Multi Processes and Scheduling

ECE 469, Feb 15

Aravind Machiry

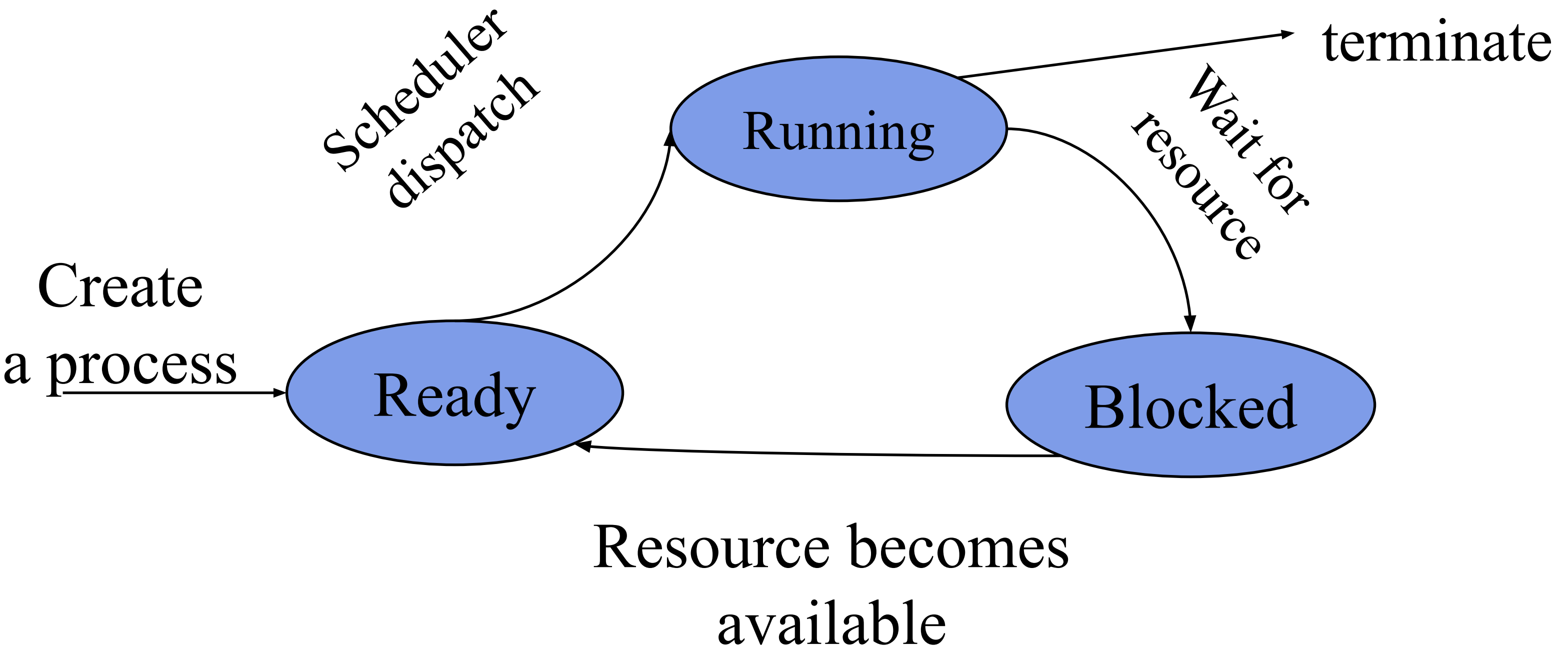
Recap: Users, Programs, Processes

- Users have accounts on the system
- Users launch programs
- There can be multiple programs (i.e., processes), which want to run at the same time

