OS: Lecture 9

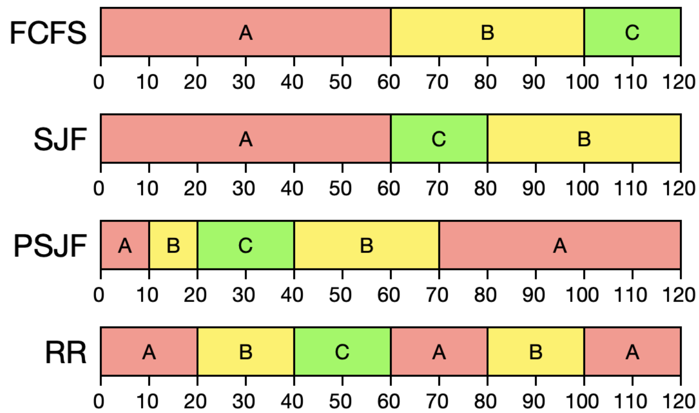
- OS: Lecture 9
  - Review
  - Scheduling Algorithms
    - Trade-offs
    - Priority scheduling
      - Static priority scheduling
      - Limitations
      - Dynamic priority scheduling
      - Can we do better?
    - Multilevel feedback queue (MLFQ)
    - Summary
  - Interprocess communication
    - What is IPC?
    - Why do we need IPC?
    - How to do IPC?
      - Case study: pipe
    - Shared memory
    - What could go wrong?
      - Out of sync?
      - Race conditions

Review

Scheduling algorithms

FCFS, SJF, PSJF, RR

Job	Arrival time	CPU requirement
A	0	60
B	10	40
C	20	20



Scheduling Algorithms

- Trade-offs

This is an inherent trade-off between performance and fairness.

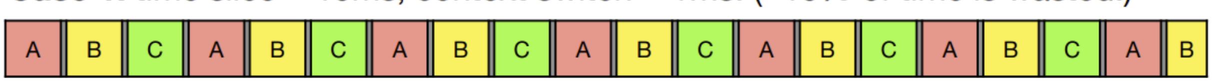
A fair scheduler (such as RR) evenly divides the CPU among active jobs on a small time scale, at the cost of turnaround time.

Most ordinary users run a lot of interactive jobs on modern operating systems.

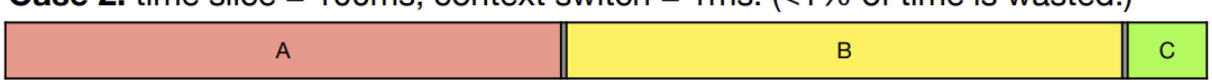
- They value responsiveness more than CPU efficiency.

Another trade-off comes from context switching. It's relatively slow.

Case 1: time slice = 10ms, context switch = 1ms. (~10% of time is wasted.)



Case 2: time slice = 100ms, context switch = 1ms. (<1% of time is wasted.)



- Short time slice ⇒ many context switches ⇒ low CPU efficiency.
- Long time slice ⇒ poor response ⇒ "sluggish."

- **Priority scheduling**

Each job is assigned a priority.

The scheduler always chooses the job with the highest priority to run.

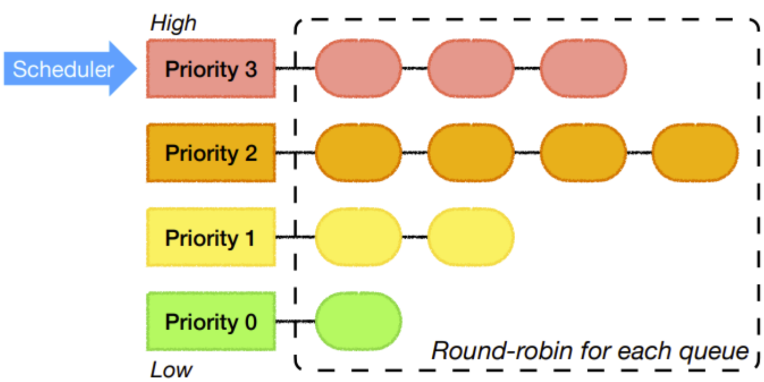
Priorities can be static or dynamic.

