# Scheduling Policies and Introduction to Signals

Lecture 14

Klara Nahrstedt

# CS241 Administrative

- Read Chapter 9. 1, 9.2 Stallings for Scheduling Policies
- Read R&R Chapter 8.1 and 8.2 for Introduction to Signals
- **SMP4** is ON this week (2/19-2/26)
- **SMP3 Quiz** will be on Wednesday (2/21)
- **Regular Quiz** will be on Friday (2/23) and it will cover Scheduling, Scheduling Policies and Signals
  - Chapters 9.1-9.2 (S) and Chapters 8.1-8.5 (R&R)

# Content of This Lecture

Basic scheduling algorithms

- Problems with FIFO (FCFS)
- Shortest job first
- Round Robin
- Priority Scheduling

Goals:

- Understand how your program is executed on the machine together with other programs
- SMP4

# Review: Simple Processor Scheduling Algorithms

- Batch systems
  - First Come First Serve (FCFS)
  - Shortest Job First
- Interactive Systems
  - Round Robin
  - Priority Scheduling
  - …

# Review: First Come First Serve (FCFS)

Process that requests the CPU FIRST is allocated the CPU FIRST.

Also called FIFO

Preemptive or Non-preemptive?

Used in Batch Systems

Real life analogy?

- Fast food restaurant

Implementation

- FIFO queues
- A new process enters the tail of the queue
- The schedule selects from the head of the queue.

Performance Metric: **Average Waiting Time.**

Given Parameters:

- Burst Time (in ms), Arrival Time and Order

# Problems with FCFS

Non-preemptive

Not optimal AWT

Cannot utilize resources in parallel:

- Assume 1 process CPU bounded and many I/O bounded processes
- result: Convoy effect, low CPU and I/O Device utilization
- Why?

# Why Convoy Effects?

Consider n-1 jobs in system that are I/O bound and 1 job that is CPU bound.

I/O bound jobs pass quickly through the ready queue and suspend themselves waiting for I/O.

CPU bound job arrives at head of queue and executes until complete.

I/O bound jobs rejoin ready queue and wait for CPU bound job to complete.

I/O devices idle until CPU bound job completes.

When CPU bound job completes, other processes rush to wait on I/O again.

CPU becomes idle.

# Exam Question

Would FCFS be an Appropriate Scheduling Policy for your Laptop OS? Explain why yes or why not!

# Interactive Scheduling Algorithms

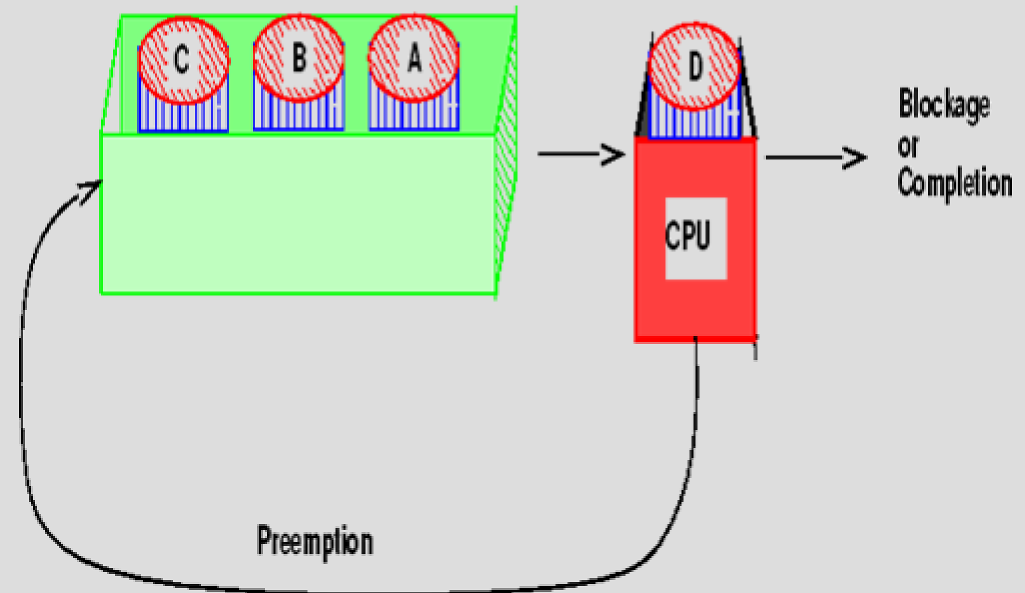
- Usually preemptive
  - Time is **sliced** into quantum (time intervals)
  - Scheduling decision is also made at the beginning of each quantum
- Performance Criteria
  - Min Response time
  - best proportionality
- Representative algorithms:
  - Priority-based
  - Round-robin
  - …