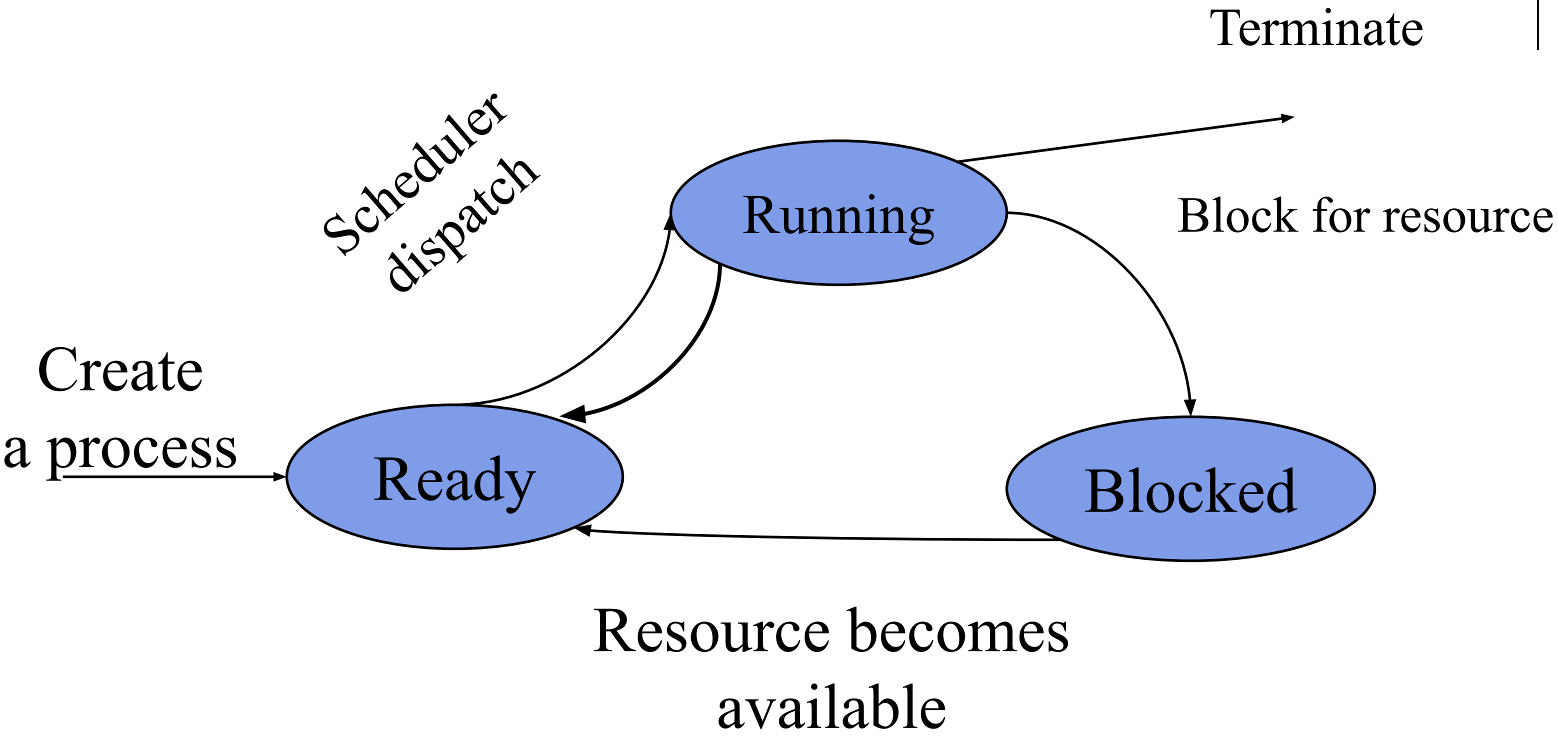
Sequential execution of each process

- Assuming single-threaded program
- No concurrency inside a process
- Everything happens sequentially
- Often with interleaved CPU/IO operations

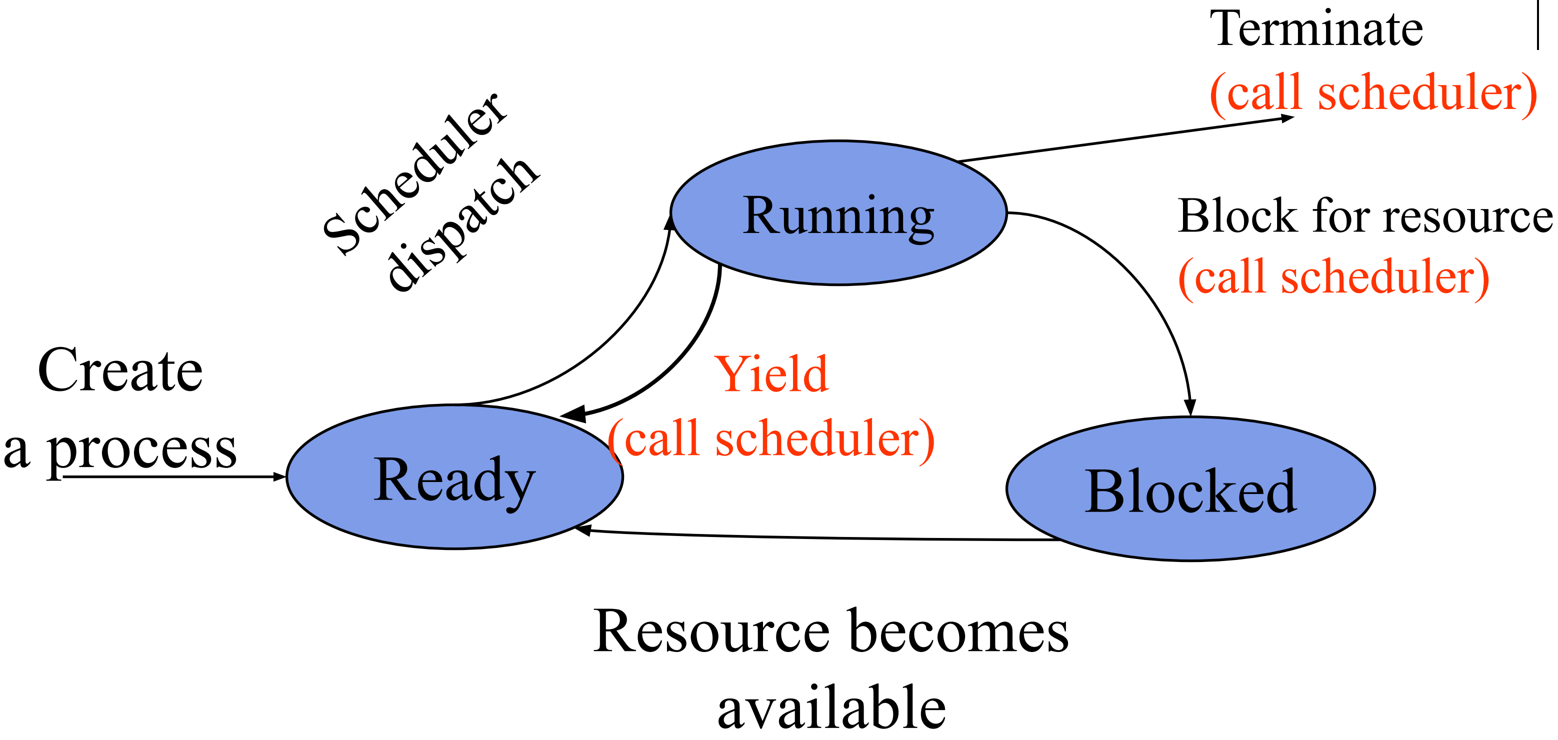
Process Life Cycle



Non-Preemptive Scheduling



Non-Preemptive Scheduling



Non-Preemptive Scheduling

- Any issues?
- What if a process runs:

```
int main() {
    while(1);
}
```

Concurrent Processes

- Processes in a system can execute concurrently (multitasking)

- Motivations for allowing concurrent execution
  - Physical resource sharing (system utilization)
  - Computational speedup – with several CPUs
  - Modularity (chrome)
  - Convenience (desktop: chrome, google drive, clock, weather)

