Welcome to Operating System Design (CS 423)

Wade Fagen-Ulmschneider
Spring 2021, University of Illinois

Slides built from Prof. Adam Bates and Prof. Tianyin Xu previous work on CS 423.

Course Overview

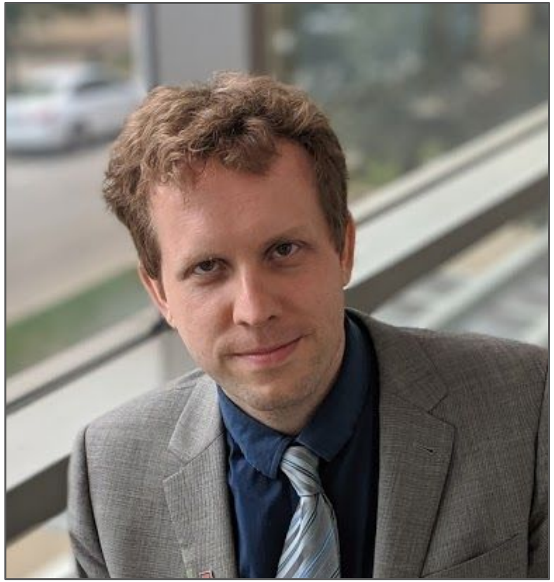
You Already Know:

- C Programming
- Basic Linux/POSIX APIs
- Basic Systems Primitives
  - Memory Allocation
  - Synchronization
  - Deadlock

After CS 423:

- Mastery of Operating System concepts
- Comprehensive understanding of virtualization techniques
- Introduction to Advanced OS topics:
  - Security
  - Power/Energy
  - Redundancy
- A kernel-level hacker, having established a kernel development environment and having modified OS code

Introductions:



Wade Fagen-Ulmschneider (waf)
Teaching Associate Prof. of Computer Science
Grainger College of Engineering

Introductions:

Why CS 423?

- ★ Understand the foundation of **all software systems**.
- ★ Apply the design of systems concepts to higher level software systems -- browsers, VMs, IoT devices, and more all use many ideas from OS design.
- ★ Acquire a very specific (and lucrative) set of skills!
  - Huge need for engineers who know OS/device drivers/kernel.
  - Increasingly few programs have a low-level systems course.

Prerequisites

- ★ We are writing kernel code, we are modifying Linux, and we're understanding every bit of how it works.

- ★ **Prerequisites**: Background in systems programming
  - ○ CS 241 or ECE 391

Textbook

- ★ **"Operating Systems: Three Easy Pieces"** by Ostep Remzi and Andrea Arpaci-Dusseau
  - Chapters available online for FREE!
  - Each lecture will have linked readings from the text.

- ★ *Additional, optional texts are listed on the syllabus.*

Course Structure

- ★ **Every Monday:**
  - All content (lectures, readings, MPs, etc) posted.
  - All due dates will be Mondays at 11:59pm Central Time.
- ★ **Every Tuesday at 2:00pm:**
  - Course meetup on Zoom; introduction to the week, discussions on news/innovation in systems; etc.
- ★ **Thursdays at 2:00pm:**
  - Usually office hours, except MP release weeks where TAs will hold an MP overview session.
- ★ **Fridays:**
  - All assignments turned in on Monday returned to you.

Assignments

- ★ **Machine Problems (MPs)**
  - MP0 “set-up” MP where you’ll get Linux compiled on your VM,
  - 4x multi-week MPs developing Linux kernel modules

- ★ **Exams**
  - *Midterm Exam: Thursday, March 18*
  - *Final Exam: Finals Week*
  - *Open-notes, closed-other people; full details in March*

- ★ **Occasional Homework and Participation**
  - Discussions on Piazza, practice final, etc

MPs

- You will implement and evaluate concepts from lecture within a real operating system (specifically, Ubuntu Linux).
  - Your code will play along with the 25,000,000 other lines of code that make up the Linux kernel.

- **Q:** *Why not make our own OS?*
  - Building a small OS is a good experience,
  - Extending a real OS is more practical and gets more done

MPs: Virtual Machines

- ★ You will be provided a VM managed by EngrIT for MP development.
  - If you brick your VM, you must open a ticket with EngrIT and they have to reset it. **This takes >24 hours!**
    - Bricked it on a weekend? You VM will be unavailable until Monday/Tuesday. :(
  - On a rare occasion, the whole VM Farm may go down. *Let's hope that doesn't happen this semester.*

MPs: Virtual Machines

- ★ Extensions for VM failures will only be given for cloud-wide failures or other extraordinary circumstances., **NOT** for self-inflicted issues!

- ★ **Strategies to ensure success:**
  - **Develop on your own VM**, using VirtualBox or other free VM tools.
    - As part of MP0, we will give you the exact VM setup!
    - However, we grade on the EngrIT VM, so make sure to deploy it to your VM before the deadline + commit it to git.

  - **Commit your code often**; if you're changing code on the VM, and brick it, all your code will be lost.

git

- ★ We will use the EngrIT-hosted GitHub Enterprise server: https://github-dev.cs.illinois.edu/

- ★ A microservice will create the repo for you. We will grade your MP is one of two ways:
  - On some MPs, we will **log into your VM** and ensure your VM has the MP integrated into your Linux.
  - On other MPs, we will **compile your source** and grade it on a new EngrIT VM.
  - Therefore, you must both run **your code on your VM and commit your code via git**.

4CR Section

- ★ Graduate students and those interested systems research can take this course for an addition credit hour.

- ★ **Requirement:** Two papers will be posted each week. You will:
  - Look over both of them,
  - Choose one to read in-depth and summarize,

- ★ **4CR Grade:** 80%*(3CR) + 20%*(Summaries) = Final Grade

4CR Summaries

- Each summary should be **1-2 pages** in length, discussing the paper in depth including:
  - **Why** you choose the paper you did (between the two),
  - The **area** of systems the paper addresses,
  - The **problem** the paper addresses,
  - The **solution** the paper presents,
  - The **methodology** the paper uses,
  - The **results** reported by the paper,
  - What did you **take away** from the paper?

Course Policies

- **No late submissions** without prior approval:
  - If you're falling behind, better to just move on and keep up with the course. We move fast!

- **One-week regrade window**:
  - What you submit on Mondays will be graded by Friday. You have until the next Friday to bring to our attention any errors.
  - If you discover an error in any automated grading (ex: autograder), we will update the grader and re-run it on everyone to ensure everyone benefits.

- **All assignments are individual.**

Course Policies

- ★ **Zero tolerance on cheating:**
  - **<u>Simple: Don’t do it.</u>**
  - **First Offense:**
    - Zero on the assignment,
    - -100 points to your course grade, **and**
    - Forfeit all extra credit for the course.
  - **Second Offense:**
    - -1000 points to your course grade (automatic “F”)
  - We consider **each instance of cheating its own offense**, even if discovered at the same time. *(Ex: Cheated on MP2+MP3, discovered after MP3 ⇒ F in course.)*

Feedback Welcome!