Static priority means that a job is assigned a fixed priority when it is submitted to the system.

- **Example:** a background email process should get a lower priority than a real-time video game process.

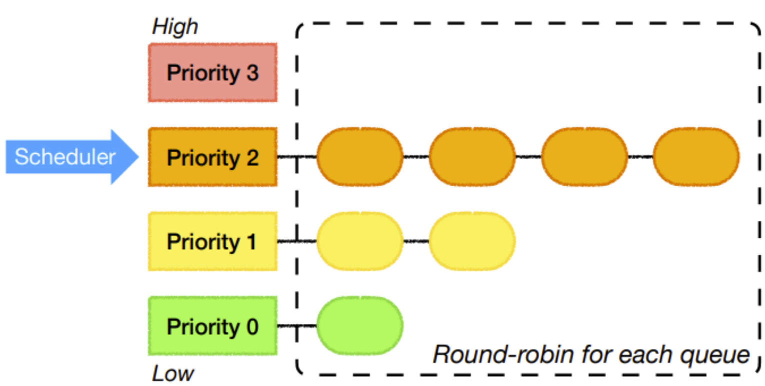
Dynamic priority means that a job's priority may be changing throughout its life in the system.

- Static priority scheduling

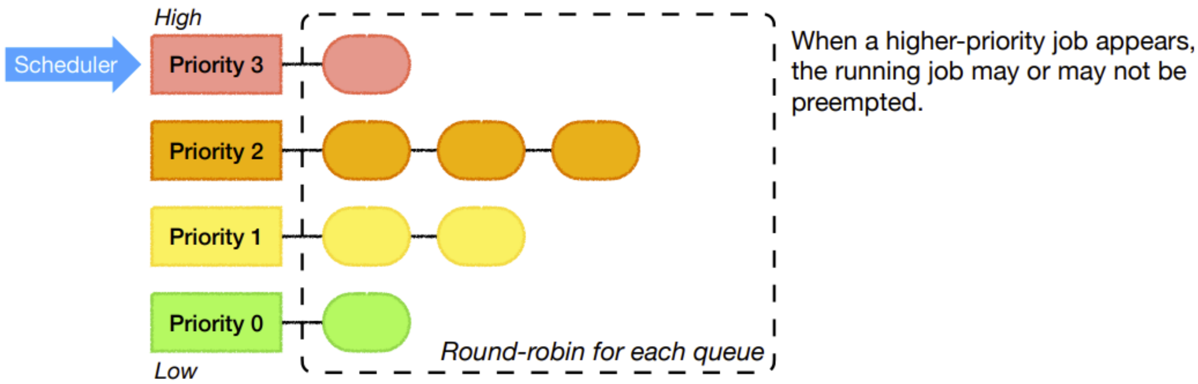


Jobs with Priority 3 are scheduled first.

They should be short-lived to prevent starvation of lower-priority jobs.



When there is no job with Priority 3 left, jobs with Priority 2 will be scheduled.



- Limitations

Limitations

High-priority jobs may run for a prolonged period, or even indefinitely.

Low-priority jobs may starve to death.

- **Rumor:** when the IBM 7094 mainframe at MIT was shut down in 1973, people found a low-priority process submitted in 1967 had not yet been run.

It does not differentiate between CPU-bound and I/O-bound jobs.

- I/O-bound jobs spend most of their time waiting for I/O to complete.
- When such a job wants the CPU, we'd better schedule it immediately to let it start its next I/O request.
- In that way, I/O requests can proceed in parallel with another process actually computing.

