CS 31: Introduction to Computer Systems

Last Class: Synchronization / Course Summary

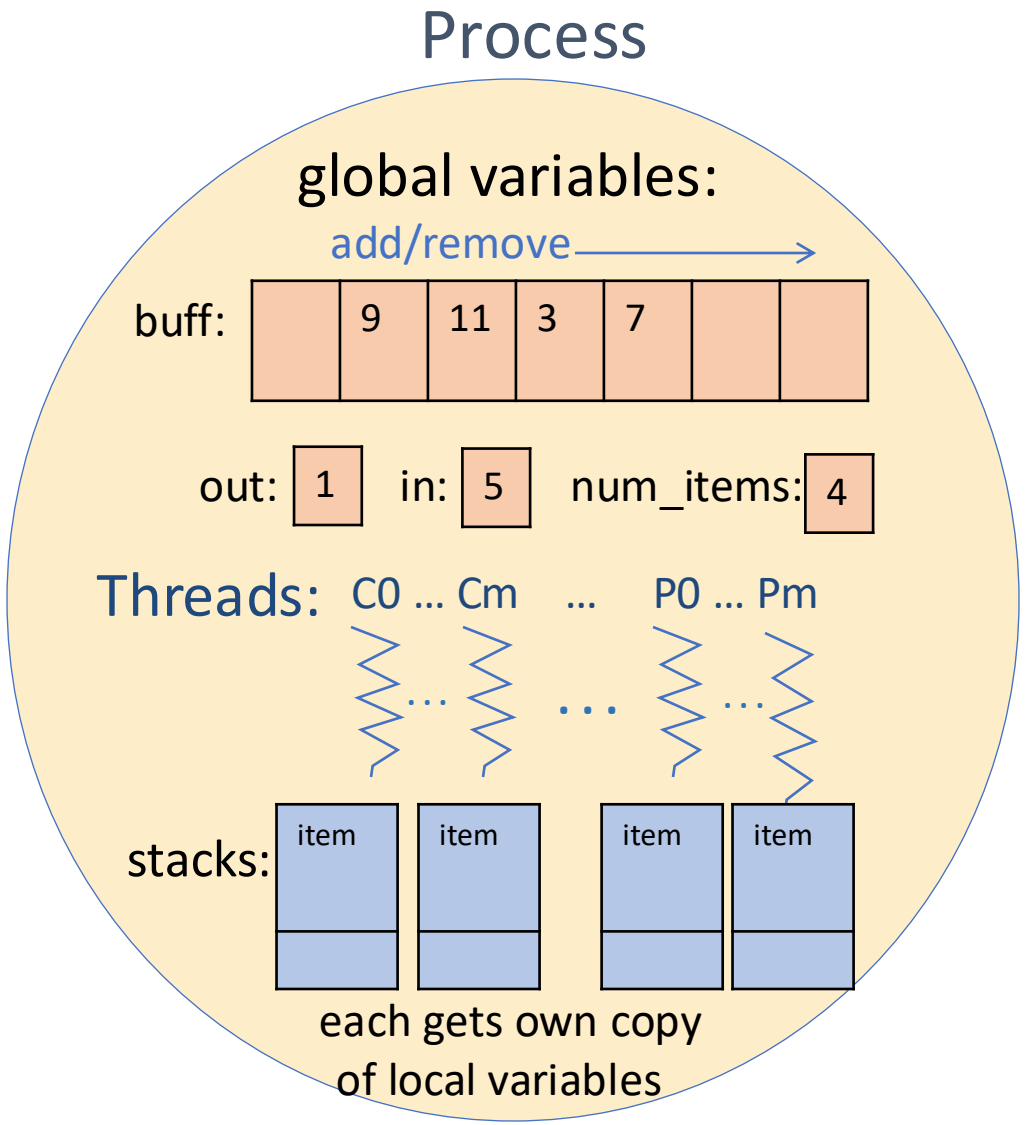
04-29-2025

Producer/Consumer Problem

- A shared fix-sized buffer
- Two types of threads:
  1. Producer: creates an item, adds it to the buffer
  2. Consumer: removes an item from buffer, and consumes it.