- ★ **This is my time with CS 423:**
  - We're on a team together to master Operating Systems.
  - I will likely screw up a few things.
  - Feedback is always welcome, and I'll actively seek it throughout the semester. :)

Everything Else:

https://courses.grainger.illinois.edu/cs423/sp2021/


ALMA MAT

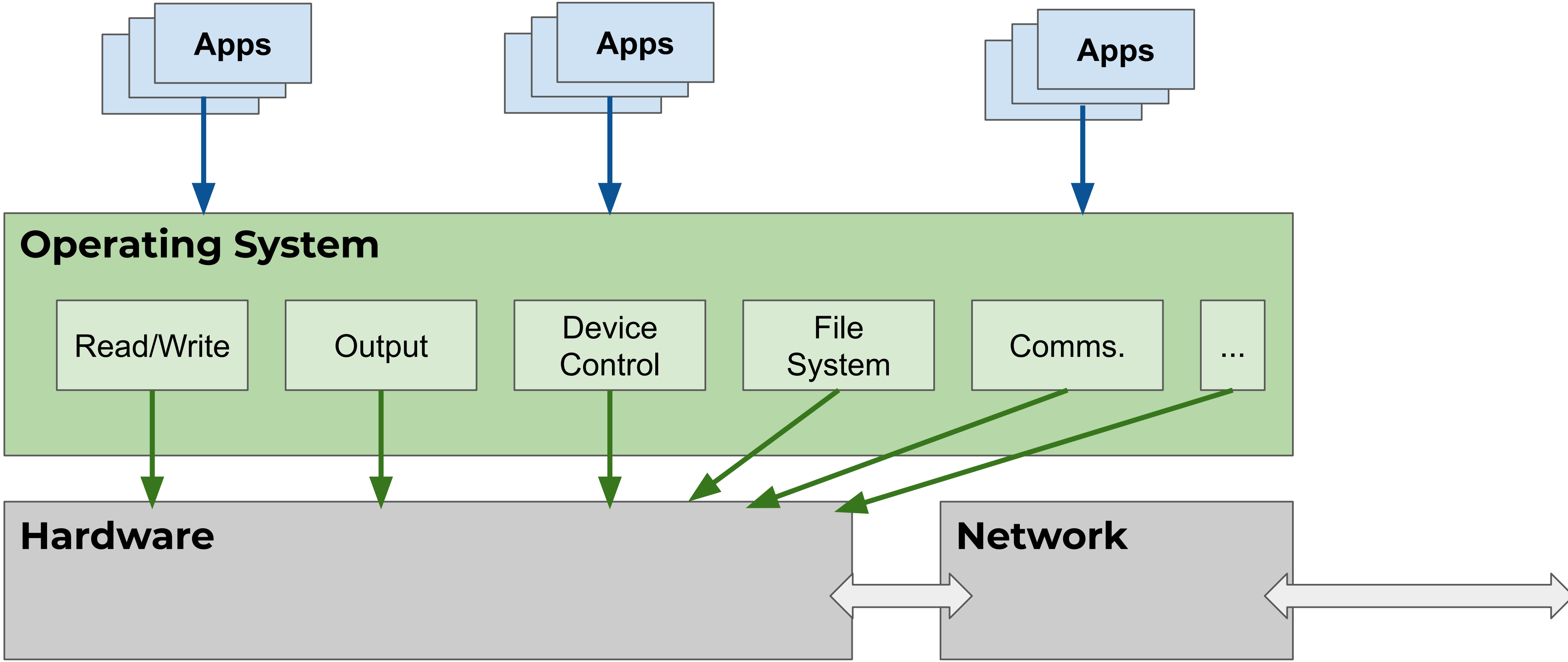
Overview of an Operating System

CS 423 - University of Illinois

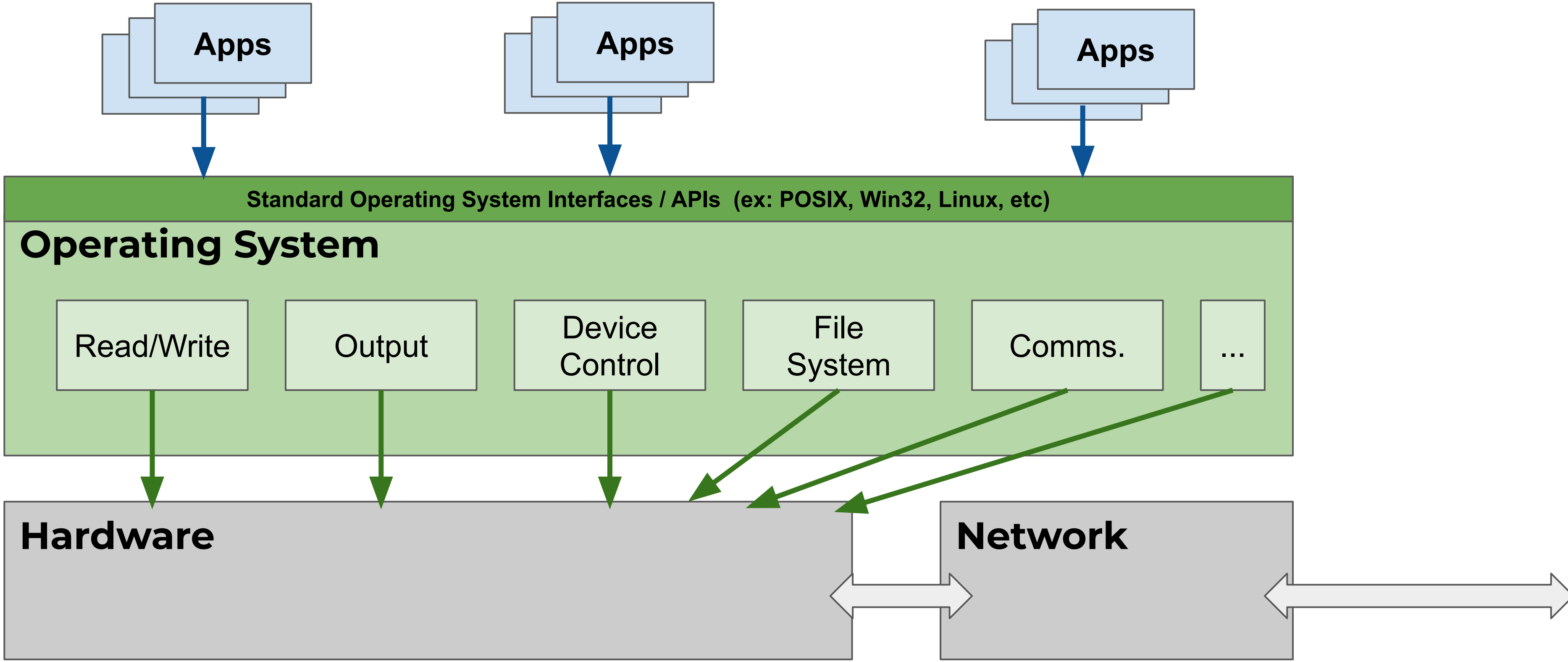
Wade Fagen-Ulmschneider
(Slides built from Adam Bates and Tianyin Xu previous work on CS 423.)

Operating System Overview:

