CS330: Operating Systems

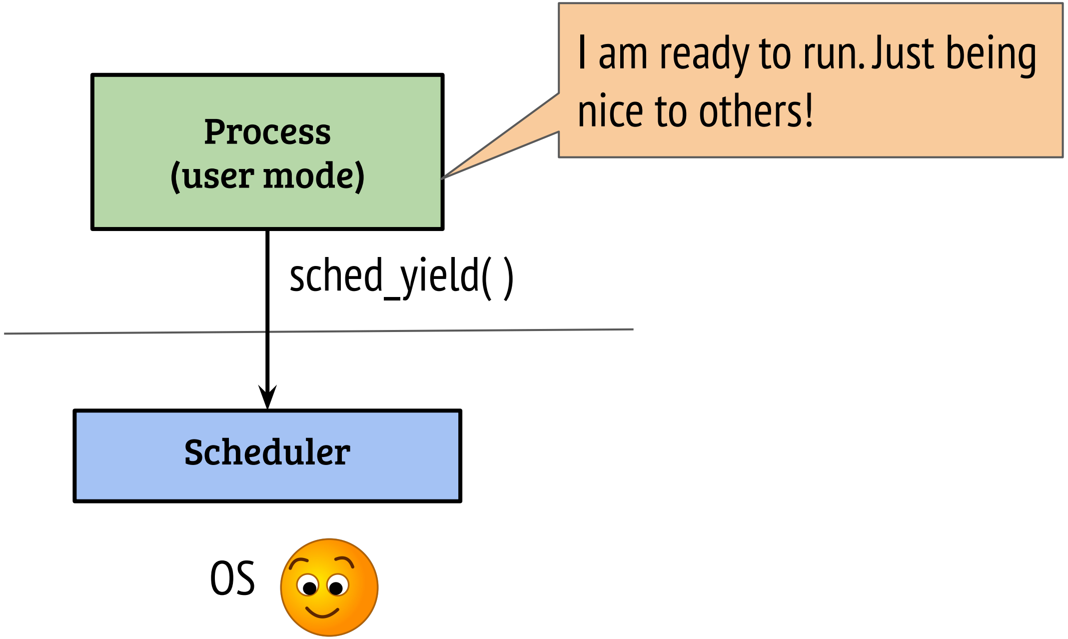
Process scheduling

Recap: OS execution, user-OS mode switch

- Which stack is used by the OS for kernel-mode execution?
- The hardware switches the SP to point it to a pre-configured per-process OS stack on mode switch
- How the user process state preserved and restored?
- The user execution state is saved/restored using the kernel stack by the hardware (and OS)
- Which address space the OS uses?
- A part of the process address space is reserved for OS and is protected

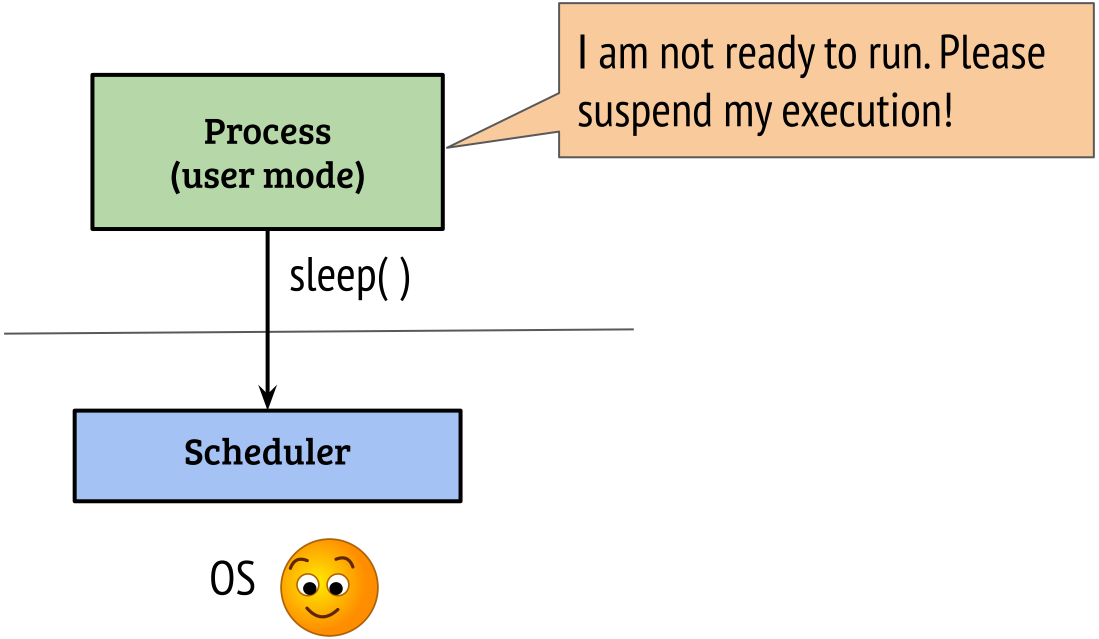
Today's agenda: Process context switch and scheduling

