
    - How does OS implement this scheduling?
  - Offer standard libraries
    - What interfaces does the OS provide for scheduling?
    - (Note) We will talk about this more in the “synchronization” lecture

Thank You!

M/W 12:00 – 1:50 PM (LINC #200)

Sanghyun Hong

sanghyun.hong@oregonstate.edu