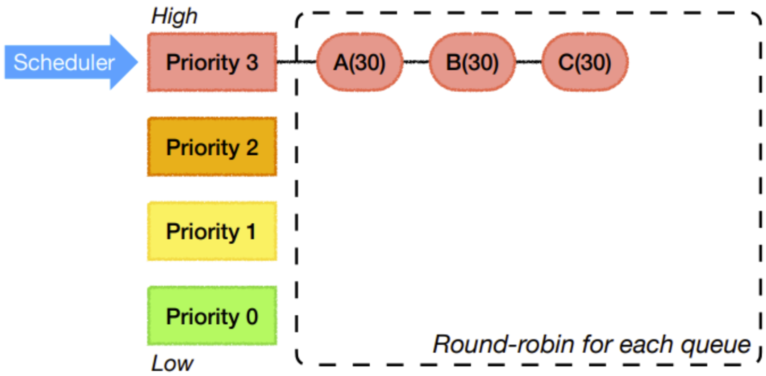
- Dynamic priority scheduling

There is no standard way to assign priorities dynamically.

Let's look at an **example** policy.

Rules:

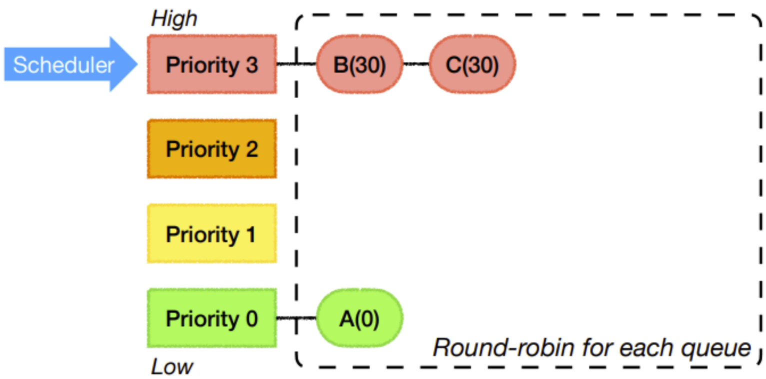
- All jobs start running at Priority 3 with time slice = 30ms.
- A job is preempted if its time slice is used up or it starts waiting for I/O.
- When a job is preempted, its priority is changed to $\left\lceil \frac{\text{its time slice left}}{10\text{ms}} \right\rceil$.



All jobs start running at Priority 3 with (time slice = 30ms).

A is scheduled first.

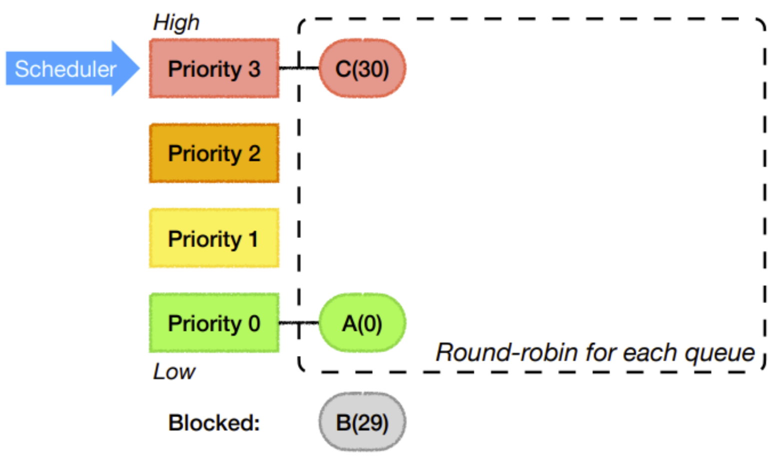
Let's say A is a CPU-bound process and will use up its time slice.



A has used up its time slice and its priority is changed to 0.

B is scheduled next.

Let's say B is an I/O-bound process, and it's blocked right after its execution starts.