Triggers for process context switch



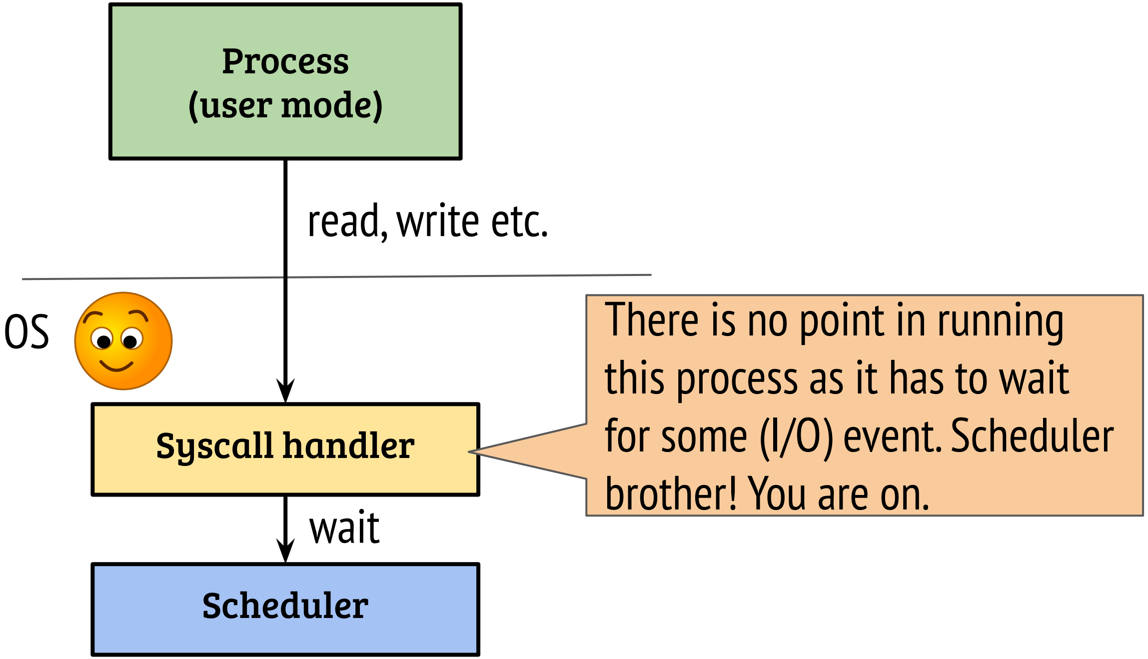
- The user process can invoke the scheduler through explicit system calls like sched_yield (see man page)