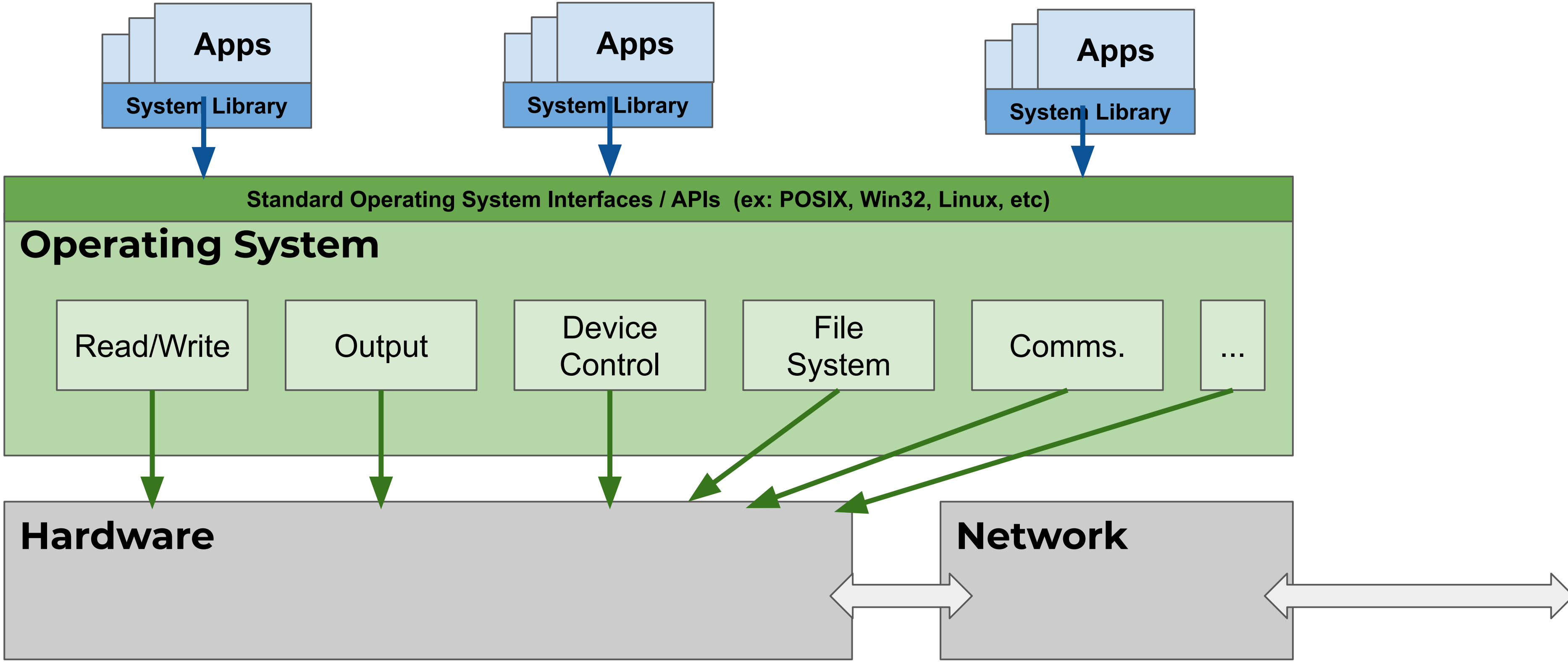
★ Software to **manage a computer's resources** for its users.



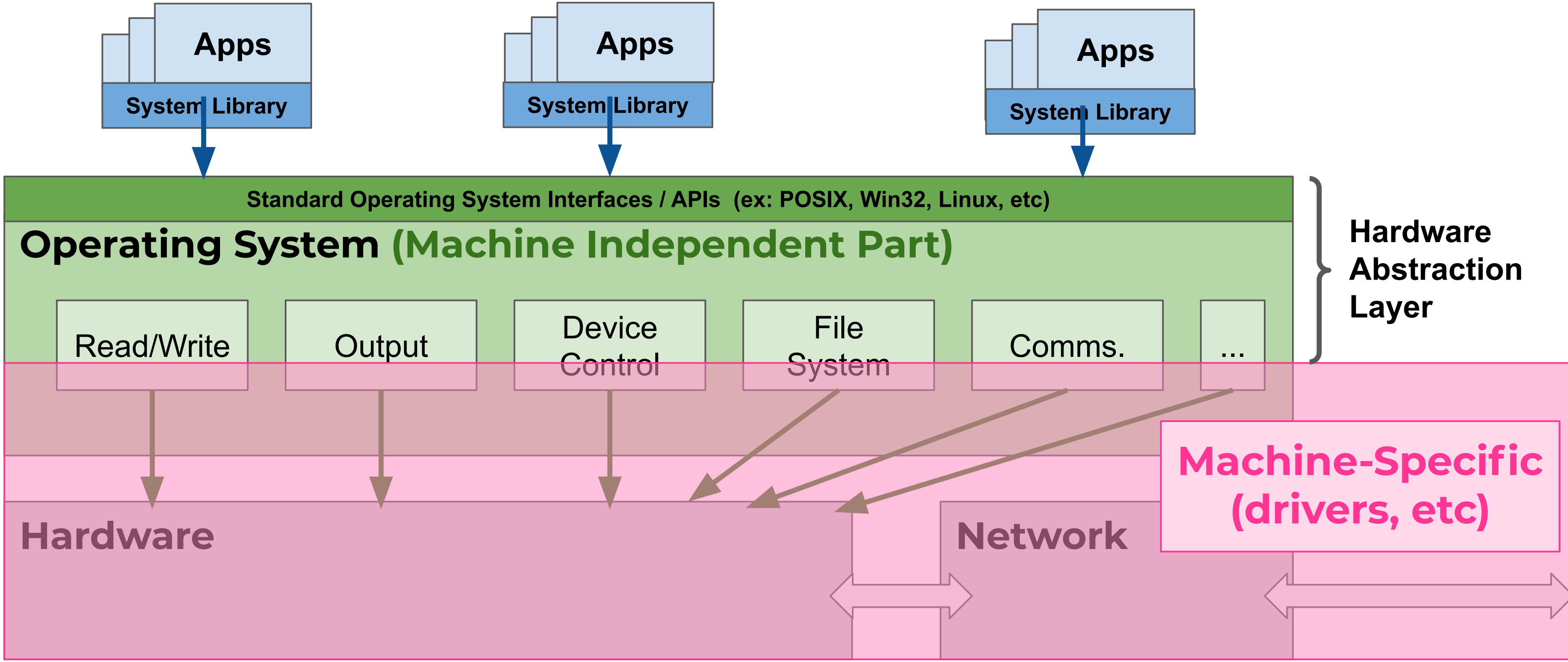
★ The **OS exports an interface (API)** for apps to use:



★ Apps are compiled with a **system-specific library** to interface w/ OS:



OS provides a common **"Hardware Abstraction Layer"** to machine-specific hardware:



Operating System Responsibilities

- **Role #1: Referee**
  - **Manage** resource allocation between many users and processes.
  - **Isolate** users and processes from each other.
  - **Facilitate** communication between isolated users and processes.

Operating System Responsibilities

- **Role #2: Illusionist**
  - Allow each user and process to believe it has the entire machine to itself.
  - Create the appearance of (near)-infinite memory, available processes, etc...
  - Abstract away complexity of reliability, networking, storage, etc...

Operating System Responsibilities

- **Role #3: Glue**
  - Manage hardware to be machine-agnostic.
  - Provide common services that are shared among applications and users.
    - **"Glue" Services:** Copy/Paste, User Interfaces, File I/O

Operating System Responsibilities: Example

- ★ **Consider the file system in an OS:**

Operating System Responsibilities: Example

- **Consider the file system in an OS:**
  - **Referee:**
    - Prevent others from accessing the file without permissions.
    - Re-use storage space after a file is deleted.
  - **Illusionist:**
    - Files grow/shrink with easy to an (nearly) infinite size.
    - Files persist even during certain hardware faults.
  - **Glue:**
    - Directories
    - Standard API for file I/O

Operating System Needs Are Changing

Operating System Needs Are Changing

- ★ **Network Speeds:**
  - 1980 ⇒ **300 bps** / $
  - 2000 ⇒ **~256 Kbps** / $
  - 2020 ⇒ **~20 Mbps** / $

- ★ In the past 40 years, the speed of home networking has creased by a factor of **~67,000x**.

Operating System Needs Are Changing

- ★ **Number of Cores /CPU:**
  - 1980 ⇒ **1** core / CPU
  - 2000 ⇒ **1** core / CPU
  - 2020 ⇒ **8+** cores / CPU and **64+** cores /server CPUs
- ★ In the past 20 years, the number of available cores have exploded.

Operating System Needs Are Changing

- **Cost per megaflop/sec:**
  - 1980 ⇒ ~**$100,000** / megaflop/sec
  - 2000 ⇒ ~**$25** / megaflop/sec
  - 2020 ⇒ ~**$0.20** / megaflop/sec

- In the past 40 years, the cost per million operations has decreased by a factor of **~500,000x**.

Operating System Needs Are Changing

- **RAM Capacity B/$:**
  - 1980 ⇒ **~2 KiB** / $
  - 2000 ⇒ **~2 MiB** / $
  - 2020 ⇒ **~2 GiB** / $

- In the past 40 years, the cost per byte of RAM has decreased by a factor **~1,000,000x**.