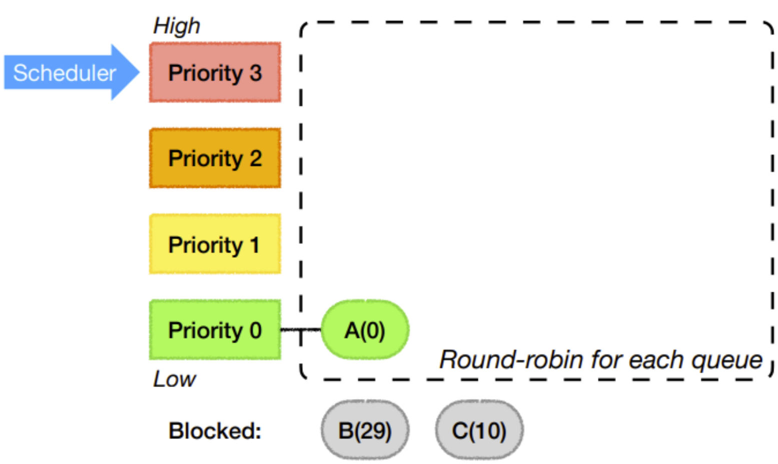


B is removed from the run queue until the I/O completes.

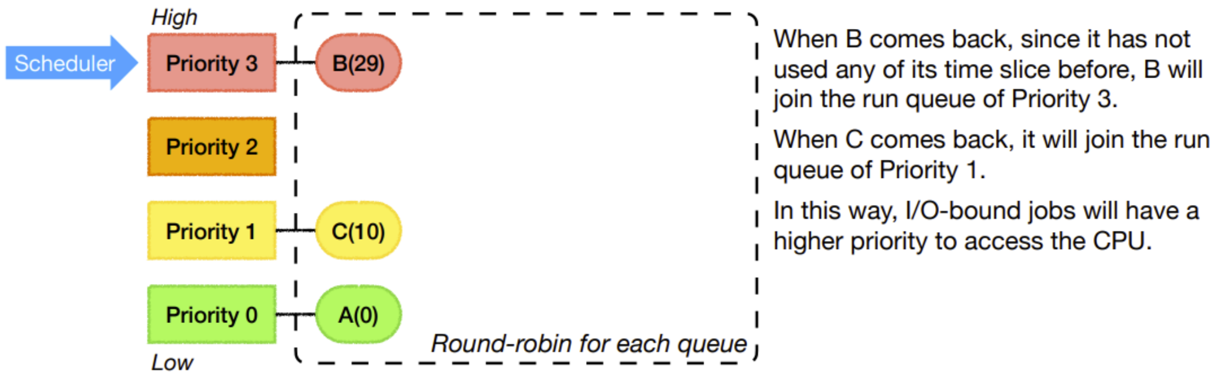
C is scheduled next.

Let's say C is a mixed process.

It runs on CPU for 20ms and starts waiting for I/O.



C's priority is changed to 1, and C is removed from the run queue.



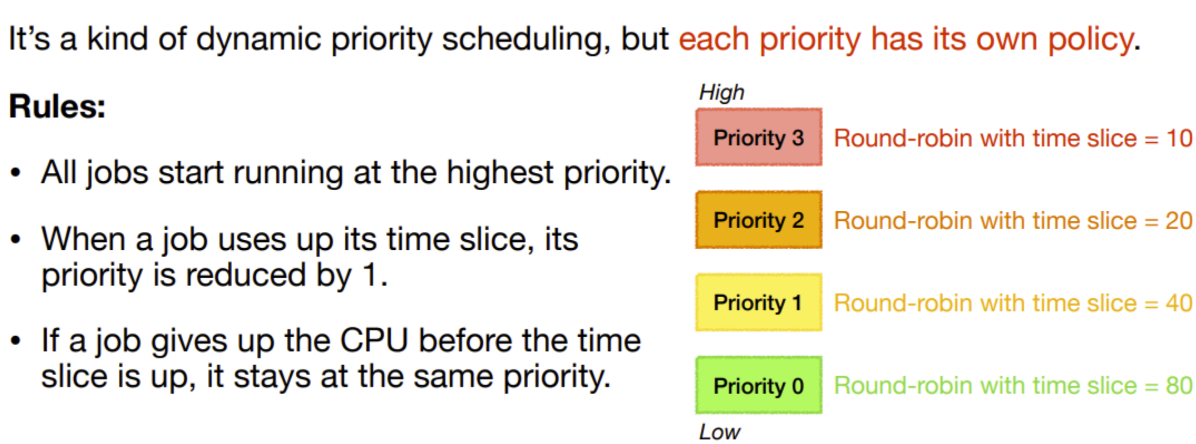
Can we do better?