Triggers for process context switch



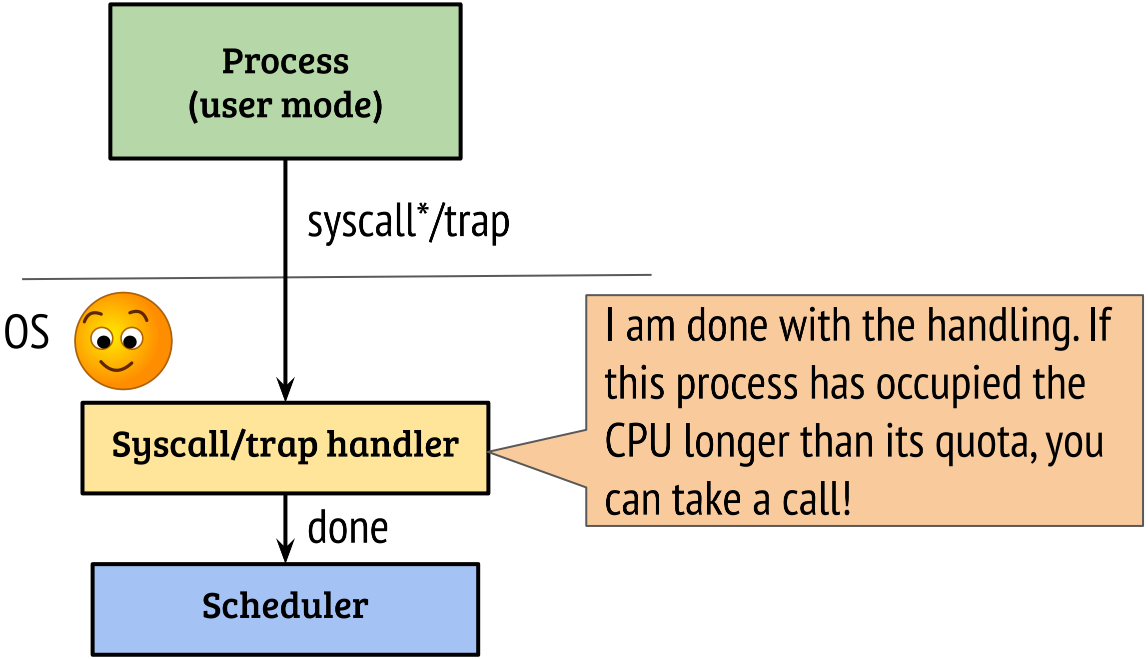
- The user process can invoke sleep( ) to suspend itself
    - sleep( ) is not a system call in Linux, it uses nanosleep( ) system call

Triggers for process context switch



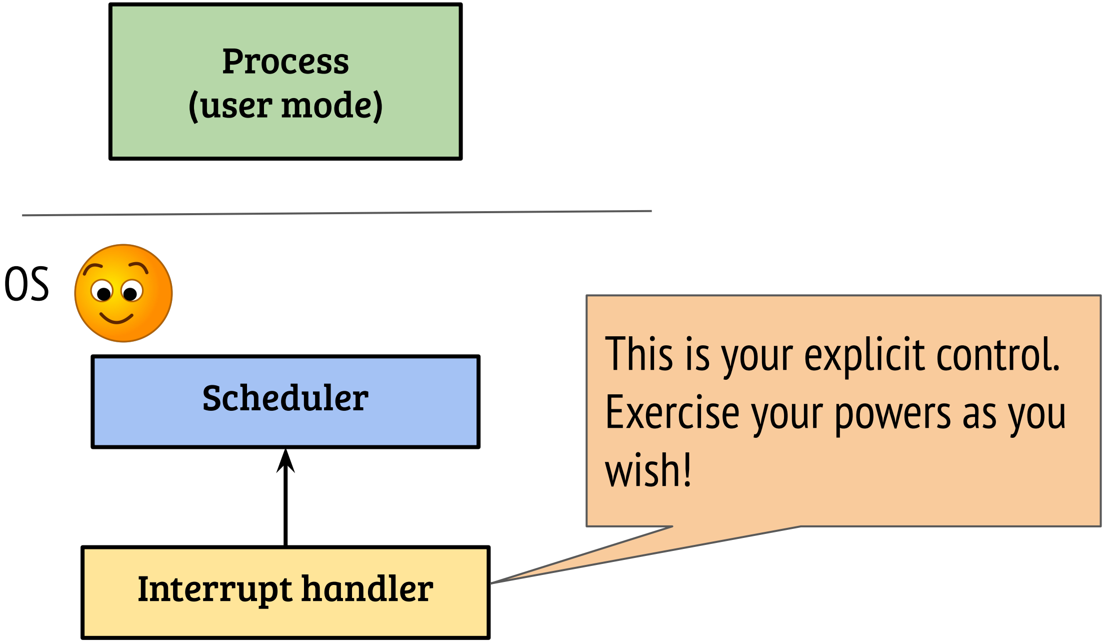
- This condition arises mostly during I/O related system calls
    - Example: read( ) from a file on disk