All kinds of real-world examples:

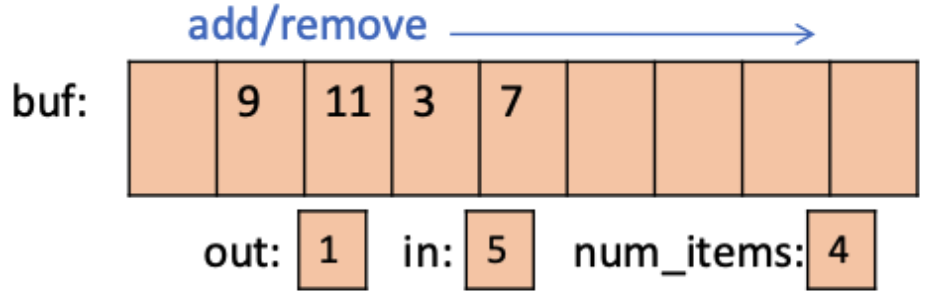
- print queue: users produce print requests, printer is consumer
- music streaming: server reads music from disk, sends data to client, client downloads music file from shared buffer
- web server: server reads file from disk, sends it on the network, client downloads file



Producer/Consumer Synchronization?

Circular Queue Buffer: add to one end (in), remove from other (out)

```
int buf[N];
int in, out;
int num_items;
```



Assume Producers & Consumers forever produce & consume

Q: Where is **Atomic** and **Scheduling** synchronization needed?

Producer:

Consumer:

Producer/Consumer Synchronization?

Producer:	Consumer:

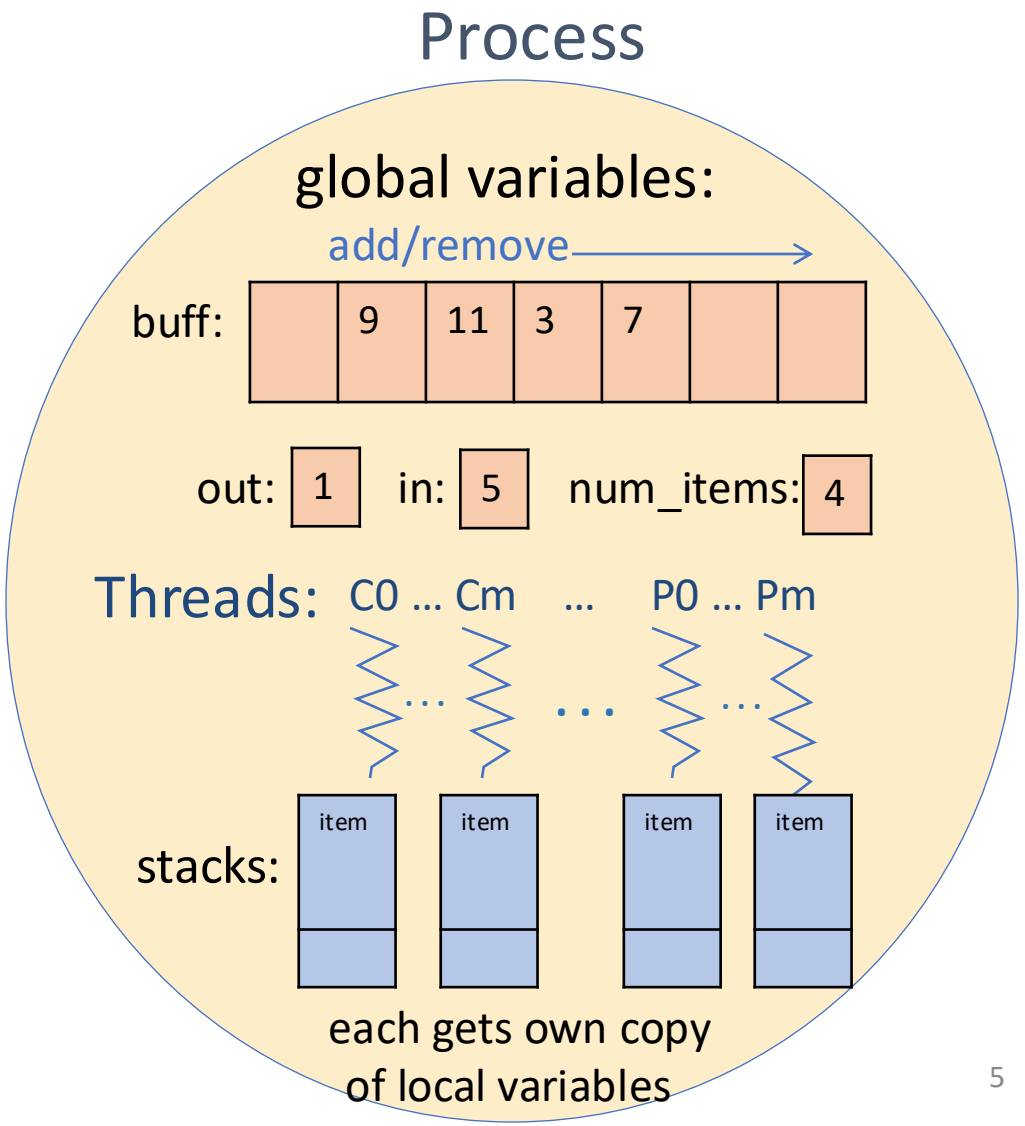
• Needs to wait if there is no space to put a new item in the buffer (Scheduling)
• Needs to wait to have mutually exclusive access to shared state associated with the buffer (Atomic):
  • Size of the buffer (num_items)
  • Next spot to insert into (in)
	• Needs to wait if there is nothing in the buffer to consume (Scheduling)
	• Needs to wait to have mutually exclusive access to shared state associated with the buffer (Atomic):
	  • Size of the buffer (num_items)
	  • Next spot to remove from (out)