# Round-robin

One of the oldest, simplest, most commonly used scheduling algorithm

Select process/thread from ready queue in a round-robin fashion (take turns)

Problem:
- Do not consider priority
- Context switch overhead



# Round-robin: Example

| Process | Duration | Order | Arrival Time |
|---|---|---|---|
| P1 | 3 | 1 | 0 |
| P2 | 4 | 2 | 0 |
| P3 | 3 | 3 | 0 |

Suppose time quantum is: 1 unit, P1, P2 & P3 never block

P1 P2 P3 P1 P2 P3 P1 P2 P3 P2

0 10

P1 waiting time: 4
P2 waiting time: 6
P3 waiting time: 6

The average waiting time (AWT):
(4+6+6)/3 = 5.33

# Time Quantum

- Time slice too large
  - FIFO behavior
  - Poor response time
- Time slice too small
  - Too many context switches (overheads)
  - Inefficient CPU utilization
- Heuristic:
  - 70-80% of jobs block within time-slice
- Typical time-slice
  - 10 to 100 ms
- Time spent in system depends on size of job.

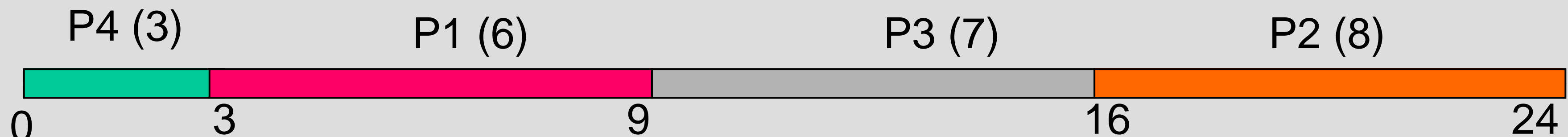
# Shortest Job First (SJF)

- Schedule the job with the shortest elapse time first
- Scheduling in Batch Systems
- Two types:
  - Non-preemptive
  - Preemptive (more advanced)
- Requirement: the elapse time needs to be known in advance
- Optimal if all jobs are available simultaneously (provable)
  - Gives the best possible AWT (average waiting time)

# Non-preemptive SJF: Example

| Process | Duration | Order | Arrival Time |
|---|---|---|---|
| P1 | 6 | 1 | 0 |
| P2 | 8 | 2 | 0 |
| P3 | 7 | 3 | 0 |
| P4 | 3 | 4 | 0 |

Do it yourself




P4 waiting time: 0
P1 waiting time: 3
P3 waiting time: 9
P2 waiting time: 16

The total time is: 24

The average waiting time (AWT):
(0+3+9+16)/4 = 7

# Comparing to FCFS

| Process | Duration | Order | Arrival Time |
|---|---|---|---|
| P1 | 6 | 1 | 0 |
| P2 | 8 | 2 | 0 |
| P3 | 7 | 3 | 0 |
| P4 | 3 | 4 | 0 |

Do it yourself

P1 waiting time: 0
P2 waiting time: 6
P3 waiting time: 14
P4 waiting time: 21

The total time is the same.
The average waiting time (AWT):
(0+6+14+21)/4 = 10.25
(comparing to 7)

# A Problem with SJF

- Starvation
  - In some condition, a job is waiting for ever
  - Example: SJF
    - Process A with elapse time of 1 hour arrives at time 0
    - But ever 1 minute, a short process with elapse time of 2 minutes arrive
    - Result of SJF: A never gets to run
- What’s the difference between starvation and a dead lock?

# Priority Scheduling

- Each job is assigned a priority.
- FCFS within each priority level.
- Select highest priority job over lower ones.
- Rational:  higher priority jobs are more mission-critical
  - Example: DVD movie player vs. send email
- Problems:
  - May not give the best AWT
  - indefinite blocking or starvation a process

# Priority Scheduling: Example

| Process | Duration | Priority | Arrival Time |
|---|---|---|---|
| P1 | 6 | 4 | 0 |
| P2 | 8 | 1 | 0 |
| P3 | 7 | 3 | 0 |
| P4 | 3 | 2 | 0 |

Do it yourself



P2 waiting time: 0
P4 waiting time: 8
P3 waiting time: 11
P1 waiting time: 18

The average waiting time (AWT):
(0+8+11+18)/4 = 9.25
(worse than SJF)