Triggers for process context switch



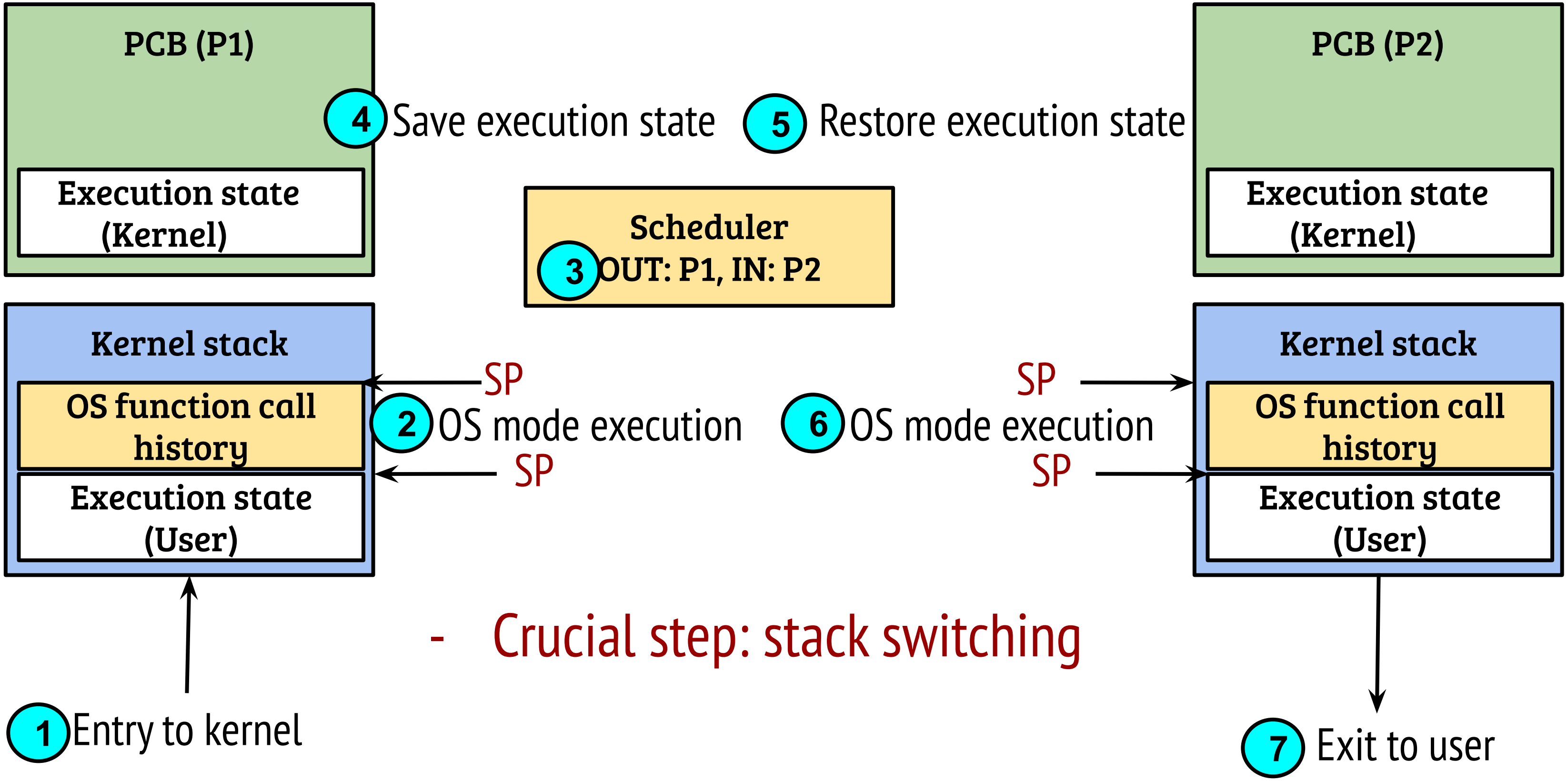
- The OS gets the control back on every system call and exception
- Before returning from syscall, the schedule can deschedule