Remember the trade-off between efficiency and responsiveness?

Ideally, we want to…

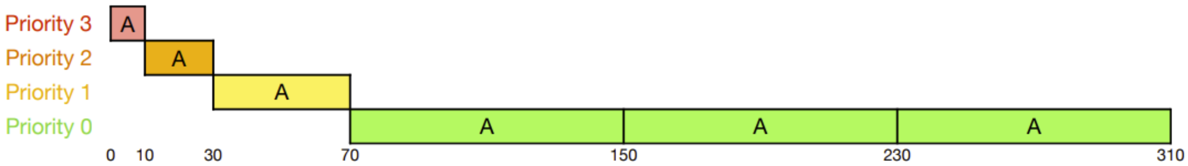
- **Optimize turnaround time.**
  - Run shorter jobs first.
  - Give CPU-bound jobs a large time slice to reduce context switching.
- **Make the system feel responsive to interactive users.**
  - Can't give all jobs a large time slice.
  - Minimize response time.

Multilevel feedback queue (MLFQ)



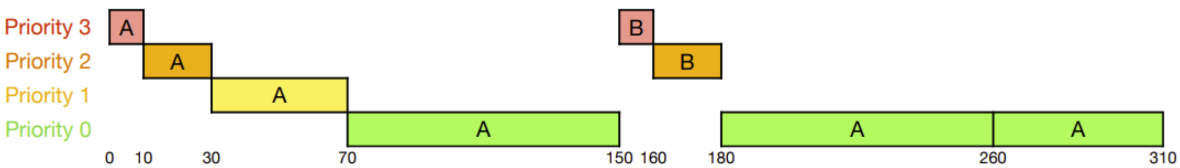
> This is just an example
>
> In reality, we might have way more levels than it. We could have 20 levels or even over a hundred.

Example 1: a single long-running job



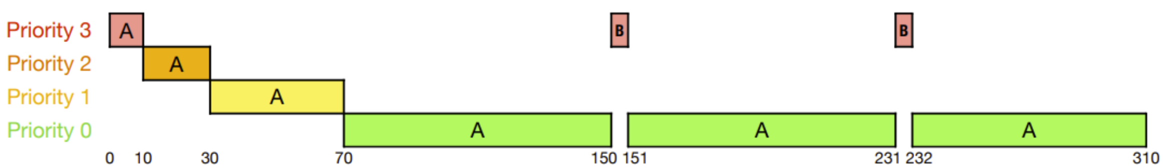
- Long-running jobs get a large time slice to reduce context switching. ✓

Example 2: along came a short job



- Short jobs run first. ✓

Example 3: what about I/O?



- I/O-bound jobs have a higher priority to access the CPU. ✓

Are there any limitations?

- Long-running jobs may starve if there are *too many* interactive jobs.
- What if a program changes its behavior over time?
  - A job may need to run for a long time when it first starts. After that, it becomes interactive.
  - Such a job will be punished forever.

New rule: the priority boost

- After some time period, move all jobs to the highest priority.

Here's a summary of what we've achieved so far…

Rule 1: If Priority(A) > Priority(B), A runs (B doesn't).