Assume: Producer & Consumer thread execute forever

Circular Queue Buffer: add to one end, remove from another

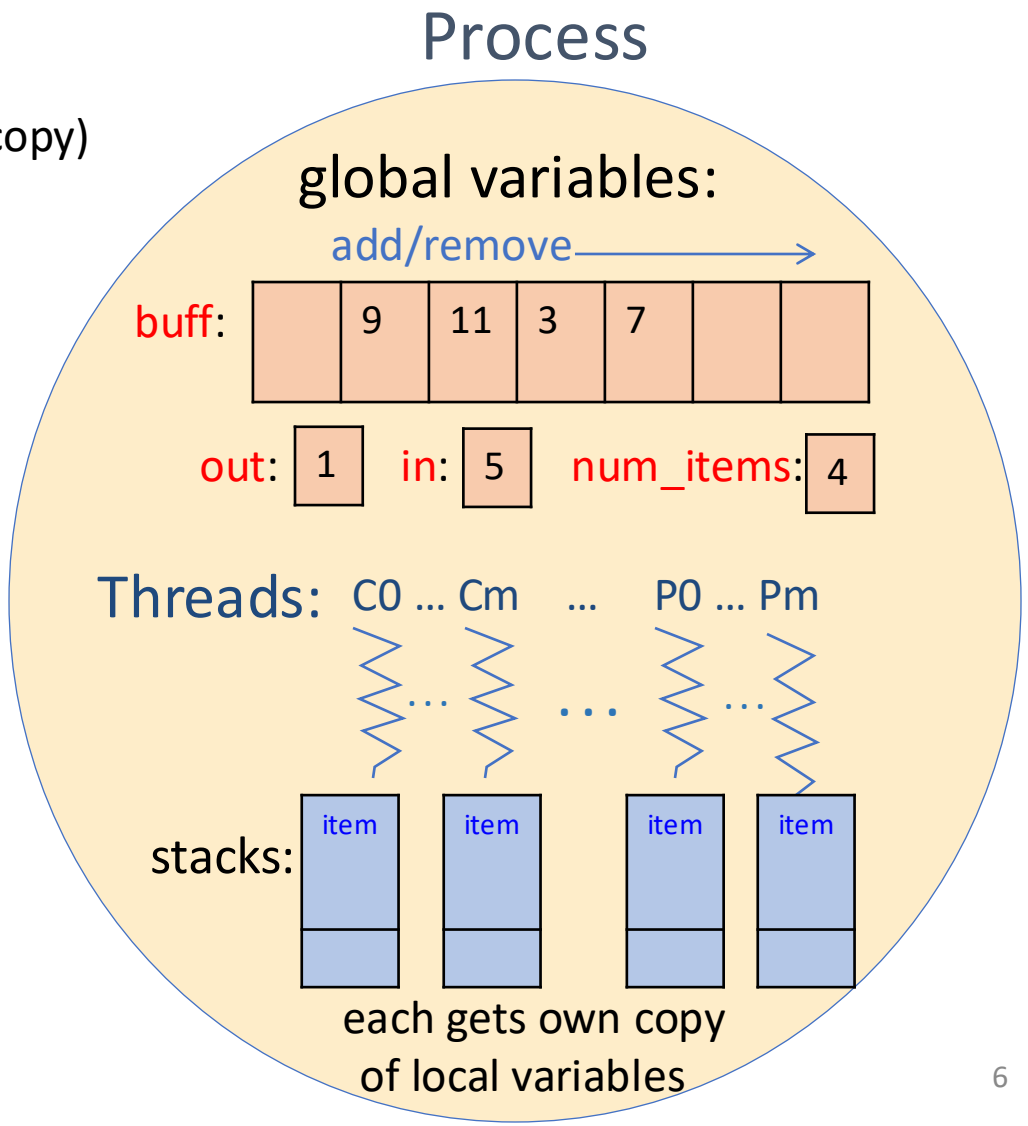
Producer/Consumer

```
int  num_items=0, in=0, out=0, buff[N];

void producer_threads(){
  int item;
  while(1) {
    item = produce_item();
    //add to queue
    buff[in] = item;
    in = (in+1)%N;
    num_items++;
  }
}
void consumer_threads(){
  int item;
  while(1) {
    //remove from queue
    item = buff[out];
    out = (out+1)%N;
    num_items--;

    consume_item(item);
  }
}
```