Triggers for process context switch



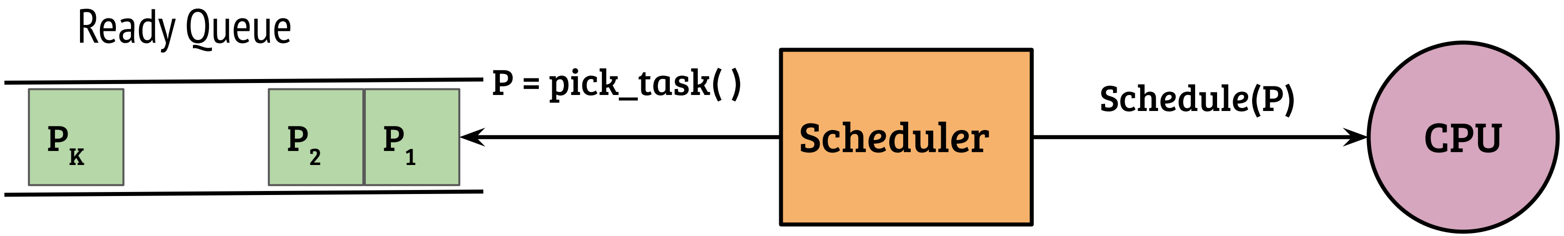
- Timer interrupts can be configured to generate interrupts periodically or after some configured time
- The OS can invoke the scheduler after handling any interrupt

Process context switch



- Crucial step: stack switching

Scheduling



- A queue of processes ready to execute is maintained
- The scheduler decides to pick the next process based on some scheduling policy and performs a context switch
- The outgoing process is put back to ready queue (if required)