Rule 2: If Priority(A) = Priority(B), A & B run in round-robin fashion using the time slice of the given queue.

Rule 3: When a job enters the system, it is placed at the highest priority (the topmost queue).

Rule 4: Once a job uses up its time slice at a given level, its priority is reduced (it moves down one queue).

Rule 5: After some time period, move all the jobs in the system to the topmost queue.

MLFQ observes how jobs behave over time, and prioritize them accordingly.

- It can deliver excellent overall performance (similar to SJF/PSJF) for short-running interactive jobs.
- It's fair and makes progress for long-running CPU-intensive workloads.

Therefore, many modern operating systems use a form of MLFQ as their base scheduler.

- Summary

So, *what's the best scheduling algorithm?*

Unfortunately, there is no best or standard algorithm, partly because…

- We can't predict the CPU requirement of a process.
  - We don't even know if a job will eventually terminate!
- Online scheduling is an NP-hard problem.

Linux employs the Completely Fair Scheduler (CFS) since kernel 2.6.23.

- It's a dynamic priority scheduling algorithm based on red-black trees.

Interprocess communication

- What is IPC?

Processes often need to communicate with one another.

This is called **interprocess communication (IPC)**.

Have you used any methods of IPC?

- Signal: `kill -STOP 2250`
- Pipeline: `ls | less`
- File, socket, shared memory…

- Why do we need IPC?

To share information

- Of course…

To reuse software

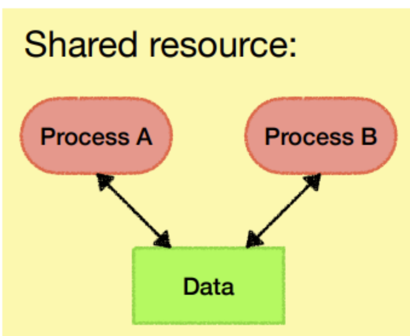
- You can implement a spell checker by…

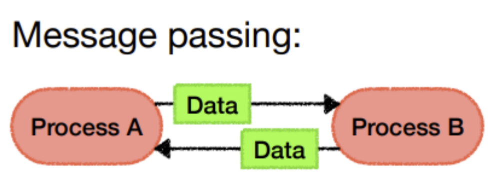
```
curl "https://en.wikipedia.org/wiki/Pipeline_(Unix)" | sed 's/[^a-zA-Z ]/ /g' | tr 'A-Z ' 'a-z\n' |
grep '[a-z]' | sort -u | comm -23 - <(sort /usr/share/dict/words) | less
```

To speedup computation

- **Example:** MapReduce.
- You can divide a job into tasks, run them as various processes in parallel, and combine the results.
- To learn more about MapReduce and big data analytics systems, take CSCI-UA.0476 or CSCI-GA.2436.

- How to do IPC?