Operating System Needs Are Changing

- **Storage (HDD) Capacity B/$:**
  - 1980 ⇒ **~3 KiB** / $
  - 2000 ⇒ **~7 MiB** / $
  - 2020 ⇒ **~25 GiB** / $

- In the past 40 years, the cost per byte of storage has decreased by a factor **~10,000,000x**.

Operating System Needs Are Changing

- ★ **Network Speeds:**
  - 1980 ⇒ **300 bps** / $
  - 2000 ⇒ **~256 Kbps** / $
  - 2020 ⇒ **~20 Mbps** / $

- ★ In the past 40 years, the speed of home networking has creased by a factor of **~67,000x**.

Operating System Needs Are Changing

- ★ **Ratio of Computers to Users**
  - 1980 ⇒ **100 users : 1 computer**
  - 2000 ⇒ **1 user : 1 computer**
  - 2020 ⇒ **1 user : many computers**

- ★ In the past 40 years, the number of users to computers has increased by a factor of at least **200x+**.

Operating System Challenges

Operating System Challenges

- ★ Reliability
- ★ Availability
- ★ Security
- ★ Privacy
- ★ Portability
- ★ Performance

Examples of Function and System Calls

Legacy Needs:

- Runs one application at a time.

- Manage “time quotas” for the many users.

- Users submit jobs and wait for results days later.

Modern Needs:

Future Needs:

Examples of Function and System Calls

Legacy Needs:

- Runs one application at a time.

- Manage "time quotas" for the many users.

- Users submit jobs and wait for results days later.

Modern Needs:

- Multiprogramming across many cores and many concurrent users.

- Interactive, completing all jobs as quickly as possible.

- Optimize for user's time, not for computer's resource time.

Future Needs:

Examples of Function and System Calls

Legacy Needs:

- Runs one application at a time.

- Manage "time quotas" for the many users.

- Users submit jobs and wait for results days later.

Modern Needs:

- Multiprogramming across many cores and many concurrent users.

- Interactive, completing all jobs as quickly as possible.

- Optimize for user's time, not for computer's resource time.

Future Needs:

- Manager and use an ever-increasing number of processors /computer.

- Peta-scale storage, data-centers, etc

- Optimize for seamless interaction between operating systems on different computers.
(Users use many computers.)

Review: System Calls

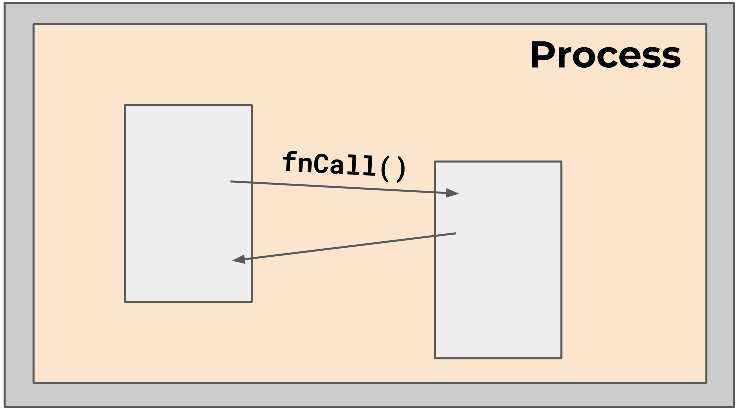
CS 423 - University of Illinois

Wade Fagen-Ulmschneider

(Slides built from Adam Bates and Tianyin Xu previous work on CS 423.)

Function Calls and System Calls

Function Call:

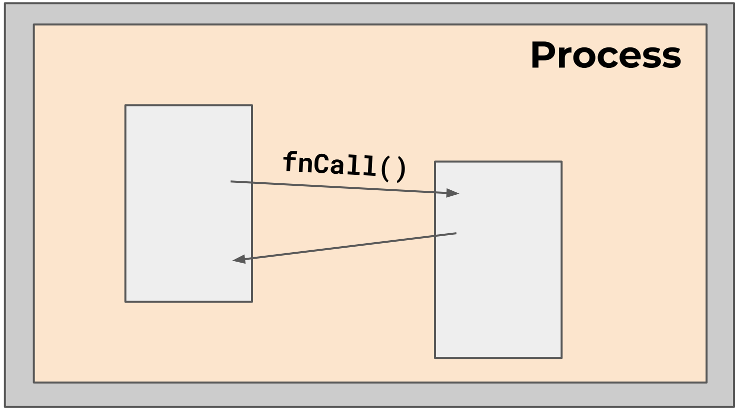


- ★ Caller and callee in the ________________.
  - ○ Same user
  - ○ Same "domain of trust"

System Call

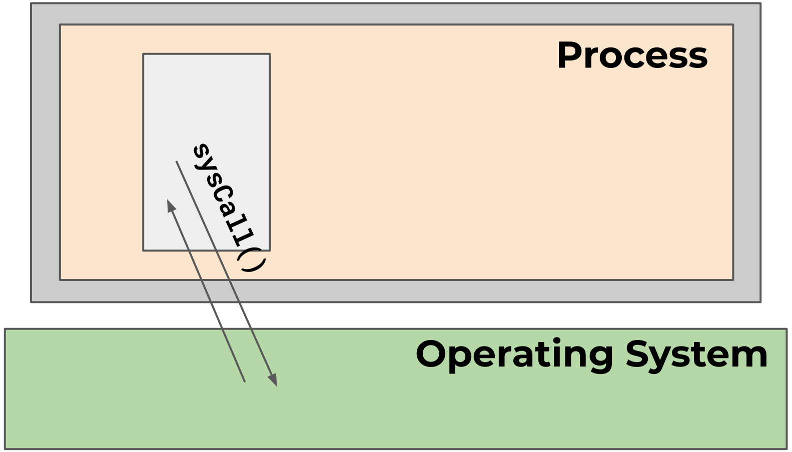
Function Calls and System Calls

Function Call:



- Caller and callee in the **same process**.
  - Same user
  - Same “domain of trust”

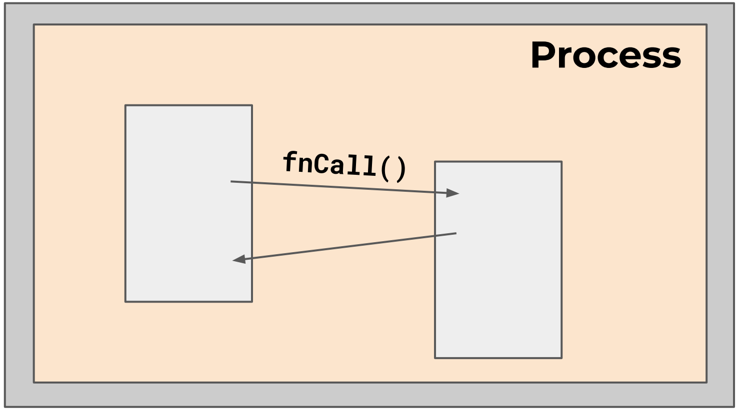
System Call



- OS is trusted; user process is not.
- OS code runs privileged with complete access to all system resources.
  - **Must prevent abuse.**

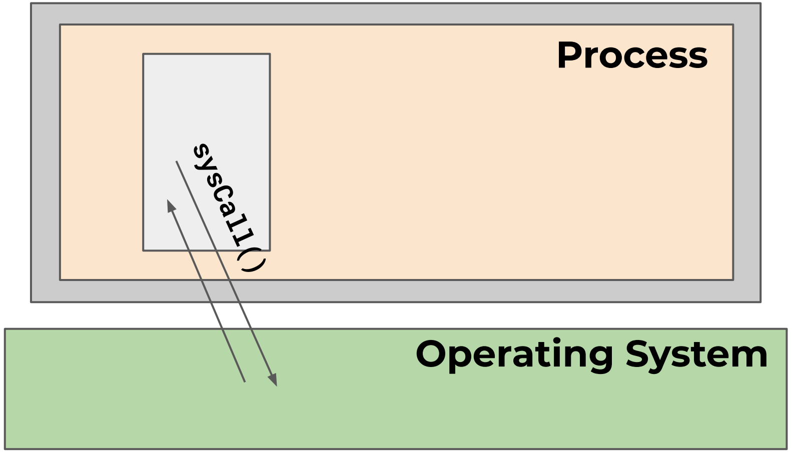
Function Calls and System Calls

Function Call:



- ★ Caller and callee in the **same process**.
  - ○ Same user
  - ○ Same "domain of trust"

System Call



- ★ OS is ___________; user process is ______.
- ★ OS code runs privileged with complete access to all system resources.
  - ○ _____________________________.