- Logical resource sharing (eg password files)

Time Sharing Systems

- Timesharing systems support interactive use:
  - each user feels he/she has the entire machine
- How?
  - optimize response time
  - based on time-slicing

Preemptive Scheduling

- Basic idea
  - before moving process to running, OS sets timer
  - if process yields/blocks, clear timer, go to scheduler
  - If timer expires, go to scheduler

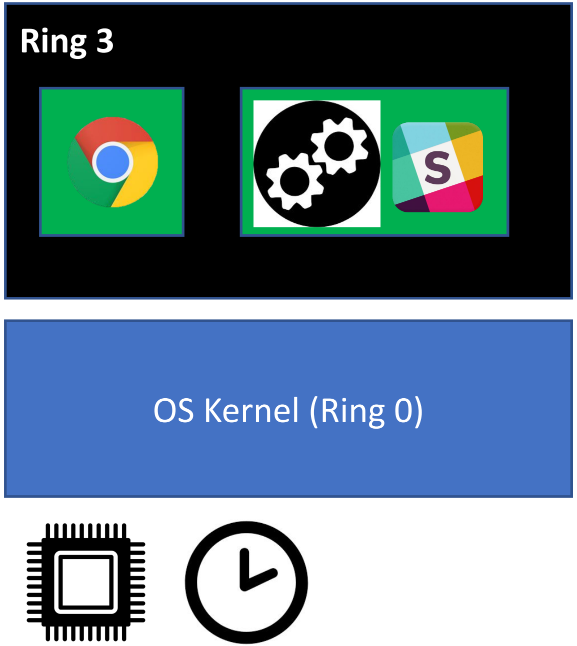
Preemptive Scheduling

- How does the OS know that the timer expired?

```c
int main() {
    while(1);
}
```

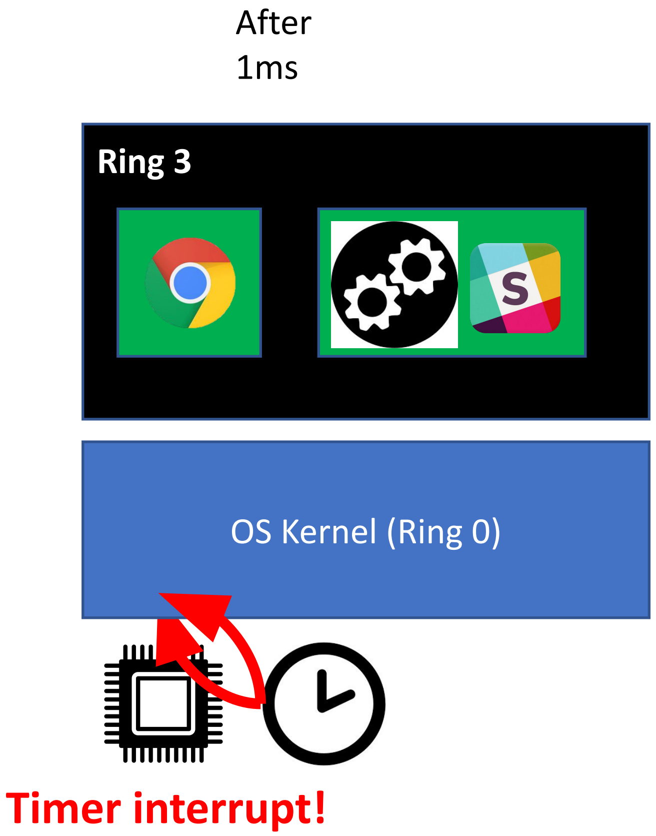
Preemptive Scheduling

- Preemptive Multitasking (Lab 4)
- CPU generates an interrupt to force execution at kernel after some time quantum
  - E.g., 1000Hz, on each 1ms..



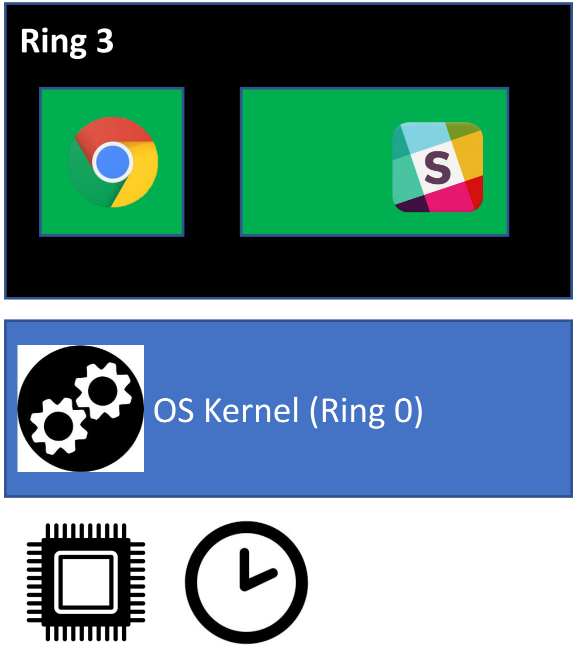
Preemptive Scheduling

- Preemptive Multitasking (Lab 4)
- CPU generates an interrupt to force execution at kernel after some time quantum
  - E.g., 1000Hz, on each 1ms..