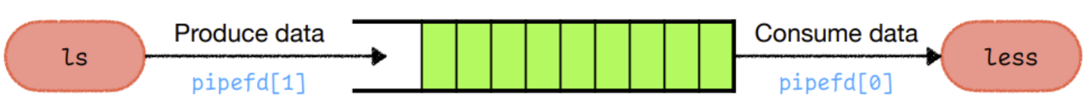


- Case study: pipe

Example: ls | less

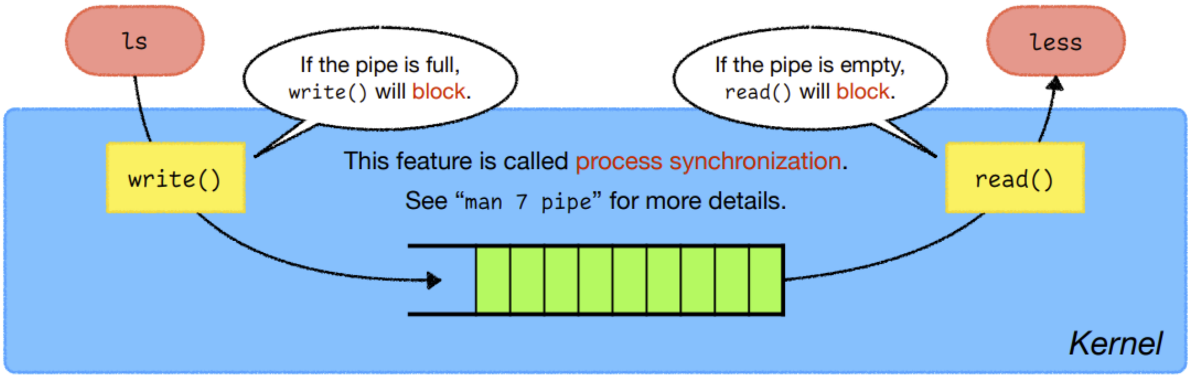
From a programmer's perspective…

The `pipe()` system call returns two file descriptors: `pipefd[0]` and `pipefd[1]`.



This is called a producer-consumer model.

From the kernel's perspective…



- Shared memory

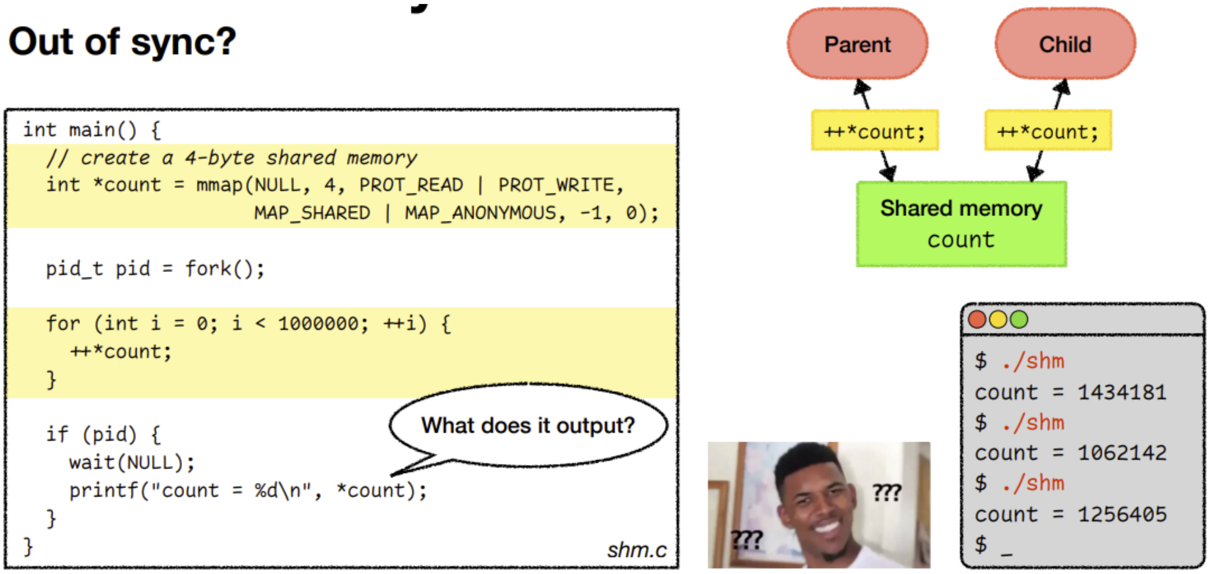
Shared memory is…

- A region of memory created by the kernel;
  - See "`man 7 shm_overview`" for more details.
- Visible to all processes in the system;
  - By contrast, a pipe is only visible to two processes.
- Accessible by all processes in the system.
  - However, there are syscalls to change the ownership and permissions of the shared memory.

- What could go wrong?

  - In the case of a pipe, the kernel provides a form of synchronization.
  - However, for shared memory, it's up to the processes to coordinate.
  - What could possibly go wrong?

- Out of sync?