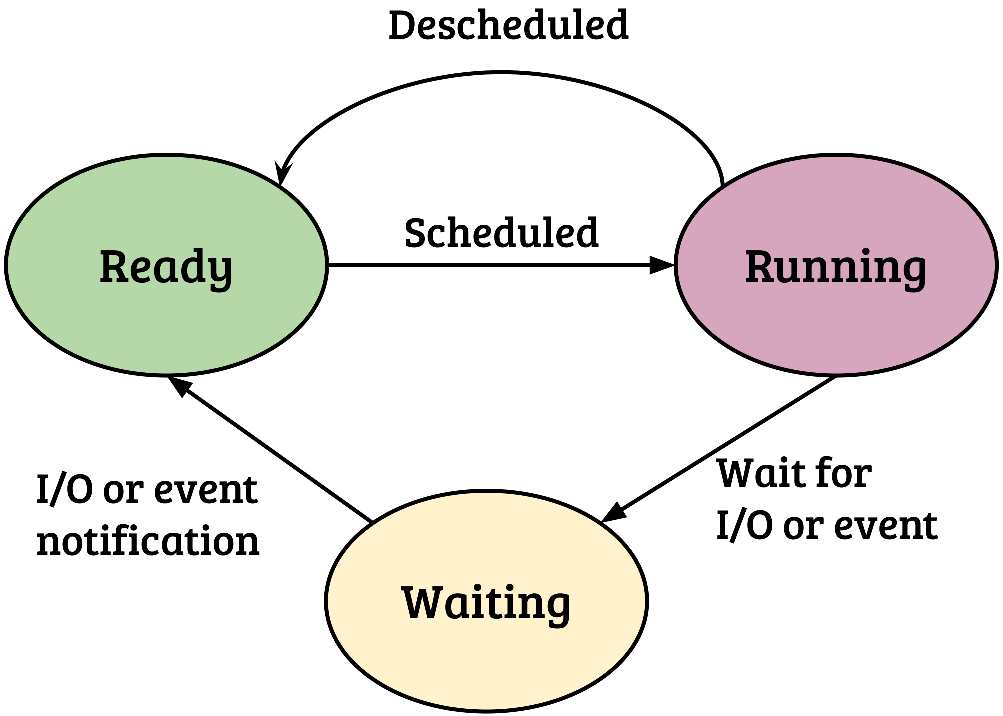
Scheduling

- How is the list of ready processes managed?
- What if there are no processes in ready queue? Can that happen?
- Can we classify the schedulers based on how they are invoked?
- What is a good scheduling strategy?

policy and performs a context switch

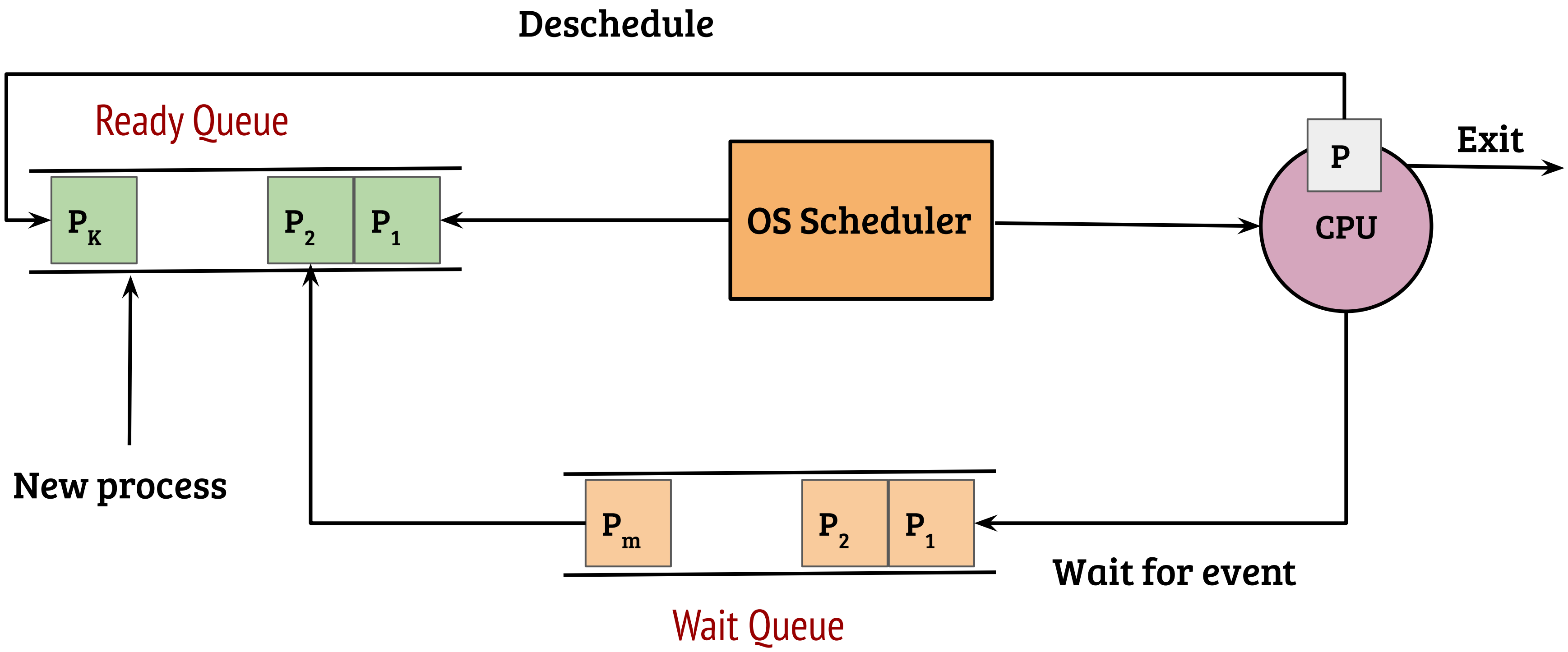
- The outgoing process is put back to ready queue (if required)