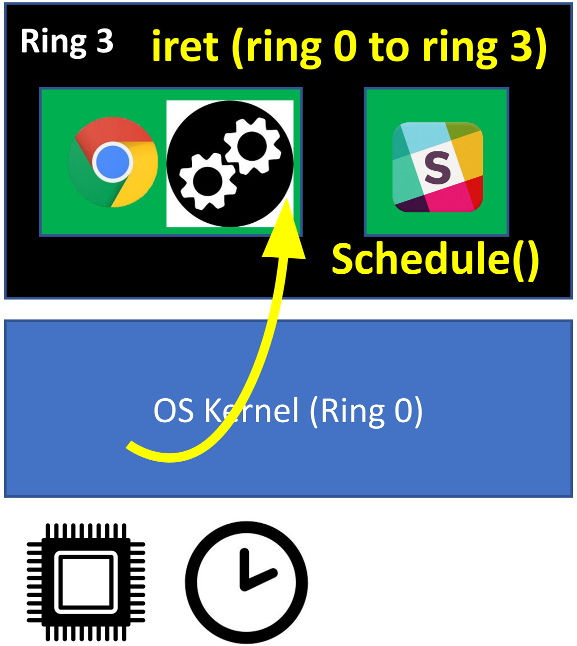
Preemptive Scheduling

- Preemptive Multitasking (Lab 4)
- CPU generates an interrupt to force execution at kernel after some time quantum
  - E.g., 1000Hz, on each 1ms..
- Guaranteed execution in kernel
  - Let kernel mediate resource contention



Preemptive Scheduling

- Preemptive Multitasking (Lab 4)
- CPU generates an interrupt to force execution at kernel after some time quantum
  - E.g., 1000Hz, on each 1ms..
- Guaranteed execution in kernel
  - Let kernel mediate resource contention



Context Switch

- Definition: Switching the CPU to another process, which involves saving the state of the old process and loading the state of the new process

- **What state**?

- Where to store them?

Process State: Process Control Block (PCB)

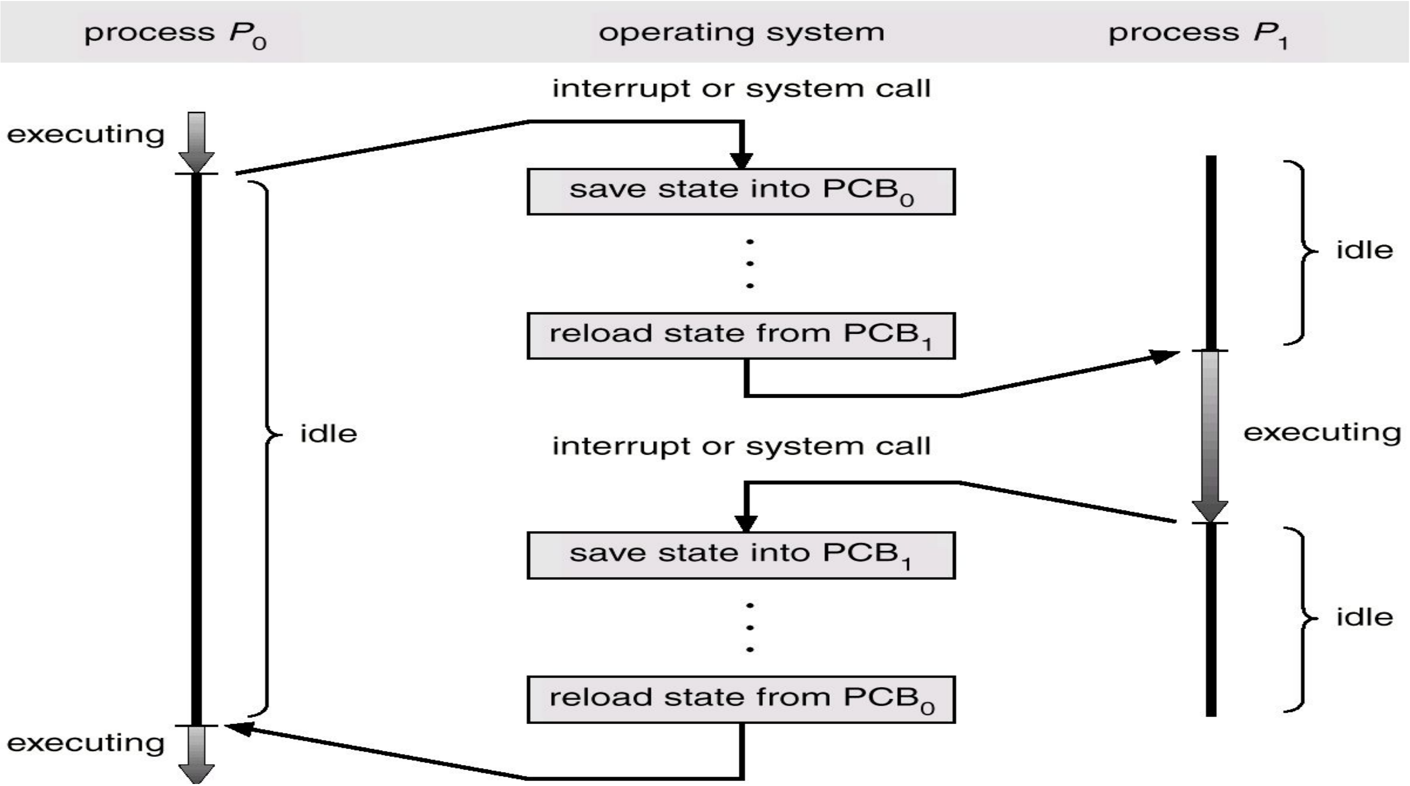
- A.K.A User Environment (JOS)

- Process management info
  - State (ready, running, blocked)
  - PC & Registers, parents, etc
  - CPU scheduling info (priorities, etc.)