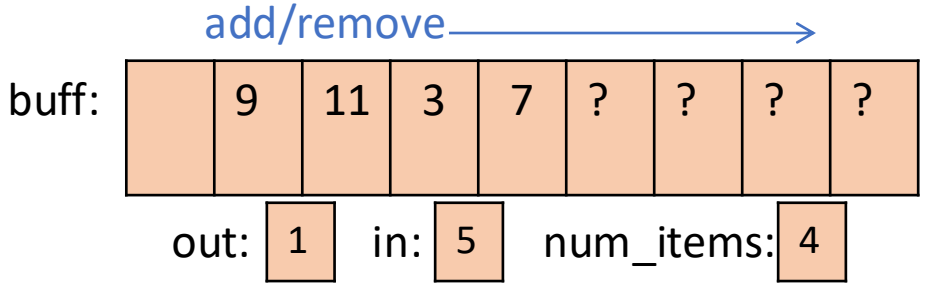


Producer/Consumer

```
// Global Variables (all threads can access):
int  num_items=0, in=0, out=0, buff[N];

void producer_threads(){
  int item; // local variable (own copy)
  while(1) {
    item = produce_item();
    buff[in] = item;
    in = (in+1)%N;
    num_items++;
  }
}
void consumer_threads(){
  int item;
  while(1) {
    item = buff[out];
    out = (out+1)%N;
    num_items--;
    consume_item(item);
  }
}
```



Producer/Consumer Solution




```
// Global Variables:
pthread_mutex_t mux  = PTHREAD_MUTEX_INITIALIZER;
int  num_items=0, in=0, out=0, buff[N];
```

Producer Threads:

```
int item;
while(1) {
  item = produce_item();
  pthread_mutex_lock(&mux);
  buff[in] = item;
  in = (in+1)%N;
  num_items++;
  pthread_mutex_unlock(&mux);
}
```

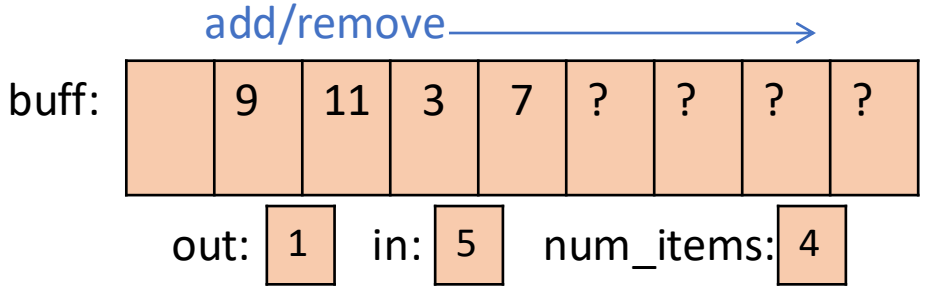
Consumer Threads:

```
int item;
while(1) {
  pthread_mutex_lock(&mux);
  item = buff[out];
  out = (out+1)%N;
  num_items--;
  pthread_mutex_unlock(&mux);
  consume_item(item);
}
```

Need Atomic Access to buff state: `num_items, buff, in, out`

- Want `num_items++` and `num_items--` to be atomic
- Don't want multiple consumer threads simultaneously using or updating `out`
- Don't want multiple producer threads simultaneously using or updating `in`

Producer/Consumer Solution




```
// Global Variables:
pthread_mutex_t mux  = PTHREAD_MUTEX_INITIALIZER;
int  num_items=0, in=0, out=0, buff[N];
```

Producer Threads:

```
int item;
while(1) {
  item = produce_item();
  pthread_mutex_lock(&mux);
  buff[in] = item;
  in = (in+1)%N;
  num_items++;
  pthread_mutex_unlock(&mux);
}
```

