Admin

- **When is it?**
  - Midterm *in class* Wednesday Feb 12

- **What resources can I use?**
  - Open note, can print lecture slides
  - No textbook or electronics

- **How much of my grade does it count for?**
  - 50% of overall grade is: \( \max( \text{midterm} > 0 \ ? \ \text{final} : 0, (\text{midterm} + \text{final})/2 ) \)
Content

- Processes & Threads
- Concurrency
- Scheduling
- Virtual Memory (HW/OS)
- Synchronization
- Linking
- Memory Allocation (Monday)

Themes

- Memory models
  - Sequential consistency
- Races
  - Implementing locks
  - Producer/consumer
- Design tradeoffs
  - Using the past to predict the future
- Uniprocessor vs. multiprocessor
Processes & Threads
Processes

● Process
  ○ An instance of a program running
  ○ Has its own view of the machine: address space, open files

● Process control block (PCB)
  ○ Stores information about the process, including:
    ■ State (running, ready, waiting)
    ■ Registers
    ■ Virtual memory mappings
    ■ Open files
  ○ struct thread in pintos
Processes

- Why?
  - Higher throughput
  - Lower latency

Running $A$ then $B$ requires 100 sec for $B$ to complete.
Running $A$ and $B$ concurrently makes $B$ finish faster.
Threads

- Thread
  - Schedulable execution context
  - Allows one process to use multiple CPUs
  - Lighter-weight than process
Kernel vs. User Threads

- **Kernel threads**
  - **Pro:** control
    - Scheduling
    - Priority
  - **Con:** heavy-weight
    - All operations go through kernel
    - More memory/features than needed

- **User threads**
  - Also known as “green threads”
  - **Pro:** more lightweight and flexible
  - **Con:** control
    - IO on one thread blocks all
Context Switching

- **Context switch**
  - Change which process is running

- **How?**
  - Save registers of current thread
  - Restore registers of next thread
  - Return into next thread

- **When?**
  - State change
    - **Blocking call**
    - **Device interrupt** (e.g. disk access completed, packet arrived on network)
  - Can **preempt** when kernel gets control*
    - Traps: system call, page fault, illegal instruction
    - **Periodic timer interrupt**

*unless non-preemptive (thread executes until blocking call)
Scheduling
Scheduling

● **Problem**  
  ○ Given a list of runnable processes, *which do we run?*

● **Goals**  
  ○ Throughput  
  ○ Turnaround time  
  ○ Response time  
  ○ CPU utilization  
  ○ Waiting time

● **Context switch costs**  
  ○ CPU time in kernel  
  ○ Indirect costs
Scheduling Algorithms

- First come first serve
- Shortest job first
- Round-robin
- Priority scheduling
- MLFQS (multilevel feedback queues)
Multiprocessor Scheduling

● Problem
  ○ Given a list of runnables processes, which do we run and \textit{which CPUs do we run them on}?

● Considerations
  ○ Load balancing
  ○ Minimize direct/indirect costs

● Approaches
  ○ Affinity scheduling
    ■ Keep process on same CPU
  ○ Gang scheduling
    ■ Schedule related processes/threads together
Virtual Memory
Virtual Memory HW

- Problem
  - Want multiple processes to co-exist
  - How should processes interface with memory?
- Issues with using physical addresses
  - Protection
  - Transparency
  - Resource exhaustion
- Solution
  - Give each program its own *virtual address space*
  - *Memory Management Unit (MMU)*
    - translates between physical and virtual addresses
How to Map Memory

- Base + bound
  - Physical address = virtual address + base

- Segmentation
  - Divide memory into segments, each of which has a base + bound

- Demand Paging
  - Divide memory into small, equal-sized pages
  - Each process has its own page table
    - Multilevel
    - Translation Lookaside Buffer (TLB) caches recently used translations
  - Any process can have any page, idle pages stored on disk, paged in when used
  - Eviction?
    - Least recently used: use past to predict future
Considerations

● Fragmentation
  ○ Inability to use free memory
  ○ **External** fragmentation (e.g. segmentation)
    ■ Many small holes between memory segments
  ○ **Internal** fragmentation (e.g. paging)
    ■ Unused memory within allocated segments

● Speed
  ○ Disk much slower than memory
  ○ 80/20 rule
    ■ Hot 20 in memory = “working set”

● Local or global page allocation

● Thrashing
  ○ Working set can’t fit in memory
Concurrency
Memory Model

- **Sequential consistency**
  - As if all operations were executed in some sequential order
  - **Downsides**
    - Thwarts hardware/compiler optimizations (e.g. prefetching/reordering)
  - **Requirements**
    - Maintain program order on individual processors
    - Ensure write atomicity
Preventing Races

- Define **critical section**
- Requirements to fake SC?
  - Mutual exclusion
  - Progress
  - Bounded waiting
- How to meet requirements?
  - **Synchronization primitives**
    - Locks, semaphores, condition variables
- What if sharing data with interrupt handler?
  - Uniprocessor: disable interrupts
  - Multiprocessor: disable interrupts + spinlock
Synchronization
Memory System Properties