Examples of Function and System Calls

C Library Call:

Linux System Call:

Win32 System Call:

Examples of Function and System Calls

C Library Call:	Linux System Call:	Win32 System Call:
`fopen`	`open`	`CreateFileA`
`fclose`	`close`	`CloseHandle`
`getc/putc`	`read/write`	`ReadFile / WriteFile`
`fread/fwrite`		
`scanf/printf`		
`fprintf`		
`fseek`	`lseek`	`SetFilePointer`
`rand`		

Python Code:

Python: `open(...)`

Python is written in C ("CPython"), making a call to the C library calls…

C++: `fopen(...)`

When compiled on Win32 system...

When compiled on a Linux system….

SysCall: `open(...)`

SysCall: `CreateFileA(...)`

CS 423 will by POSIX-focused

- We will focus on the **Linux/POSIX** system/standard.
  - Other systems are very similar.
  - Virtualization and containerization has also made the universe smaller (ex: Windows Subsystem for Linux, etc).

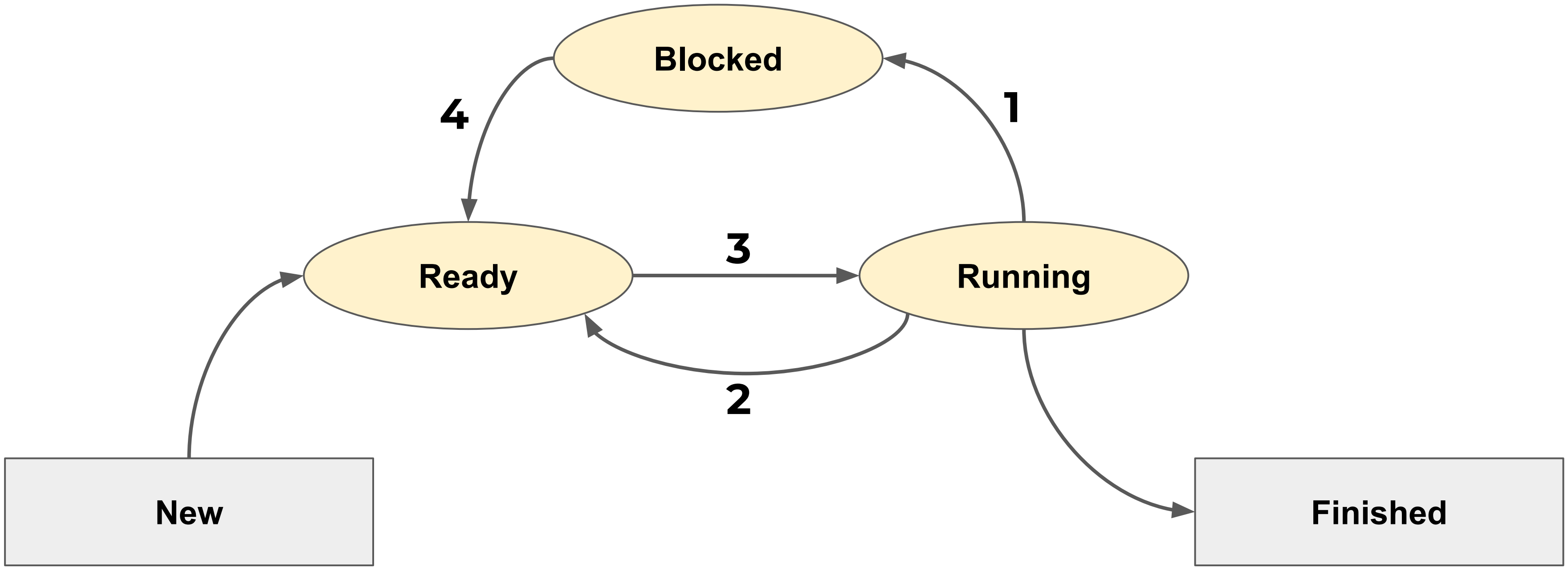
Review: Processes

CS 423 - University of Illinois

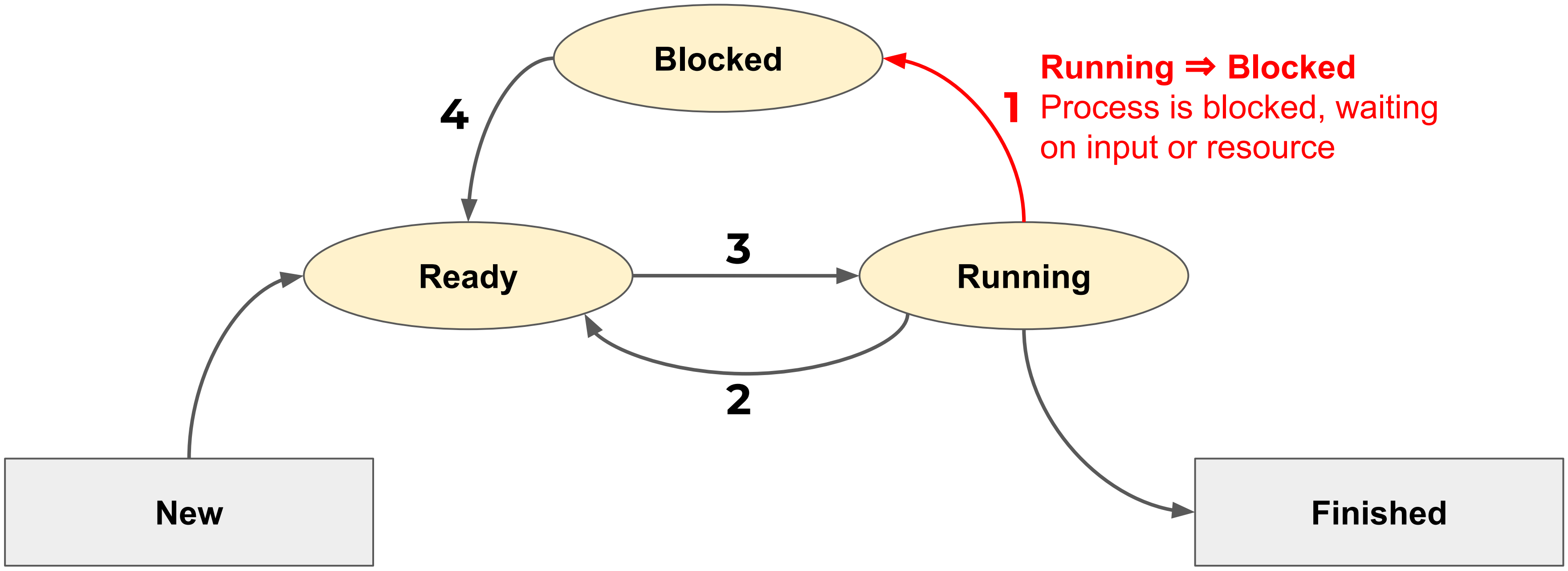
Wade Fagen-Ulmschneider

(Slides built from Adam Bates and Tianyin Xu previous work on CS 423.)