Consumer Threads:

```
int item;
while(1) {
  pthread_mutex_lock(&mux);
  item = buff[out];
  out = (out+1)%N;
  num_items--;
  pthread_mutex_unlock(&mux);
  consue_item(item);
}
```

Still need some Scheduling Type of Synchronization:

- Consumer threads must wait when there are no items in buff to consume
- Producer threads must wait when there there is no space in buff to put item

Producer/Consumer Solution

```
// Global Variables:
pthread_cond_t full  = PTHREAD_COND_INITIALIZER;
pthread_cond_t empty = PTHREAD_COND_INITIALIZER;
pthread_mutex_t mux  = PTHREAD_MUTEX_INITIALIZER;
int  num_items=0, in=0, out=0, buff[N];
```

Producer Threads:

```
int item;
while(1) {
  item = produce_item();
  pthread_mutex_lock(&mux);
  while(num_items >= N) {
    pthread_cond_wait(&full,
                      &mux);
  }
  buff[in] = item;
  in = (in+1)%N;
  num_items++;
  pthread_cond_signal(&empty);
  pthread_mutex_unlock(&mux);
}
```

Consumer Threads:

```
int item;
while(1) {
  pthread_mutex_lock(&mux);
  while(num_items == 0) {
    pthread_cond_wait(&empty,
                      &mux);
  }
  item = buff[out];
  out = (out+1)%N;
  num_items--;
  pthread_cond_signal(&full);
  pthread_mutex_unlock(&mux);
  consume_item(item);
}
```

Synchronization Can be Tricky

- Race Condition: missing synchronization

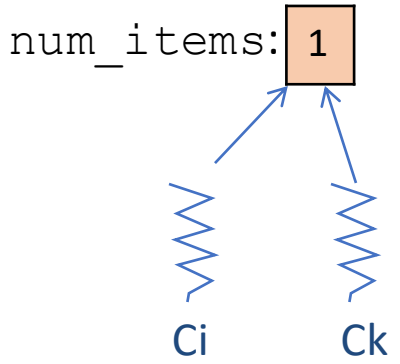
```
if(num_items > 0) {
  pthread_mutex_lock(&mutex);
  item = buff[out];
  out = (out+1)%N;
  num_items--;
  pthread_mutex_unlock(&mutex);
}
…
if(num_items < N) {
  pthread_mutex_lock(&mutex);
  buff[in] = item;
  in = (in+1)%N;
  num_items++;
  pthread_mutex_unlock(&mutex);
}
```

Q: Where is the Race Condition?

Race Condition:

Reading value of `num_items` is NOT Atomic

```
if(num_items > 0) {
  pthread_mutex_lock(&mutex);
  item = buff[out];
  out = (out+1)%N;
  num_items--;
  pthread_mutex_unlock(&mutex);
}
…
if(num_items < N) {
  pthread_mutex_lock(&mutex);
  buff[in] = item;
  in = (in+1)%N;
  num_items++;
  pthread_mutex_unlock(&mutex);
}
```



(1) threads both read 1 simultaneously, and evaluate (1 > 0) is TRUE

(2) One gets mutex first, gets valid item from buff

(3) Other gets mutex second, gets garbage item from buff

Synchronization Can be Tricky

- Deadlock: tid(s) blocked waiting forever
  - bad ordering of concurrent synch calls:

Tid 1:	Tid 2:
pthread_mutex_lock(&**mux1**);	pthread_mutex_lock(&**mux2**);
pthread_mutex_lock(&**mux2**);	Pthread_mutex_lock(&**mux1**);

```
pthread_mutex_lock(&mux1);
if(some bool expr) {
  // do some atomic stuff
  pthread_mutex_unlock(&mux1);
}
```

If cond isn't true, then no call to unlock mutex -> next call to lock will block forever

Parallel Programming Can be Hard

Thinking about all possible orderings of concurrent actions and how Tids may interact/interfere

Some types of errors:

- ***Race conditions:***
  outcome depends on arbitrary OS scheduling order
- ***Deadlock:***
  errors in resource allocation prevent forward progress
- ***Livelock / Starvation / Fairness***:
  arbitrary scheduling can prevent tids progress

These can be difficult to find or difficult to fix

Parallel Programming Can be Hard

Thinking about all possible orderings of concurrent actions and how Tids may interact/interfere

Some types of errors:

- ***Race conditions:***
  outcome depends on arbitrary OS scheduling order
- ***Deadlock:***
  errors in resource allocation prevent forward progress
- ***Livelock / Starvation / Fairness***:
  arbitrary scheduling can prevent tids progress

These can be difficult to find or difficult to fix

Parallel Programming Benefits

- Can greatly improve performance of program if can divide computation into parts that multiple threads can run simultaneously on multiple cores.