> “
>
> `mmap()` system call can create a shared memory

```
int main() {
  // create a 4-byte shared memory
  int *count = mmap(NULL, 4, PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMO

  pid_t pid = fork();

  for (int i = 0; i < 1000000; ++i) {
    ++*count;
  }

  if (pid) {
    wait(NULL);
    printf("count = %d\n", *count);
  }
}
[yt2475@linserv1 proc]$ gcc -o shm shm.c
```

```
[yt2475@linserv1 proc]$ gcc -o shm shm.c
[yt2475@linserv1 proc]$ ./shm
count = 1157264
[yt2475@linserv1 proc]$ ./shm
count = 1251921
[yt2475@linserv1 proc]$ ./shm
count = 1190847
```

```c
int main() {
  // create a 4-byte shared memory
  int *count = mmap(NULL, 4, PROT_READ | PROT_WRITE,
                    MAP_SHARED | MAP_ANONYMOUS, -1, 0);

  pid_t pid = fork();

  for (int i = 0; i < 1000000; ++i) {
    ++*count;
  }

  if (pid) {
    wait(NULL);
    printf("count = %d\n", *count);
  }
}
```

shm.c

```
$ gcc -S shm.c
$ cat shm.s
…
    movq    -16(%rbp), %rax
    movl    (%rax), %eax
    leal    1(%rax), %edx
    movq    -16(%rbp), %rax
    movl    %edx, (%rax)
…
```

```
$ gcc -S shm.c
$ cat shm.s
…
    movq    -16(%rbp), %rax
    movl    (%rax), %eax
    leal    1(%rax), %edx
    movq    -16(%rbp), %rax
    movl    %edx, (%rax)
…
```

```
a = count;
a = *a;          →  a = *count;
d = a + 1;       →  d = a + 1;
a = count;
*a = d;          →  *count = d;
```

What could go wrong?

	Parent	Child	count
Parent is running	`a = *count; (a = 0)`		0
	`d = a + 1; (d = 1)`		0
Context switch			
Child is running		`a = *count; (a = 0)`	0
		`d = a + 1; (d = 1)`	0
		`*count = d; (*count = 1)`	1
Context switch			
Parent is running	`*count = d; (*count = 1)`		1

Race conditions

This scenario is called a **race condition** (or, more specifically, a **data race**).

The results depend on the **timing** of the execution, i.e., the particular order in which the shared resource is accessed.

Race conditions are always bad…

- Worse yet, compiler optimizations may generate crazy output if your code has data races.
- What if you compile the previous code with "gcc -O1" and "gcc -O2"?
- To learn more about undefined behavior (a.k.a. "nasal demons"), read Schrödinger's Code.

Because the computation is **nondeterministic**, debugging is no fun at all.

- Heisenbug: bugs that disappear or change behavior when you try to debug.

> “
>
> This is also one of the **undefined behavior**s in C
>
> The **undefined behaviors** come from data races.
>
> If your code contains a data race, it's undefined, and the compile can choose whatever code it want to generate.
>
> Will cause Heisenbug:
>
> - it seems to work when you try to debug
>
> ```
> [yt2475@linserv1 proc]$ gcc -O1 -o shm shm.c
> [yt2475@linserv1 proc]$ ./shm
> count = 1000000
> [yt2475@linserv1 proc]$ ./shm
> count = 1000000
> [yt2475@linserv1 proc]$ ./shm
> count = 1000000
> [yt2475@linserv1 proc]$ ./shm
> count = 1000000
> ```
>
> ```
> [yt2475@linserv1 proc]$ gcc -O2 -o shm shm.c
> [yt2475@linserv1 proc]$ ./shm
> count = 2000000
> [yt2475@linserv1 proc]$ ./shm
> count = 2000000
> [yt2475@linserv1 proc]$ ./shm
> count = 2000000
> [yt2475@linserv1 proc]$ ./shm
> count = 2000000
> ```