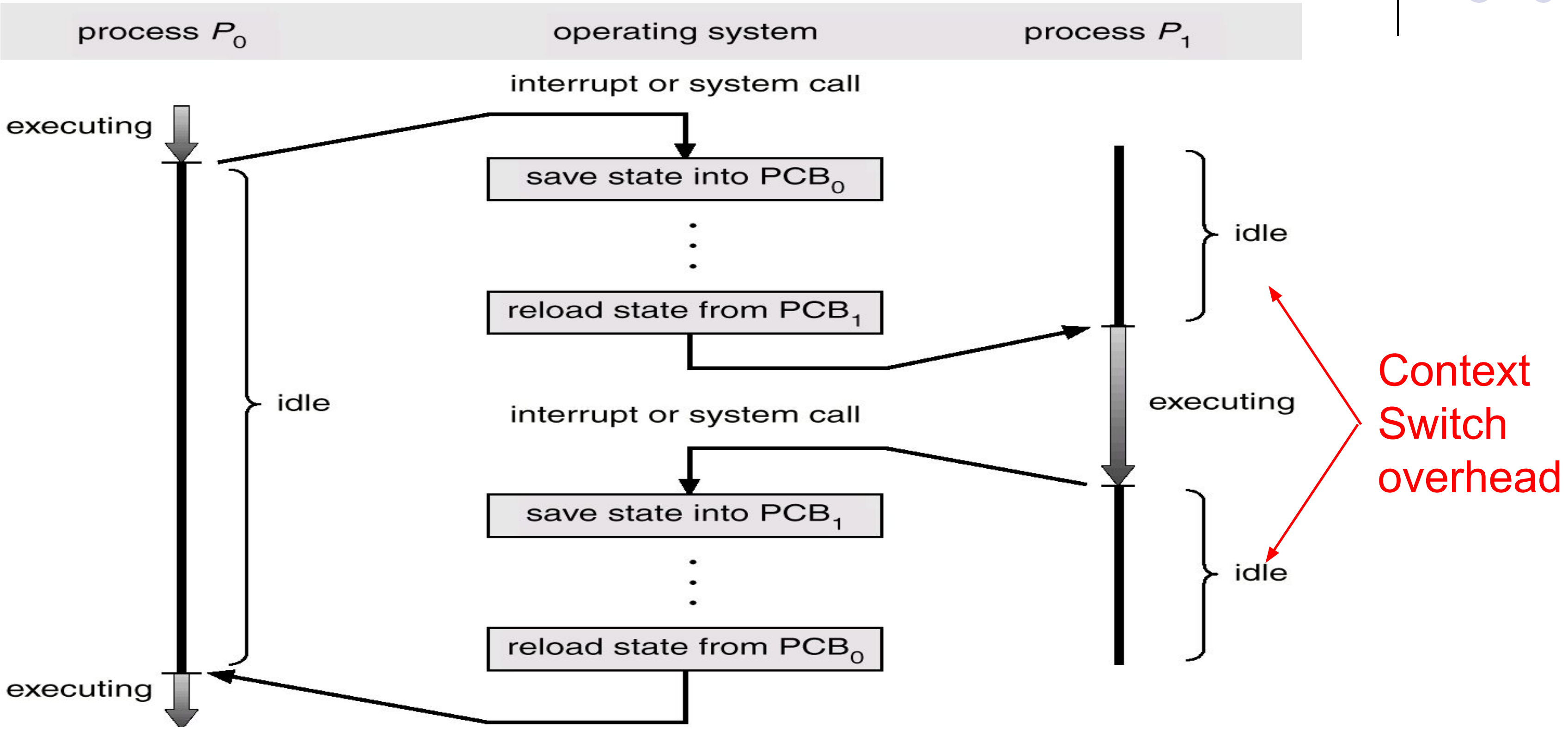
- Memory management info
  - Segments, page table, stats, etc

- I/O and file management
  - Communication ports, directories, file descriptors, etc

Context Switch



Context Switch



Preemptive Scheduling Considerations

- Timer granularity
  - **Finer timers = more responsive, high overhead**
  - **Coarser timers = less responsive, more efficient**

- CPU Accounting (CPU running stats)
  - Used by the scheduler
  - Useful for the programmer

Preemptive Scheduling Considerations

- Mechanism + policy
- Mechanisms fairly simple:
  - Save state into a PCB and Restore state from another PCB

Preemptive Scheduling Considerations

- Mechanism + policy
- Mechanisms fairly simple:
    - Save state into a PCB and Restore state from another PCB
- Policy choices harder:
    - When should we switch?

Challenges in Policy

- Flexibility - variability in job types
  - Long vs. short
  - Interactive vs. non-interactive
  - I/O-bound vs. compute-bound
- Issues
  - Short jobs shouldn't suffer
  - (Interactive) Users shouldn't be annoyed

Challenges in Policy (2)

- Fairness
  - All users should get access to CPU
  - Amount of CPU should be roughly even?

- Issue
  - Short-term vs. long-term fairness

Goals

- Goals (Performance metrics)
  - Minimize turnaround time
    - avg time to complete a job
    - $T_{turnaround} = T_{completion} - T_{arrival}$
  - Maximize throughput
    - operations (jobs) per second
    - Minimize overhead of context switches: large quanta
    - Efficient utilization (CPU, memory, disk etc)
  - Short response time
    - $T_{response} = T_{firstrun} - T_{arrival}$
    - type on a keyboard
    - Small quanta
  - Fairness
    - fair, no stavaton, no deadlock

Goals

- Goals often conflict
  - Response time vs. throughput
  - fairness vs. avg turnaround time?

Goals and Assumptions

- Goals (Performance metrics)
  - Minimize turnaround time
    - avg time to complete a job
    - $T_{turnaround} = T_{completion} - T_{arrival}$
  - Maximize throughput
    - operations (jobs) per second
    - Minimize overhead of context switches: large quanta
    - Efficient utilization (CPU, memory, disk etc)
  - Short response time
    - $T_{response} = T_{firstrun} - T_{arrival}$
    - type on a keyboard
    - Small quanta
  - Fairness
    - fair, no stavaton, no deadlock

Scheduling Policies

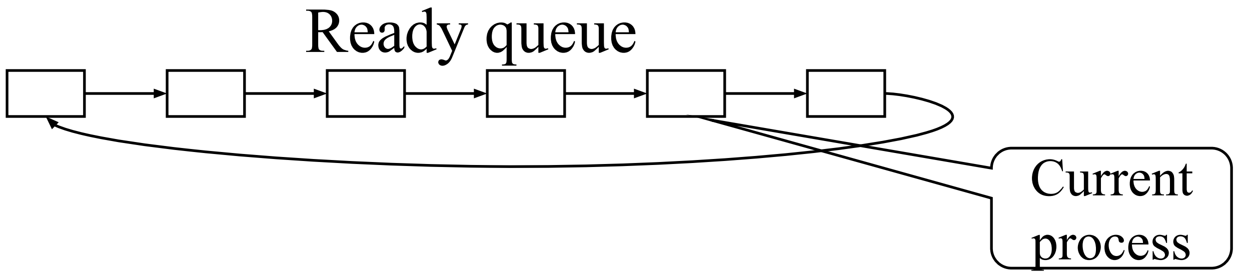
- Is there an optimal scheduling policy?
  - Even if we narrow down to one goal?