- How much faster can we get?
  - With 4 cores? 8 cores? 16 cores? …

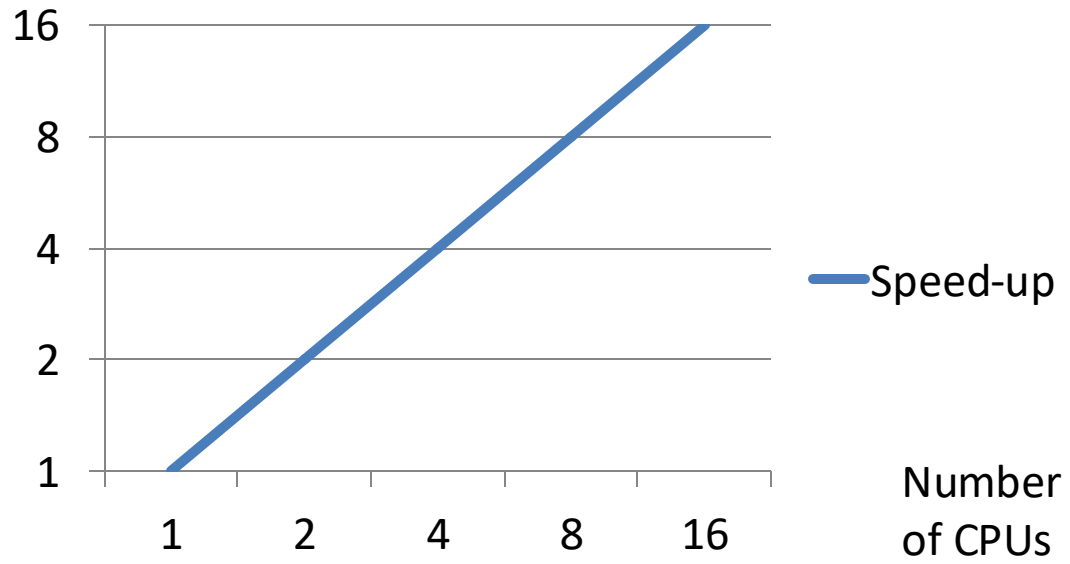
Parallel Performance Metrics

- Speed-up: Ratio of Sequential Time to Parallel

  Ts/Tp: Ts: time to execute sequential version

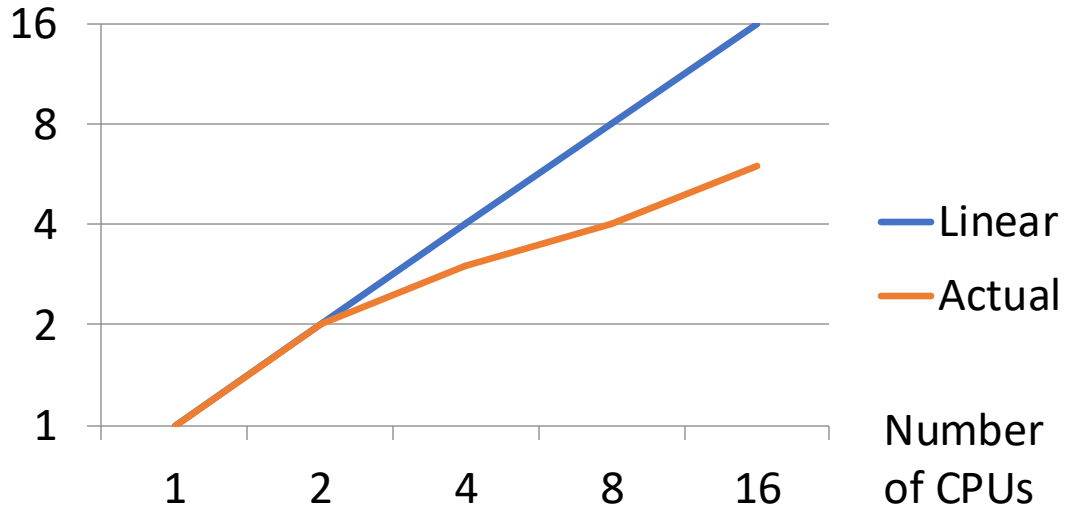
  Tp: time to execute a parallel version

- Ideal: linear speed-up as increase the number of processors (degree of parallelism)