# Set Priority

Every process has a default priority

User can also change a process priority

- The Nice command

```
NICE(1)                    User Commands                    NICE(1)

NAME
       nice - run a program with modified scheduling priority

SYNOPSIS
       nice [OPTION] [COMMAND [ARG]...]

DESCRIPTION
       Run  COMMAND  with  an adjusted scheduling priority.  With no COMMAND,
       print the current scheduling  priority.   ADJUST  is  10  by  default.
       Range goes from -20 (highest priority) to 19 (lowest).

       -n, --adjustment=ADJUST
              increment priority by ADJUST first

       --help display this help and exit
```

# Set/Get Process Priority

int getpriority(int which, id_t who);
int setpriority(int which, id_t who, int value);

```
NAME
       getpriority, setpriority - get and set the nice value

SYNOPSIS
       #include <sys/resource.h>

       int getpriority(int which, id_t who);
       int setpriority(int which, id_t who, int value);

DESCRIPTION
       The  getpriority()  function shall obtain the nice value of a process,
       process group, or user. The setpriority() function shall set the  nice
       value of a process, process group, or user to value+ {NZERO}.

       Target  processes  are  specified  by  the values of the which and who
       arguments. The which argument may be  one  of  the  following  values:
       PRIO_PROCESS,  PRIO_PGRP,  or PRIO_USER, indicating that the who argu-
       ment is to be interpreted as a process ID, a process group ID,  or  an
       effective user ID, respectively.  A 0 value for the who argument spec-
       ifies the current process, process group, or user.
```

# Introduction to Signals

- What is Signal?

A signal is a software notification to a process of an event.

- Why do we need Signals?

In systems we need to enable asynchronous events

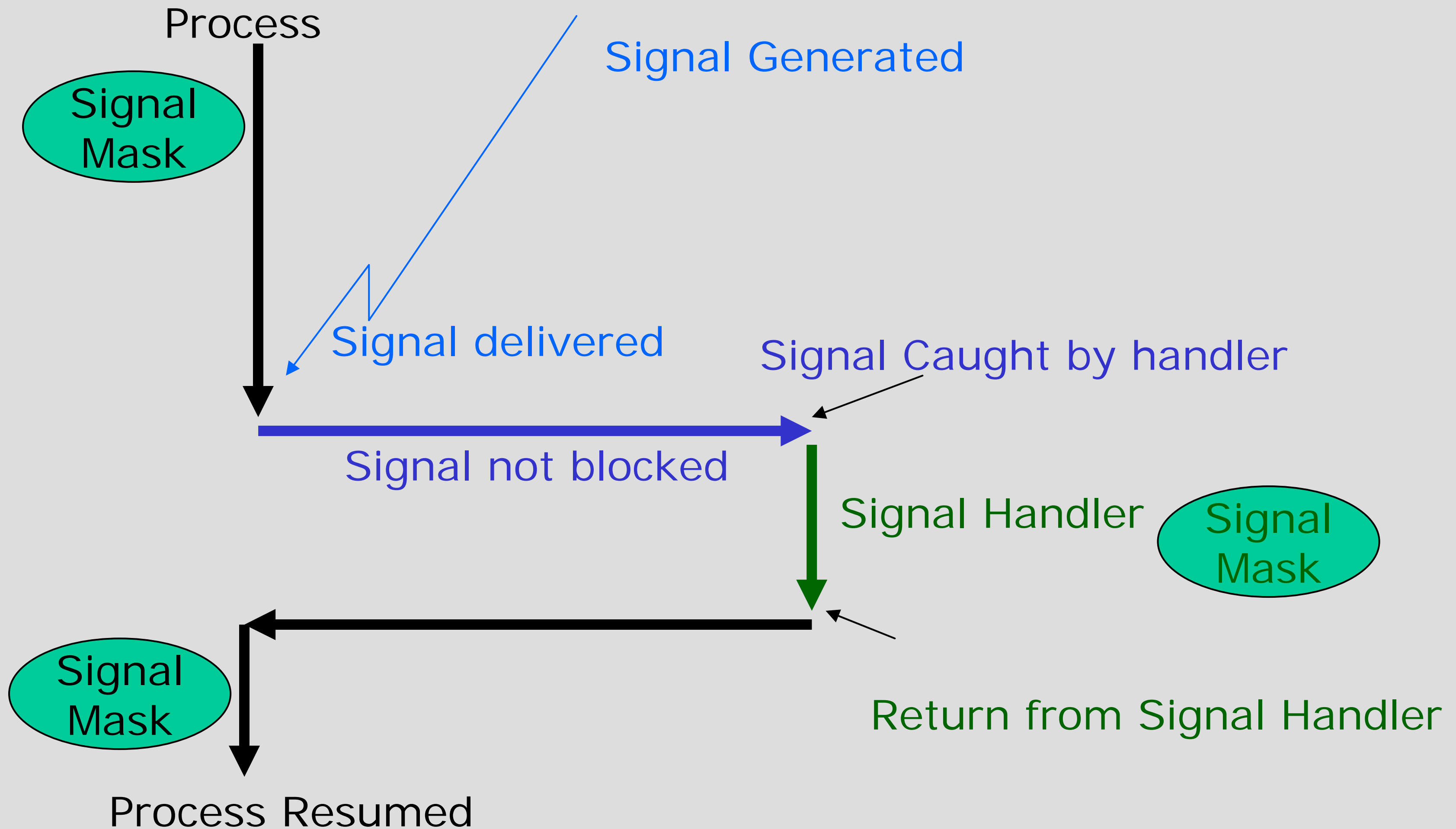
Examples of asynchronous events:

- **Email message arrives** on my machine – mailing agent (user) process should retrieve
- **Invalid memory access happens** – OS should inform scheduler to remove process from the processor
- **Alarm clock goes on** – process which sets the alarm should catch it

# Basic Signal Concept (Definitions)

- Signal is *generated* when the event that causes the signal occurs.
- Signal is *delivered* when the process takes action based on the signal.
- The *lifetime* of a signal is the interval between its generation and delivery.
- Signal that has been generated but not yet delivered is *pending*.
- Process *catches* signal if it executes *signal handler* when the signal is delivered.
- Alternatively, a process can *ignore* as signal when it is delivered, that is to take no action.
- Process can temporarily prevent signal from being delivered by *blocking* it.
- *Signal Mask* contains a set of signals currently blocked.

# How Signals Work



# Examples of POSIX Required Signals

| Signal | Description | default action |
|---|---|---|
| SIGABRT | process abort | implementation dependent |
| **SIGALRM** | **alarm clock** | **abnormal termination** |
| SIGBUS | access undefined part of memory object | implementation dependent |
| SIGCHLD | child terminated, stopped or continued | ignore |
| SIGILL | invalid hardware instruction | implementation dependent |
| **SIGINT** | **interactive attention signal (usually ctrl-C)** | **abnormal termination** |
| **SIGKILL** | **terminated (cannot be caught or ignored)** | **abnormal termination** |

| Signal | Description | default action |
|---|---|---|
| **SIGSEGV** | **Invalid memory reference** | **implementation dependent** |
| SIGSTOP | Execution stopped | stop |
| **SIGTERM** | **termination** | **Abnormal termination** |
| SIGTSTP | Terminal stop | stop |
| SIGTTIN | Background process attempting read | stop |
| SIGTTOU | Background process attempting write | stop |
| **SIGURG** | **High bandwidth data available on socket** | **ignore** |
| **SIGUSR1** | **User-defined signal 1** | **abnormal termination** |

# Generating Signals

- Signal has a symbolic name starting with SIG
- Signal names are defined in signal.h

Users can generate signals (e.g., SIGUSR1)

OS generates signals when certain errors occur
(e.g., SIGSEGV – invalid memory reference)

Specific calls generate signals such as alarm
(e.g., SIGALARM)

# Command Line Generates Signals

- You can send a signal to a process from the command line using **kill**
- kill -l will list the signals the system understands
- kill [-signal] pid will send a signal to a process.
  - The optional argument may be a name or a number.
  - The default is SIGTERM.
- To unconditionally kill a process, use:
- kill -9 pid which is
- kill -SIGKILL pid.

# Command Line Generates Signals

stty –a

- CTRL-C is SIGINT (interactive attention signal
- CTRL-Z is SIGSTOP (execution stopped – cannot be ignored)
- CTRL-Y is SIGCONT (execution continued if stopped)
- CTRL-D is SIGQUIT (interactive termination: core dump)

# Timers Generate SIGALRM Signals

```
#include <unistd.h>
unsigned alarm (unsigned seconds);
```

- alarm(20) creates SIGALRM to calling process after 20 real time seconds.
- Calls are not stacked
- alarm(0) cancels alarm

# Summary

Important Issues to remember:

How to compare different policies?

What are the pros and cons of each scheduling policy?

What are signals and why are they important?

What does it mean to catch a signal?

What are the different ways to generate signals?