- But we don't know about future
  - Offline vs. online

Scheduling Policies

- Round Robin
- SJCF
- SRTCF

Round Robin



- Each runs a time slice or quantum: Fair
- How do you choose time slice?
  - Overhead vs. response time
  - Overhead is typically about 1% or less
  - Quantum typically between 10 ~ 100 millisec

Is Fairness always good?

- Assume 10 jobs waiting to be scheduled, each takes 100 seconds
  - Assume no other overhead
  - Total CPU time? 1000 seconds, always

- Implications?
  - Last job always finishes at 1000 seconds
  - So what's the point of scheduling?

Non-Preemptive Scheduling

- Job 1 – start 0, end 100
- Job 2 – start 100, end 200
- …
- Job 10 – start 900, end 1000

- Average turnaround time =100 + 200 +… /10 = 550 sec

Round Robin

- Assume each quantum is 1 second

- Job 0 – 0, 10, 20, 30, 40,…, 990
- Job 1 – 1, 11, 21, 31,…, 991
- Job 2 – 2, 12, 22, 32,…, 992
- …

- Avg turnaround time = 990+991+…/10 = 995 sec

Is Fairness always good?

- Unfair policy was faster!
- Job 10 always ended at the same time
- Round-Robin just hurt jobs 1-9 with no gain

Why use Round Robin?

- Imagine 10 jobs
  - Jobs 1-9 are 100 seconds
  - Job 10 is 10 seconds

- Which policy is better now?

Non-preemptive scheduling

- Jobs 1-9 are 100 seconds
- Job 10 is 10 seconds

Non-preemptive scheduling

- Jobs 1-9 are 100 seconds
- Job 10 is 10 seconds

- Job 0 – start 0, end 100
- Job 1 – start 100, end 200
- Job 10 – start 900, end 910

- Avg turnaround time = 100+200+…910/N = 541

Round Robin scheduling

- Jobs 1-9 are 100 seconds
- Job 10 is 10 seconds

- Job 0 – 0, 10, 20, …, 900
- Job 1 – 1, 11, 21, …, 901
- Job 10 – 9, 19, 29, …, 99

- Avg turnaround time = 900 + 901 + 908 + 99 / 10 = 824

Round Robin scheduling

- Jobs 1-9 are 100 seconds
- Job 10 is 10 seconds

- Job 0 – 0, 10, 20, …, 900
- Job 1 – 1, 11, 21, …, 901
- Job 10 – 9, 19, 29, …, 99

9% work drop

2% avg turnaround drop for FIFO

17% avg turnaround drop for RR

- Avg turnaround time = 900 + 901 + 908 + 99 / 10 = 824

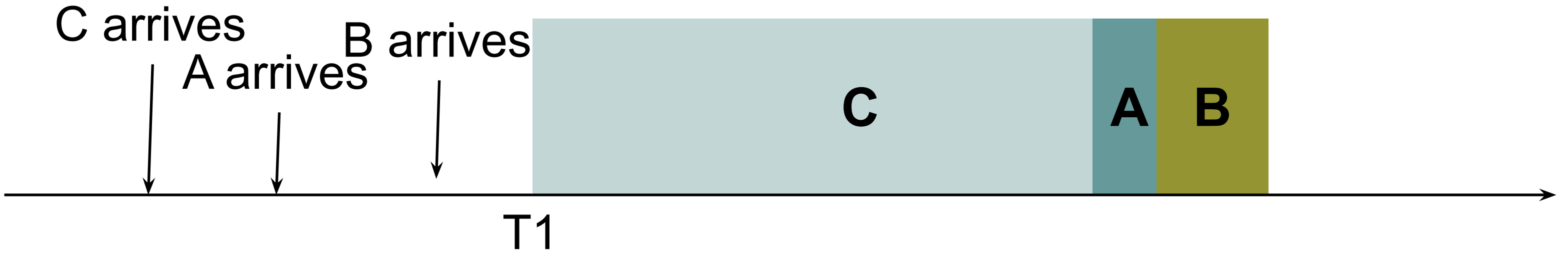
Why use Round Robin?

- Imagine 10 jobs
  - Jobs 1 is 100 seconds
  - Job 2-10 is 10 seconds

- Which policy is better now?
  - FIFO: average turnaround 145
  - RR: average turnaround 105