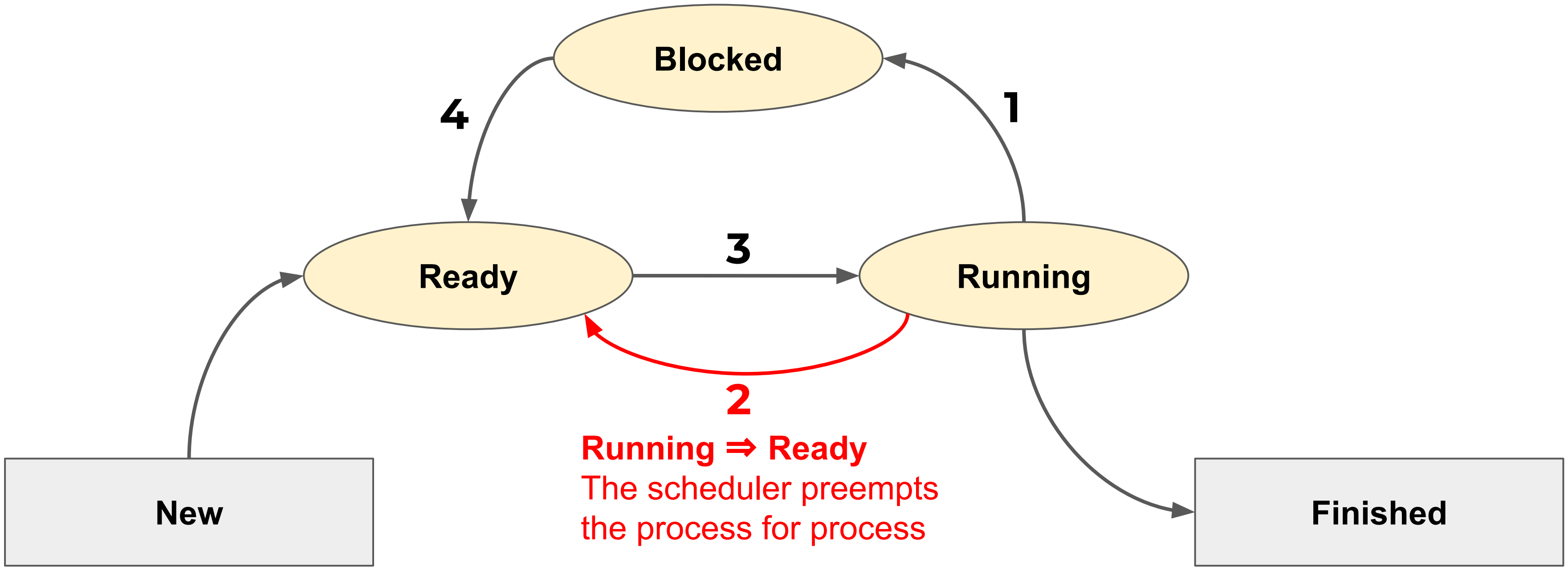
Five State Model for Processes



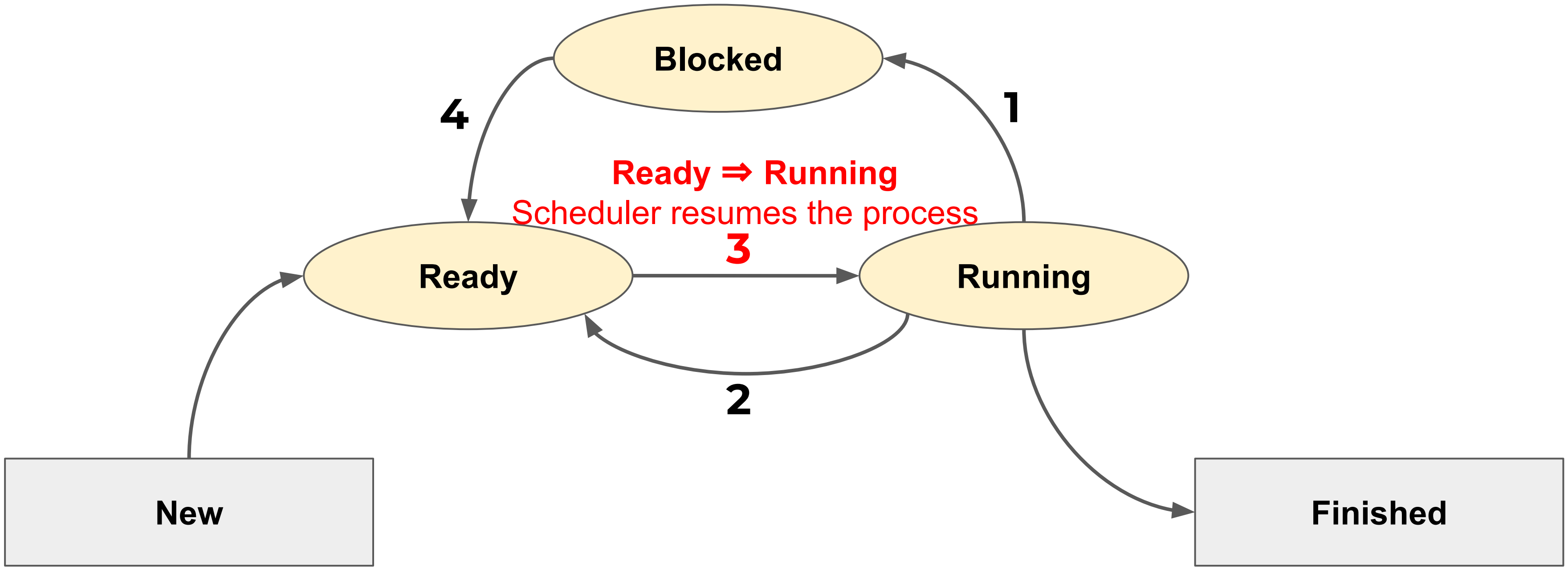
Process State Transitions



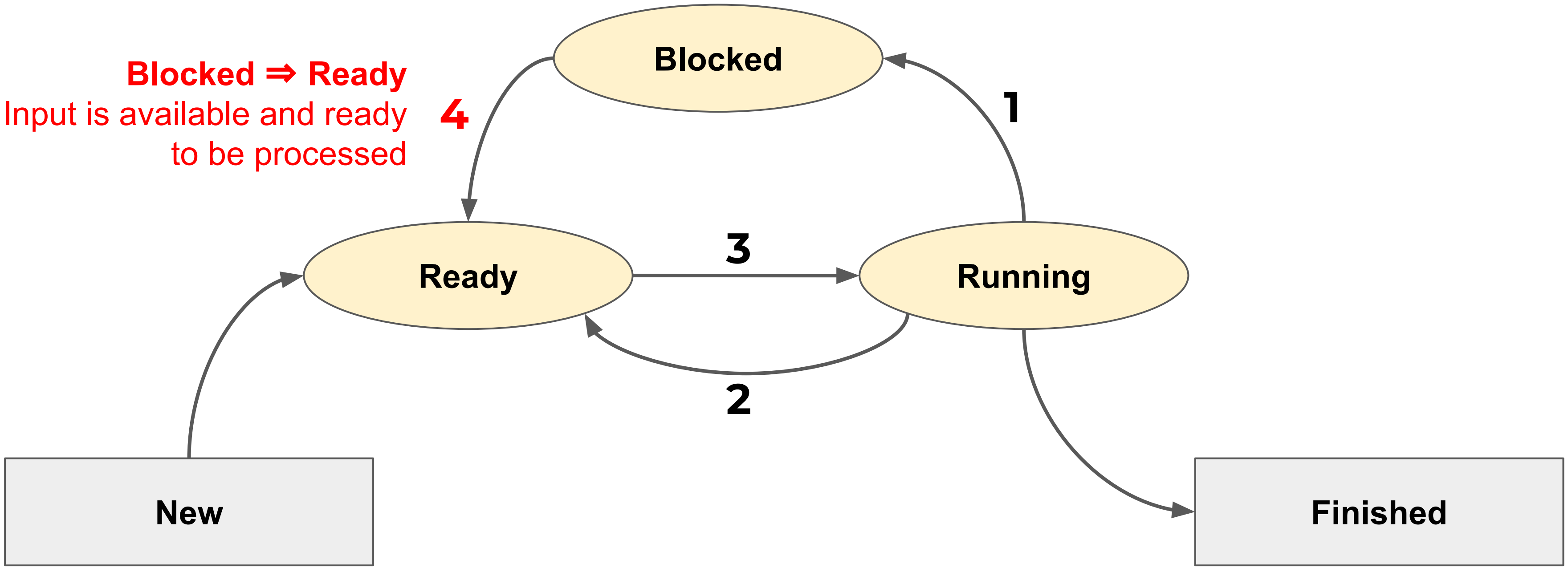
Process State Transitions



Process State Transitions



Process State Transitions

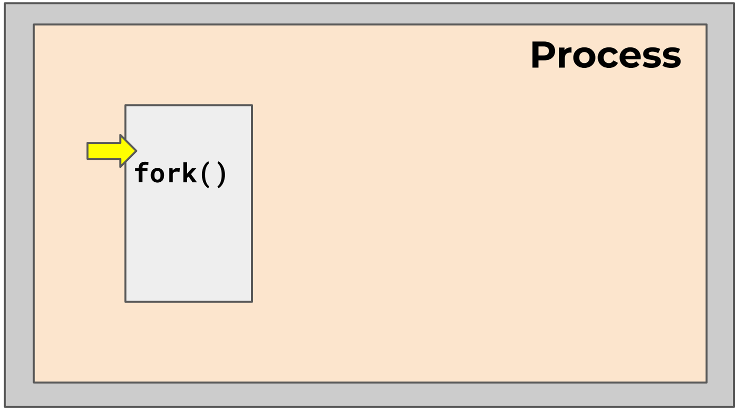


Creating a Process

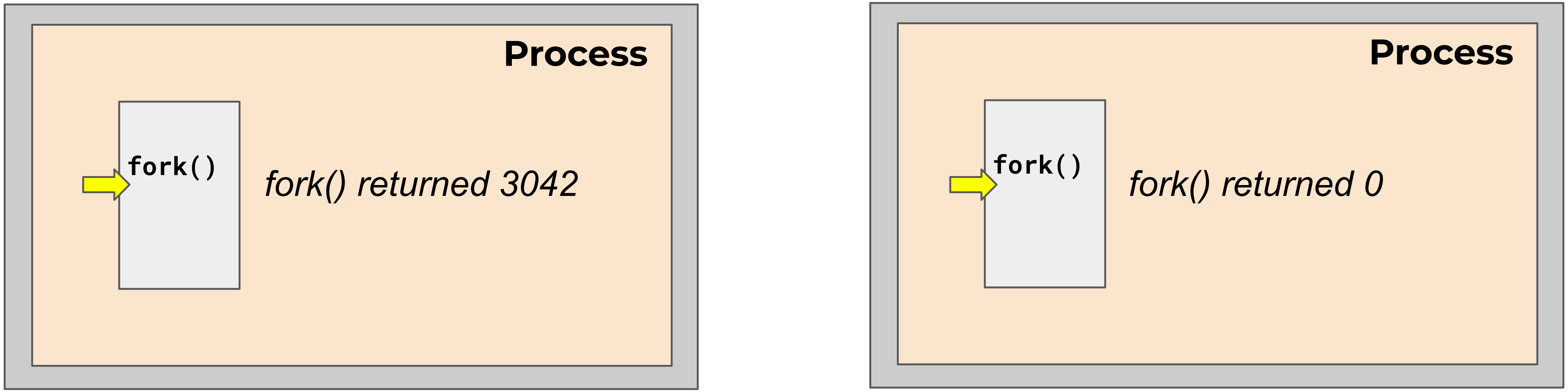
- All processes are created using the **fork** system call:
  - Creates an **exact copy** of the current process.

  - Both processes continue in parallel from the statement that follows the **fork** call.
  - Only difference is the return value:
    - **Parent:** Child Process ID (“pid”, non-zero)
    - **Child**: 0
      - Child can get parent ID via getppid()
    - **Failure:** -1

Creating a Process



Creating a Process



Executing a New Program

- A common use of fork is to launch a new executable program.

- The **exec** system call replaces the current process image with a new image.
  - *If exec succeeds, it never returns.*

- **exec** requires you to specify the file you program to run.


ALMA

Review: Threads

CS 423 - University of Illinois

Wade Fagen-Ulmschneider

(Slides built from Adam Bates and Tianyin Xu previous work on CS 423.)

Threads

★ In the most general terms, threads require only two things to be true:

1. **Independent execution sequence**, and

2. **Shared memory space** with other threads in the same process

User Threads vs. Kernel Threads

★ Threads can be scheduled by a process ("user-thread") or by the kernel. *(Both are useful!)*

User Threads vs. Kernel Threads

User Threads:

- ★ **Shared memory** within a process
- ★ **Separate execution** sequence

- ★ **Fast context switching**
- ★ **User-defined scheduling**

Kernel Thread:

- ★ **Shared memory** within a process
- ★ **Separate execution** sequence

- ★ **Each thread can make blocking calls**
- ★ **Can run concurrently on multiple CPUs**