- **Coherence**
  - Concerns access to a single memory location
    - If A writes $x=1$ and B writes $x=2$, all processes should see the same ordering
  - MESI/MOESI multicore cache coherence
    - Modified, Exclusive, Shared, Invalid, Owned

- **Consistency**
  - Concerns ordering across multiple memory locations
    - If $x=1, y=2$, A reads $x,y$ and B writes $x=3,y=4$, could A ever see $x=1, y=4$?
    - Sequential consistency matches our intuition
Considerations

- **Amdahl’s law**
  - Ultimate limit on parallel speedup if part of task must be sequential

- **Necessary conditions for data race**
  - Multiple threads access the same data
  - At least one of the accesses is a write

- *There is no such thing as a benign data race*

- **Necessary conditions for deadlock**
  - Limited access (mutual exclusion)
  - No preemption
  - Multiple independent requests (hold and wait)
  - **Circularity in graph of requests**
    - A holds mutex x, wants mutex y; B holds y, wants x
Memory Ordering and Fences

- What if we don’t need sequential consistency?
  - Weaker consistency models
  - Atomics, lock-free data structures, read-copy update, MCS spinlock, futex
- X-Y fence
  - Operations of type X sequenced before the fence happen before operations of type Y sequenced after the fence
Linking
Components of Memory

- **Heap**
  - Allocated and laid out at runtime by malloc

- **Stack**
  - Allocated at runtime, layout by compiler

- **Global data/code**
  - Allocated by compiler, layout by **linker**

- **Mmapped regions**
  - Managed by programmer or linker
Program Lifecycle

- Source code → program running
- Compiler/Assembler
  - Generates one **object file** for each source file (e.g. main.c → main.o)
    - References to other files are incomplete (e.g. printf is in stdio.o)
- **Linker**
  - Combines all object files into **executable** file
- **OS Loader**
  - Reads executables into memory
Linker

● Goal
  ○ Object files $\rightarrow$ executable

● How
  ○ Pass 1
    ■ Coalesce like segments
    ■ Construct global symbol table
    ■ Compute virtual address of each segment
  ○ Pass 2
    ■ Fix addresses of code and data using global symbol table
Object Files

```
main.c
extern float sin();
extern printf(), scanf();

main() {
    double x, result;
    printf("Type number: ");
    scanf("%f", &x);
    result = sin(x);
    printf("Sine is %f\n", result);
}
```

```
main.o
"text section"
0 main:
... call printf
... call scanf
... call sin
... call printf

"data section"
0 _s1: "Type number: 
14 _s2: "%f"
17 _s3: "Sine is %f\n"

"symbols"
main T[0]
_s1 D[0]
_s2 D[14]
_s3 D[17]

"relocation"
printf T[30]
printf T[86]
scanf T[52]
sin T[60]
_s1 T[24]
_s2 T[54]
_s3 T[80]
```

"Store the final location of sin at offset 60 in the text section"
Pass 1

Memory map:

Symbol table:

<table>
<thead>
<tr>
<th>Name</th>
<th>File</th>
<th>Sec</th>
<th>Offset</th>
<th>Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td>main.o</td>
<td>T</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>_s1</td>
<td>main.o</td>
<td>D</td>
<td>0</td>
<td>720</td>
</tr>
<tr>
<td>_s2</td>
<td>main.o</td>
<td>D</td>
<td>14</td>
<td>734</td>
</tr>
<tr>
<td>_s3</td>
<td>main.o</td>
<td>D</td>
<td>17</td>
<td>737</td>
</tr>
<tr>
<td>printf</td>
<td>stdio.o</td>
<td>T</td>
<td>38</td>
<td>134</td>
</tr>
<tr>
<td>scanf</td>
<td>stdio.o</td>
<td>T</td>
<td>232</td>
<td>328</td>
</tr>
<tr>
<td>stdin</td>
<td>stdio.o</td>
<td>D</td>
<td>0</td>
<td>760</td>
</tr>
<tr>
<td>stdout</td>
<td>stdio.o</td>
<td>D</td>
<td>8</td>
<td>768</td>
</tr>
<tr>
<td>sin</td>
<td>math.o</td>
<td>T</td>
<td>0</td>
<td>508</td>
</tr>
</tbody>
</table>
Pass 2

text section in main.o

relocation record in main.o

symbol table

text section in a.out
Shared Libraries & Dynamic Linking

- Keep a single shared copy of common libraries in memory
[Unsolicited] Advice
Advice

- Old exams won’t necessarily cover the same material or have the same format
- Understand core themes
  - Identify races in code
  - Identify pros/cons of new approaches
  - Given a workload, be able to select a good approach
- Notice what is/isn’t specified in a question (and state assumptions!)
  - Sequential consistency
  - Uniprocessor vs. multiprocessor
- Rely on notes for facts
  - Might be time-constrained
  - Create a cheat sheet instead of printing all lecture slides (or both?)
- Deep understanding of most material > cursory understanding of all
Good luck!