Process states and transitions



- Most processes perform a mixture of CPU and I/O activities
- When the process is waiting for an I/O, it is moved to waiting state
- A process becomes ready again when the event completion is notified (e.g., a device interrupt)

Scheduler overview



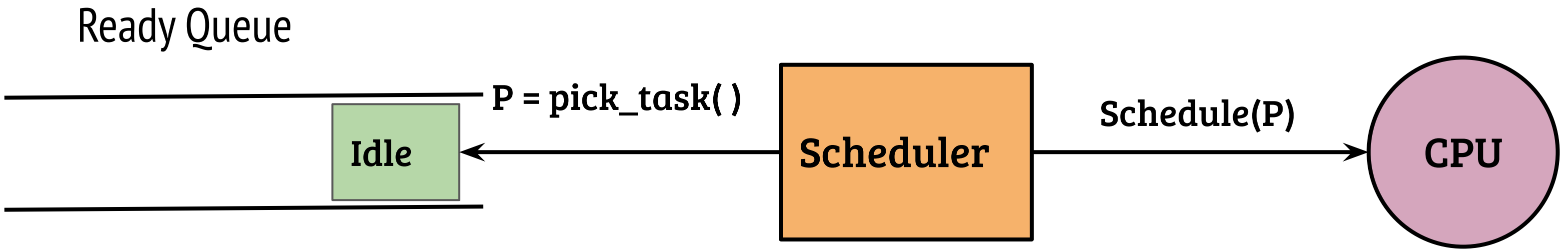
Scheduling

Ready Queue

- How is the list of ready processes managed?
- Each process is associated with three primary states: Running, Ready and Waiting. A process can moved to waiting state from running state, if needed.
- What if there are no processes in ready queue? Can that happen?
- Can we classify the schedulers based on how they are invoked?
- What is a good scheduling strategy?

- The outgoing process is put back to ready queue (if required)

System idle process



- There can be an instance when there are zero processes in ready queue
- A special process (system idle process) is always there
- The system idle process halts the CPU
- HLT instruction on X86_64: Halts the CPU till next interrupt

Scheduling

Ready Queue

- How is the list of ready processes managed?
- Each process is associated with three primary states: Running, Ready and Waiting. A process can moved to waiting state from running state, if needed.
- What if there are no processes in ready queue? Can that happen?
- There is always an idle process which executes HLT
- Can we classify the schedulers based on how they are invoked?
- What is a good scheduling strategy?

Scheduling: preemptive vs. non-preemptive

- There are scheduling points which are triggered because of the current process execution behavior (non-preemptive)
  - Process termination
  - Process explicitly yields the CPU
  - Process waits/blocks for an I/O or event

Scheduling: preemptive vs. non-preemptive

- There are scheduling points which are triggered because of the current process execution behavior (non-preemptive)
    - Process termination
    - Process explicitly yields the CPU
    - Process waits/blocks for an I/O or event
- The OS may invoke the scheduler in other conditions (preemptive)
    - Return from system call (specifically fork( ))
    - After handling an interrupt (specifically timer interrupt)

Scheduling

Ready Queue

- How is the list of ready processes managed?
- Each process is associated with three primary states: Running, Ready and Waiting. A process can moved to waiting state from running state, if needed.
- What if there are no processes in ready queue? Can that happen?
- There is always an idle process which executes HLT
- Can we classify the schedulers based on how they are invoked?
- Non-preemptive: triggered by the process, Preemptive: OS interjections
- What is a good scheduling strategy?