Speed-up Reality

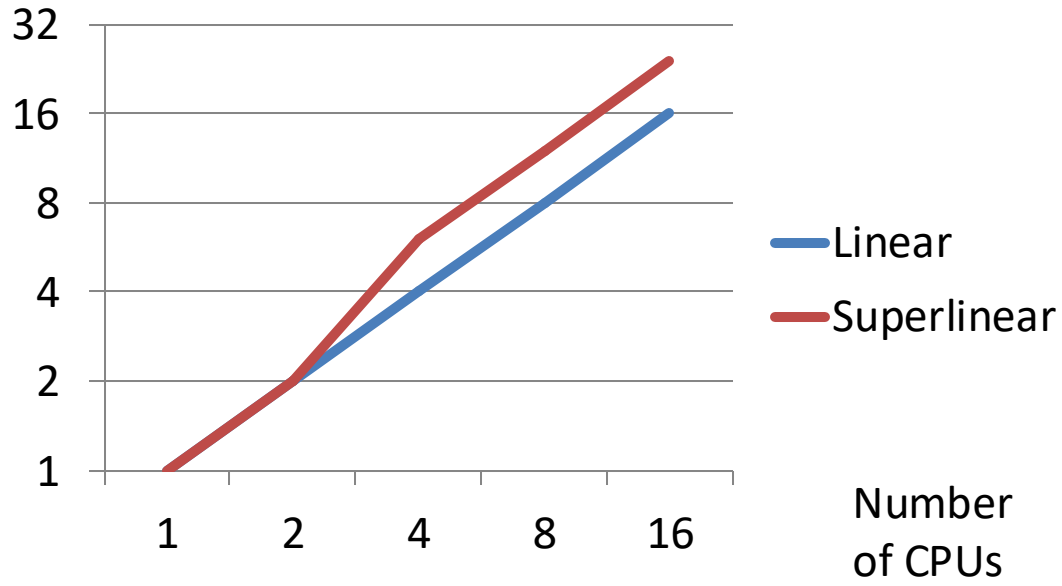
- Often unable to achieve ideal linear speedup



- Why?
  - Added overheads in parallel version of code:
    - thread create/join, synchronization, thread CXS
  - Often limits to parallelism
    - Need good amount of parallel computation between synch
      - 10x10 GOL, 100 threads could be slower than 10 threads
    - Number CPUs limit number threads actually running at once

Speed-up Awesomeness!

- Sometimes parallel program can actually do better than linear speed-up



- Why?
  - Better locality with smaller problem sizes
  - More cache hits (faster memory accesses!)

Performance Limits

- Usually linear speed-up is hard to achieve
  - Embarrassingly parallel solutions often can achieve
    - Parallelism requiring basically no synchronizing actions
  - But most parallel solutions have some parallel overheads that interfere with achieving the ideal
    - Typically some points where threads need to synchronize their actions
- Can we quantify the limits to parallel speed-up?

Amdahl's Law

- Execution time after improvement:

```
(time effected by improvement)/(amount of improvement)
 +  (time not effected by improvement)
```

- Example: 90% of application can be parallelized
  - **10 processors:**
    Tp = .9*Ts/10 + .1*Ts = .19*Ts
    Speed-up = Ts/Tp = **~5.3**
  - **100 processors:**
    Tp = .9*Ts/100 + .1*Ts = .109*Ts
    Speed-up = Ts/Tp = **~9.2**
  - **1000 processors:**
    Speed-up = **~9.9**

Amdahl's Law

Tells us to focus our efforts on the hot spots:

- Look to parallelize (or optimize in any way) the portion of code that accounts for the largest portion of execution time
- Example:
  - If one part accounts for 10% of execution time, the best we can do is completely optimize this portion away
  - At best, the overall effect on performance are reducing total runtime by 10%
- Where is all the time?
  - Usually in loops
  - Could also be implicit and explicit I/O costs

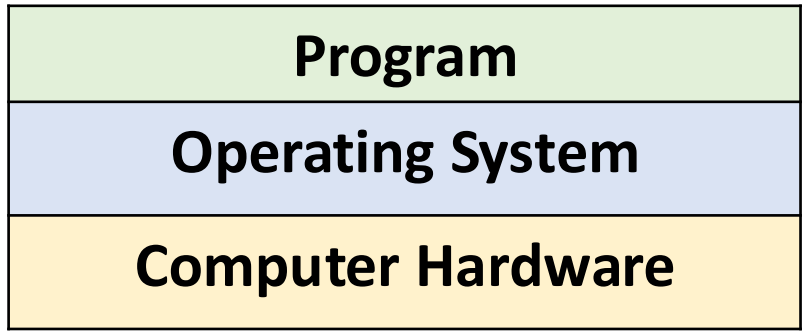
Summary

- Thread: an OS abstraction
  - Multiple threads of execution in single process
  - Private Stack and Register values, shared everything else
- Threads and Parallelism Computing
  - Shared memory model of parallel computing
  - Need to think about parallel ordering of threads
  - Need to think about synchronization
    - Atomic Type
    - Scheduling Type
  - Race Conditions and Deadlock
  - Quantifying the effects of Parallelization:
    - Speed-up and Amdahl's law