STCF (SJF) – Shortest Job First

- What shall we do if we care about turn-around time?
  - FIFO can be bad



- STCF/SJF
  - schedule shortest (total completion time) job first

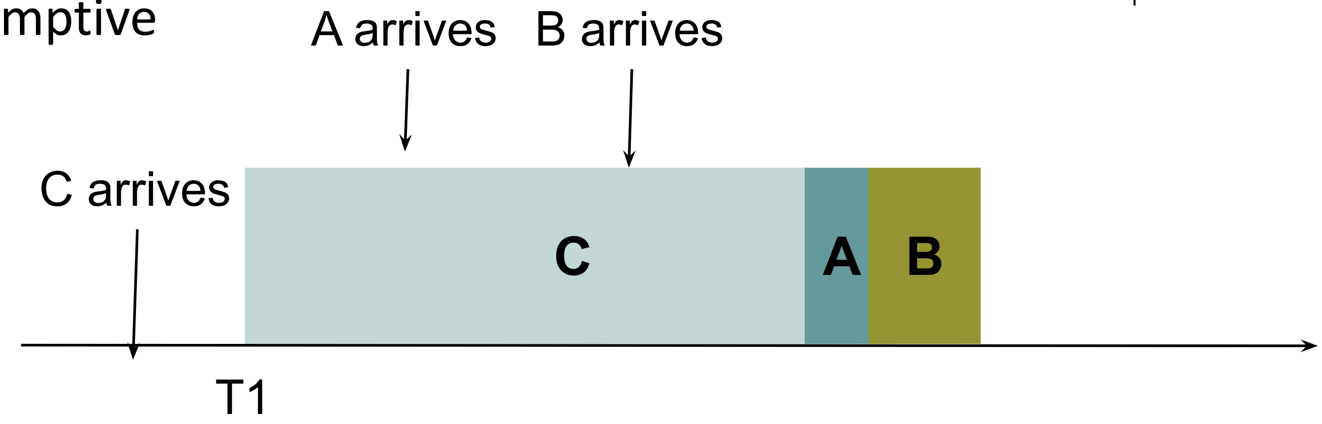
SJF: Pros and Cons

- Can we do better than Shortest Job First in terms of average turnaround time?
    - Assume all jobs arrive at the beginning
- In fact, SJF can be proved to be the optimal scheduling algorithm with the above assumption
    - But we are not going to prove it, since this is not a theory class 😊

- SJF Advantage
    - Minimal average turnaround time
- Disadvantage
    - Difficult to know the future, has to run until finish

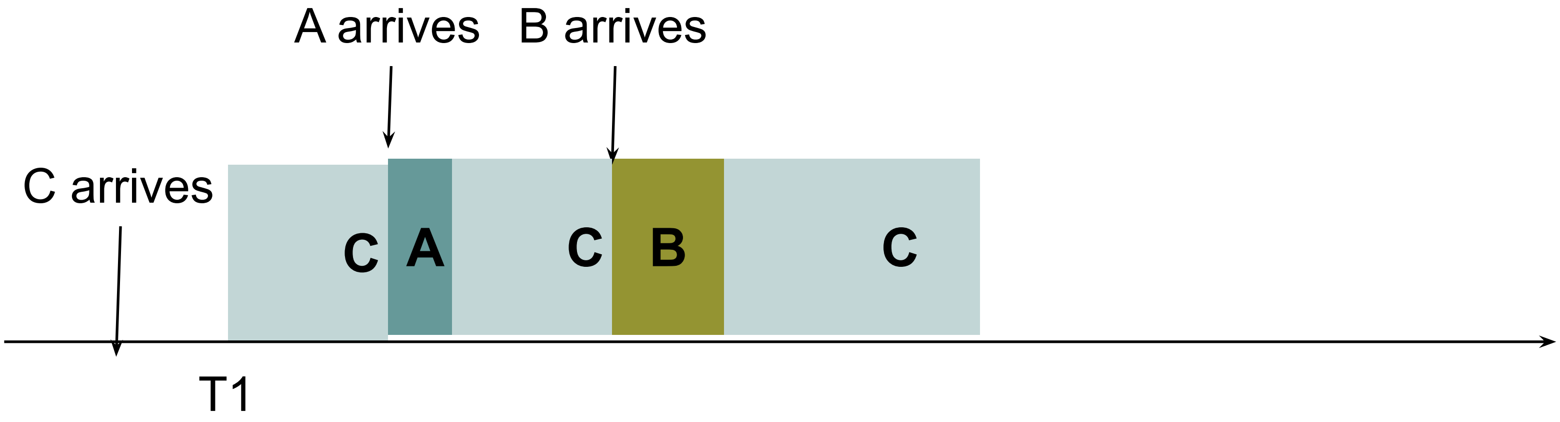
STCF

- Shortest time to completion first (shortest job first)
  - Non-preemptive



SRTCF

- Shortest remaining time to completion first
  - Preemptive



Any potential problems?
- Can cause starvation!

Policy Decisions

- Need to accommodate interactive jobs
  - Need some kind of RR
- Diversity in jobs – job length, I/O mix
  - RR also appears to help
- SJF also has virtue
  - Reduce avg. turnaround time
- Can we accommodate all?