POSIX Threads

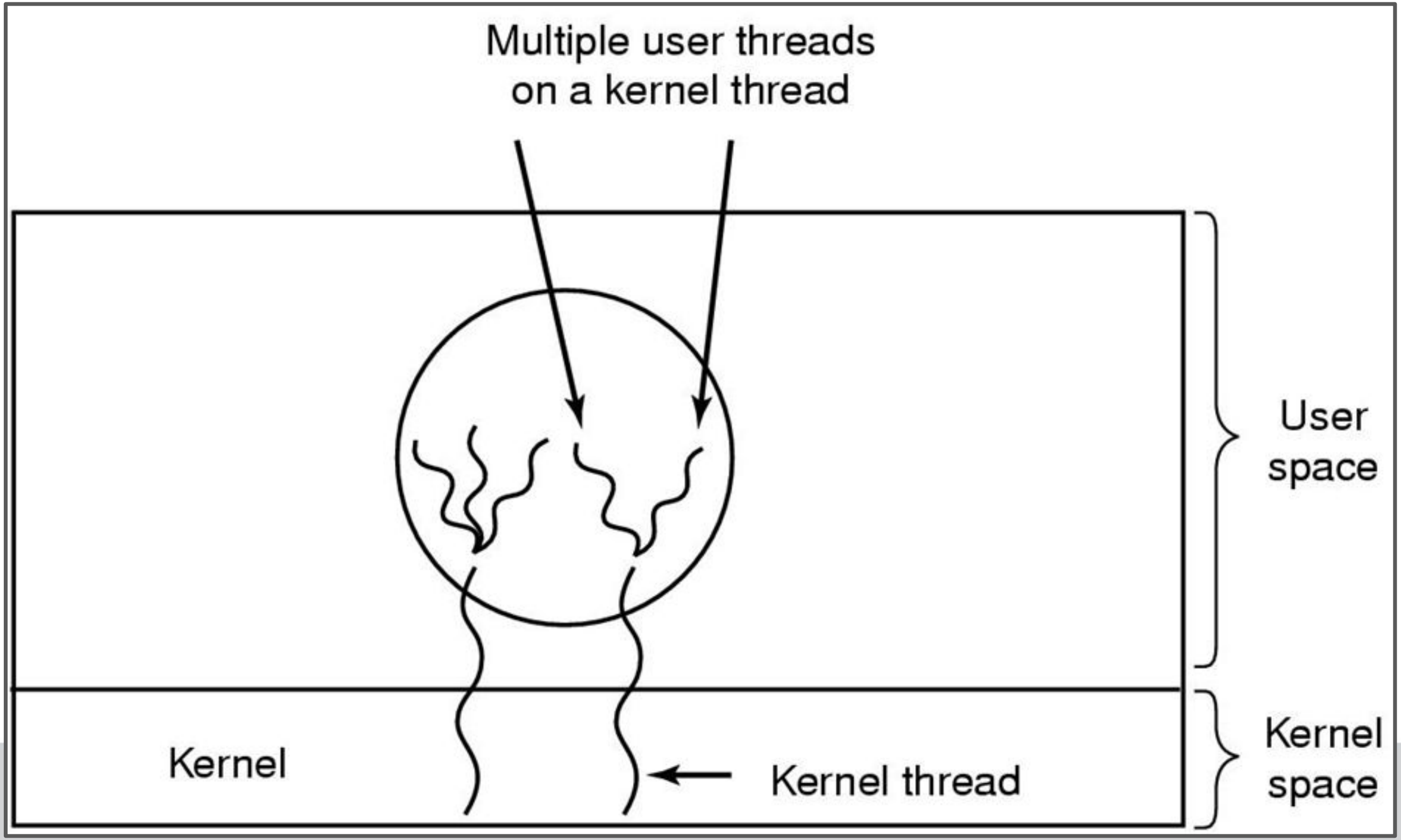
POSIX

- “The Portable Operating System Interface (POSIX) is a **family of standards** specified by the IEEE Computer Society for maintaining compatibility between operating systems. **POSIX defines the application programming interface (API)**, along with command line shells and utility interfaces, for software compatibility with variants of Unix and other operating systems.”

POSIX Threads

- A POSIX thread is a created with the POSIX call **`pthread_create()`**.
  - Since 2003 (kernel 2.6), Linux implements POSIX threads as kernel-scheduled threads.
    - *See: Native POSIX Thread Library*

Hybrid Threads (N:M, Solaris Threads)



Hybrid Threads (N:M, Solaris Threads)

- ★ M:N was once thought to provide better performance, but:
  - HARD to implement
  - Now need two layers of blocking, one for user space threads and another for the kernel space thread
  - Multicore processors bring more performance for more kernel threads

Review: Synchronization

CS 423 - University of Illinois

Wade Fagen-Ulmschneider

(Slides built from Adam Bates and Tianyin Xu previous work on CS 423.)

Motivation

★ Processes and threads can be preempted at any time and can generate problems:

Thread #1:	**Thread #2:**

`read X`
`add 1 to X`
`write X`
	`read X`
	`add 1 to X`
	`write X`

X *is a shared variable*

Mutex

Mutex

- ★ Simplest and most efficient thread synchronization mechanism
- ★ A special variable that can be either in
  - **locked state**: a distinguished thread that holds or owns the mutex; or
  - **unlocked state**: no thread holds the mutex
- ★ When several threads compete for a mutex, the losers block at that call
  - The mutex also has a queue of threads that are waiting to hold the mutex.
- ★ POSIX does not require that this queue be accessed FIFO.
- ★ *Helpful note — Mutex is short for "Mutual Exclusion"*

Mutex

```
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
                const pthread_mutexattr_t *restrict attr);
   (Or: PTHREAD_MUTEX_INITIALIZER)
int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Counting Semaphore

★ Allows for an arbitrary number of consumers to use a resource simultaneously.

sem_wait

```
if (sp->value == 0) {
   // Add thread to sp->blockList
   // Block thread
}
sp->value--;
```

sem_singal

```
sp->value++;
if (sp->list != NULL) {
   // Unblock thread on sp->blockList
}
```


ALMA

Review: Signals

CS 423 - University of Illinois

Wade Fagen-Ulmschneider

(Slides built from Adam Bates and Tianyin Xu previous work on CS 423.)

Signals

- Signals are a simple way for **one process to send a notification to another.**

- Signals must be handled in one of three ways:
  - Signals can be **caught**,
  - Signals can be **ignored**, or
  - Signals can be **blocked**.

  - *All signals have a **default action** defined by the system.*

Signals

- A signal is **generated** when the event that causes it occurs.
- Signal is **delivered** when a process receives it.
  - The **lifetime** of a signal is the interval between its generation and delivery.
  - A signal is **pending** when it has been generated but not delivered.

- The process can:
  - **Catch** the signal by **executing a signal handler** when signal is delivered.
  - **Ignore** a signal when it is delivered, results in the **default signal action**.
  - **Block** the by adding the signal to the signal mask.

- *The "signal mask" contains the set of signals currently blocked.*

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**

Deliver Signals

★ The linux utility **`kill`** allows us to deliver signals to a process:

- **`kill -l`**, lists all signals available
- **`kill [signal=SIGTERM] pid`**, sends the **signal** to **pid**

- **`kill -9 pid`**, send a **SIGKILL** to **pid**. (Terminates the process.)
  - ***-9*** *is shorthand for* ***-SIGKILL***

Signal Masks

- A process can temporarily prevent a signal from being delivered by blocking it.

- Signal mask contains a set of signals currently blocked.
  - ***Blocking a signal is different from ignoring signal.***

Signal Masks

- Signal mask contains a set of signals currently blocked.
  - ***Blocking a signal is different from ignoring signal.***

- When a process blocks a signal, the **OS does not deliver signal until the process unblocks the signal**.
  - A blocked signal is not delivered to a process until it is unblocked.
- When a process ignores signal, signal is delivered and the process handles it by throwing it away.

Review: Deadlock

CS 423 - University of Illinois

Wade Fagen-Ulmschneider
(Slides built from Adam Bates and Tianyin Xu previous work on CS 423.)

Deadlock

- ★ Four necessary conditions for deadlock:

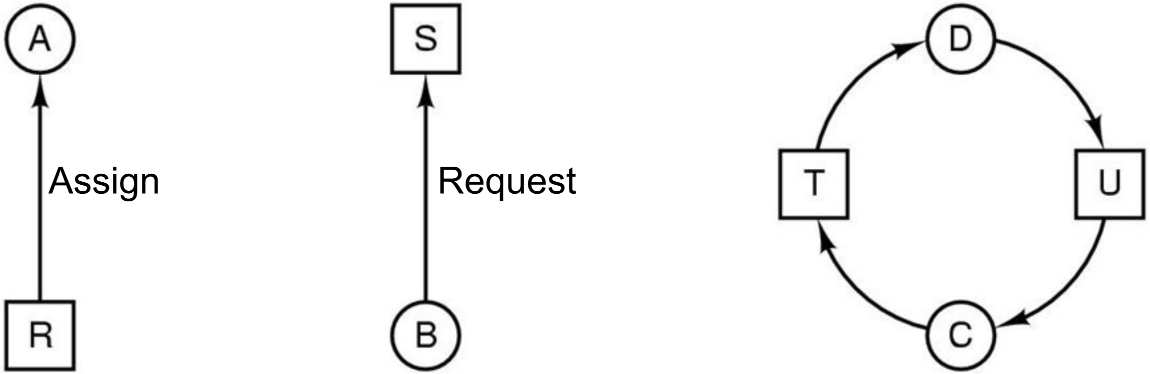
Deadlock

★ Four necessary conditions for deadlock:

1. Mutual exclusion
2. Hold and wait condition
3. No preemption condition
4. Circular wait condition

Deadlock Detection

★ Resource Allocation Graphs:



Resolving Deadlock

- ★ Detection and Recovery
- ★ Dynamic Avoidance (run-time)
- ★ Prevention (design-time)
    - *Eliminate* ***any one*** *of the four conditions*