Scheduling metrics

- Turnaround time: Time of completion - Time of arrival
    - Objective: *Minimize turnaround time*

Scheduling metrics

- Turnaround time: Time of completion - Time of arrival
  - Objective: *Minimize turnaround time*
- Waiting time: Sum of time spent in ready queue
  - Objective: *Minimize waiting time*

Scheduling metrics

- Turnaround time: Time of completion - Time of arrival
  - Objective: *Minimize turnaround time*
- Waiting time: Sum of time spent in ready queue
  - Objective: *Minimize waiting time*
- Response time: Waiting time before first execution
  - Objective: *Minimize response time*

Scheduling metrics

- Turnaround time: Time of completion - Time of arrival
  - Objective: *Minimize turnaround time*
- Waiting time: Sum of time spent in ready queue
  - Objective: *Minimize waiting time*
- Response time: Waiting time before first execution
  - Objective: *Minimize response time*
- Average value of above metrics represent the average efficiency

Scheduling metrics

- Turnaround time: Time of completion - Time of arrival
  - Objective: *Minimize turnaround time*
- Waiting time: Sum of time spent in ready queue
  - Objective: *Minimize waiting time*
- Response time: Waiting time before first execution
  - Objective: *Minimize response time*
- Average value of above metrics represent the average efficiency
- Standard deviation represents fairness across different processes

First Come First Served (FCFS)

- FIFO queue based non-preemptive scheduling
- Example

First Come First Served (FCFS)

- FIFO queue based non-preemptive scheduling
- Example
- Advantages
  - Easy to implement
- Issues with FCFS
  - Convoy effect
  - Not suitable for interactive applications

Shortest Job First (SJF)

- Select the process with shortest CPU burst
- Pick the next process only when the current process is finished (non-preemptive)
- Example

Shortest Job First (SJF)

- Select the process with shortest CPU burst
- Pick the next process only when the current process is finished (non-preemptive)
- Example
- Optimal on waiting time and turnaround time
- Not realistic (how can we know the execution time?)

Shortest Time to Completion First (STCF)

- Pick the process with shortest remaining time when a new process arrives in the ready queue (SRTF)
- Example
- Improves the efficiency of SJF at the cost of more context switches

Round-robin scheduling

- Preemptive scheduling with time slicing
- Ready queue is maintained as a circular queue
- At end of the time quantum, If there are other processes in the queue
  - Current process goes to the TAIL of the queue
  - Next process is picked up from the HEAD of the queue
- New processes are added to the TAIL of the queue
- Design choice: size of time quantum

Priority scheduling

- Select the process with highest priority
- Can be preemptive and non-preemptive
- SJF: priority defined by job length
- Advantages: practical (no assumptions)
- Disadvantages: Starvation

Problem formulation with I/O bursts

Process	Arrival Time	CPU bursts	I/O bursts
P1	0	0-3, 7-9, 14-15	3-7,9-14
P2	2	2-10, 12-15	10-12
P3	3	3-4, 10-11	4-10

- Most processes goes through a series of CPU and I/O bursts
- Looks complicated for analysis, can it be simplified?

Problem formulation with I/O bursts

Process	Arrival Time	CPU bursts	I/O bursts
P1	0	0-3, 7-9, 14-15	3-7,9-14
P2	2	2-10, 12-15	10-12
P3	3	3-4, 10-11	4-10

- Most processes require a series of CPU and I/O bursts
- Looks complicated for analysis, can it be simplified?
- Every CPU burst is treated as a new process where the CPU burst start is the process arrival time and burst length is the execution time