CS31 Course Summary

What do you Know?

(1) How a Computer Runs a Program:



How program is encoded to run on HW

OS manages HW provides abstractions

How HW is organized to run programs

(2) How to Efficiently Run Programs

The Memory Hierarchy & its effect performance

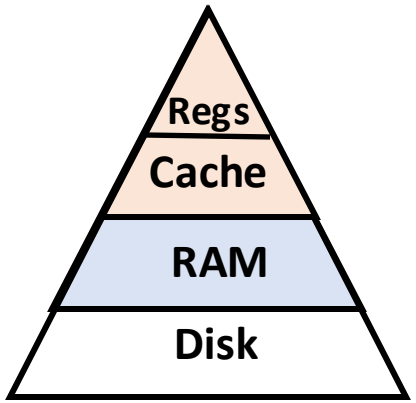
- Caching

OS abstractions for running programs efficiently

- Processes, VM, Threads

Support for Parallel programming

- Threads and Synchronization, Multicore



Why does my program run so slowly?

One Answer: bad algorithm, big-O analysis

Another Answer: Space Usage and Systems costs

- How is my program accessing its memory?
  - Memory Hierarchy Effects, caching, page faults
  - Memory Layout of data structures & access patterns
- Can my program be parallelized?
  - Am I doing it efficiently? Synchronization Costs?
- Where should I focus my performance tuning efforts?

First half of semester

- How HW is designed to run program:
  - Based on Von Neumann Architecture
  - Basic Logic gates to Circuits to big components:
    - Types of Circuits to build CPU, RAM
    - Buses connect components
- Binary encoding of programs
    - Data and Instructions (IA32 assembly)
    - Translation from a high-level-languge (C) to language HW can execute (IA32 assembly)
    - How data structures are laid-out in memory
    - Support for functions, the stack

Second Half of Semester

Efficiency:

- Memory Hierarchy and caching
- How OS manages HW & run programs
  - Implements Easy-to-Use Abstractions on top of HW
  - Process:
    - multi-programming: more than one process in the system at a time
    - process hierarchy, creation, reap, fork-exec, wait, signals
    - CPU context switching and scheduling policies
  - Virtual Memory:
    - protection and lone-view
    - Efficient Use of RAM, paging and page replacement
  - Threads: for Shared Memory Programming
    - Multiple streams of execution in same virtual address space
    - Need Synchronization primitives

Second Half of Semester

- Efficiency
  - Shared Memory Parallel Computing
    - Multicore systems
    - Threaded programming
      - Pthread library functions for creating, synchronizing and joining threads
    - Solving Synchronization Problems
      - Atomic and Scheduling types of synchronization
      - Deadlock and Race conditions
  - Evaluating Program Performance:
    - Speed-up, Amdahl's Law

Final Exam

- When: May 12
- Time: 9 – 12pm
- Where: Singer 33

Studying for the Final

- Know Big Themes from first half of the course
- Know Big Themes AND the details from the second half of the course
  - The Memory Hierarchy and Caching
  - Processes
  - Virtual Memory and Paging
  - Threads and Synchronization

Upper-level Courses that Expand on CS31 Topics

- CS45: Operating Systems
  - Much more in depth on all 2nd half of course topics (and more)
- CS87: Parallel and Distributed Computing
  - Threaded parallelism and a lot of other models, systems, algorithms, associated with parallel and distributed computing
- CS43: Computer Networks
  - Systems issues associated with implementing networked and distributed systems
- CS44: Databases
  - Focus on Database Management Systems, data layout and access methods and the Memory Hierarchy, Parallel & Distr. DB
- CS75: Compilers
  - How compiler is designed to translate a program written in a HLL to assembly (ex. C to IA32), optimization
- CS88: Computer Security and Privacy
  - Focus on Security: (correctness) and (protection) mechanisms on computer systems and architecture using CIA: Confidentiality, Integrity and Availability