Scheduling Policies Advantages

FIFO	Response time
RR	Throughput
SJF	Avg. turnaround time
	Fairness

Scheduling Policies Advantages

FIFO	Response time
	Throughput
RR	
	Avg. turnaround time
SJF	
	Fairness

Scheduling Policies Advantages

FIFO

RR

SJF

Response time

Throughput

Avg. turnaround time

Fairness

Scheduling Policies Advantages

FIFO

RR

SJF

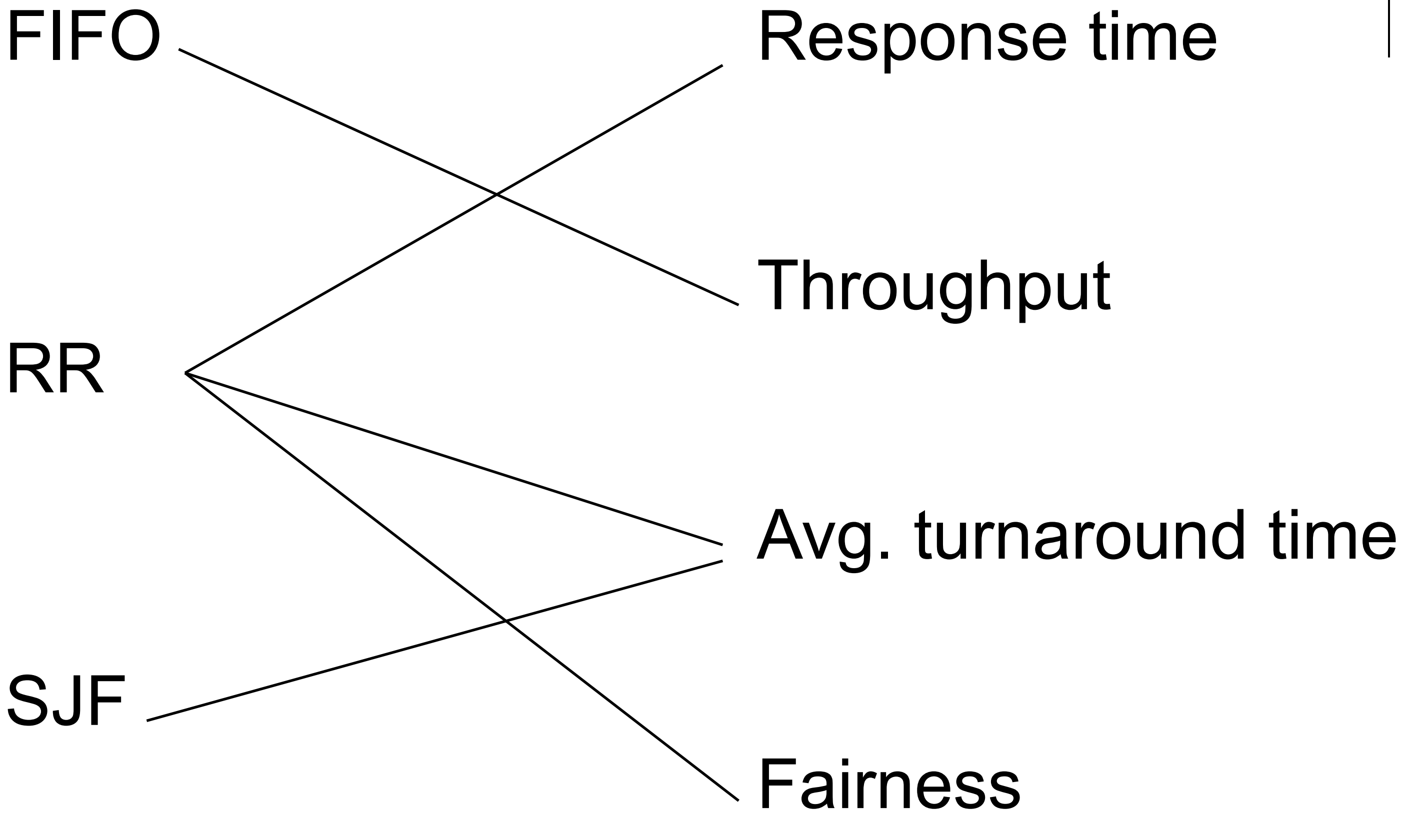
Response time

Throughput

Avg. turnaround time

Fairness

Scheduling Policies Advantages



Scheduling Policy Issues

- Fairness
- Flexibility
- **High utilization (efficiency)**
- Good response time
- Good turnaround time

Scheduling Policy Issues

- High utilization (efficiency)
  - Lots of processes (want diff resources)
  - Lots of resources (want full parallelism)

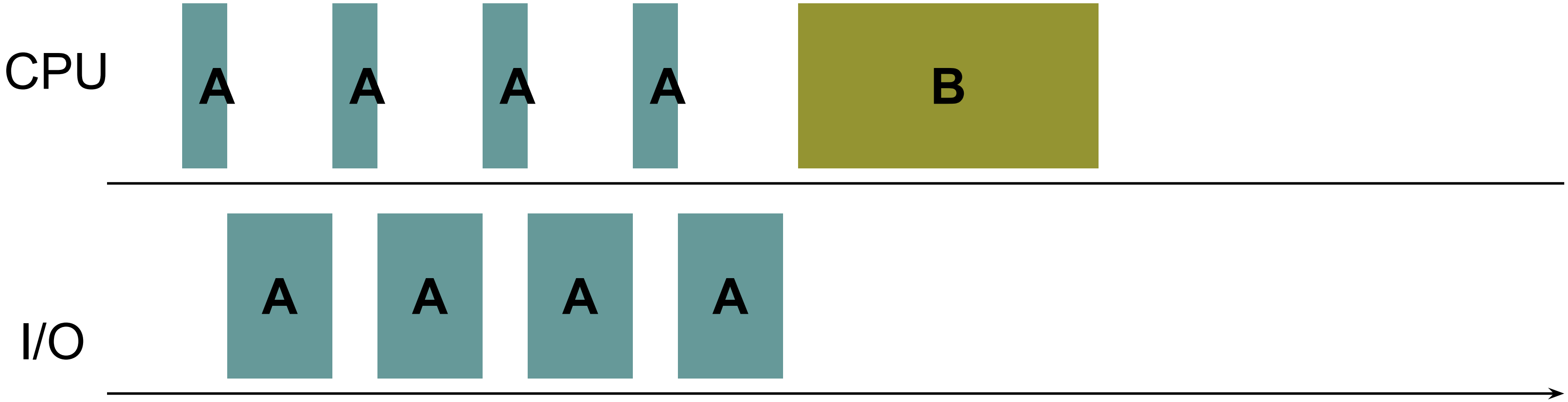
- Issue?
  - How do you get the most useful work out of the system? (job throughput)

Adding I/O into mix

- Resource utilization example
  - A and B each uses 100% CPU
  - C loops forever (1ms CPU and 10ms disk)
  - Time slice 99ms: roughly 30% of disk utilization with Round Robin and roughly 70% of CPU utilization
  - Time slice 1ms: roughly 90% of disk utilization with Round Robin and nearly 100% of CPU utilization

- What do we learn from this example?
  - Small time slice can improve utilization / fairness to I/O jobs

Handling I/Os



Handling I/Os