COMP 206 – Intro to Software Systems

Distributed Software Systems made of multiple coordinating processes

Lecture 21 – November 21st, 2018

COMP 206 Final Review

Fall 2018

Thu Dec 13, 6-9 PM
@ MAASS 112

fb.com/myCSUS/events

Distributing the work of a software system

- We have seen that today's most important systems are distributed world-wide
  - The web
  - Social networks
  - Banking systems

- We now know how those various processes can pass information between each other, but we still need more tools to coordinate them effectively:
  - Web servers that need to handle millions of connected users
  - Data science code that wants to process billions or trillions of records
  - Web search “spiders” crawling different parts of the internet and recording sites in a single place

Distributed Systems Outline

- Linux as a multi-process operating system
  - Basic concepts in scheduling
- A taste of coordination theory: Dining Philosophers
- Practical C coding tools related to these problems:
  - system() C library function
  - fork() and exec() system calls
  - Semaphores

Getting off the ground: two processes on Linux

- In one terminal, run "tail" and ask it to continue refreshing:

```
$ tail –f junk.txt
Hello world
Do you hear me?
```

- In a second terminal, start dumping text into the file in some way:

```
$ echo "Hello world" >> junk.txt
$ echo "Do you hear me?" >> junk.txt
```

Getting off the ground: two processes on Linux

- In one terminal, run "tail" and ask it to continue refreshing:

$ tail –f junk.txt

Hello world

Do you hear me?

- In a second terminal, start dumping text into the file in some way:

$ echo "Hello world" >> junk.txt

$ echo "Do you hear me?" >> junk.txt

This is all no problem in Linux, and illustrates that the Operating System is managing our two or more processes fairly.

system() : run a shell command

- Anything that you can run on the shell can also be executed with system()
- The command is run in a new shell instance (new variables etc), which is a separate process
- Your process blocks until the shell command returns

```
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main () {
   char command[50];

   strcpy( command, "ls -l" );
   system(command);

   return(0);
}
```

Let us compile and run the above program that will produce the following result on my unix machine –

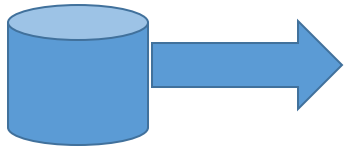
```
drwxr-xr-x 2 apache apache 4096 Aug 22 07:25 hsperfdata_apache
drwxr-xr-x 2 railo railo 4096 Aug 21 18:48 hsperfdata_railo
rw------ 1 apache apache 8 Aug 21 18:48 mod_mono_dashboard_XXGLOBAL_1
rw------ 1 apache apache 8 Aug 21 18:48 mod_mono_dashboard_asp_2
srwx---- 1 apache apache 0 Aug 22 05:28 mod_mono_server_asp
rw------ 1 apache apache 0 Aug 22 05:28 mod_mono_server_asp_1280495620
```

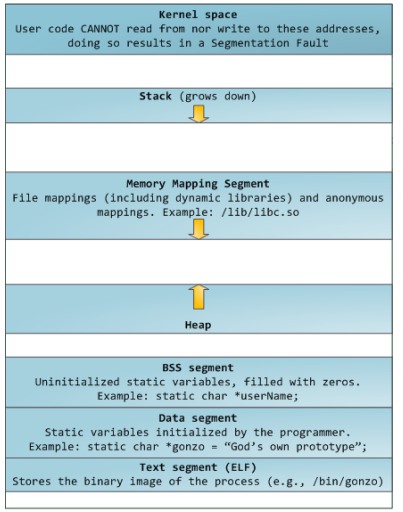
Starting processes ourselves without the shell

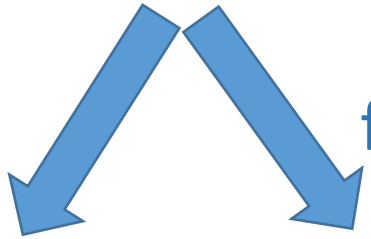
- The fork() system call
- Including fork() in a C program spawns another process by copying the current one.
  - This includes all memory, variables, code, and the process pointer
- The new process starts executing at the line immediately after fork(). The old process continues on that line also, immediately.
- fork() returns an informative value that is our main way to determine if we are in the “new” process or the original:
  - Negative if error
  - Zero in the child process
  - A positive value, the child's process ID in the parent

The process memory view of fork()

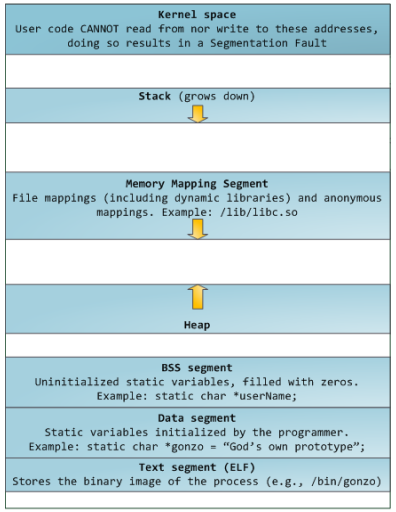
User runs $ ./a.out. OS loads from disk.

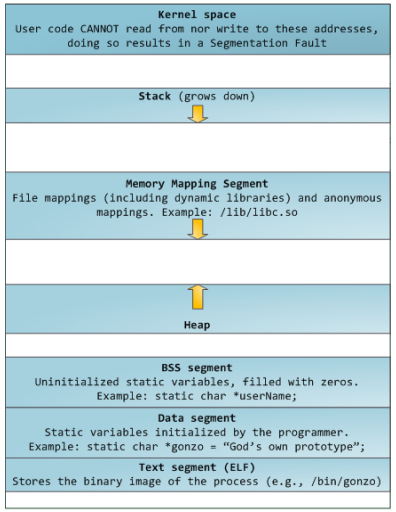






fork() call





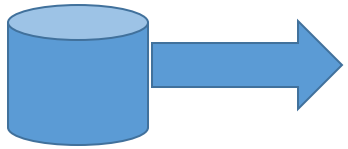
- Prior to fork we have 1 process as usual, started by the user
- At the "fork", the OS copies all of the process memory into a new spot. Includes:
  - The program counter (says which line of code to execute next)
  - The stack frames
  - The heap
- The two processes now execute in parallel

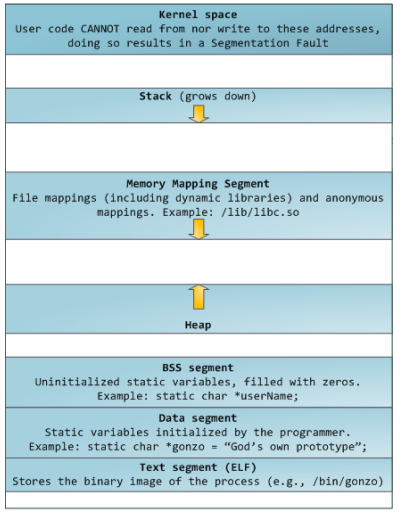
The exec() family of system calls

- Sometimes, we don't want the exact same code running twice, but rather a different kind of process.
  - Example, the system() call to the shell

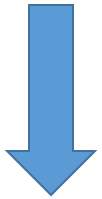
- Here, exec, execv, execl etc give us some options for how to bring this new code into existence

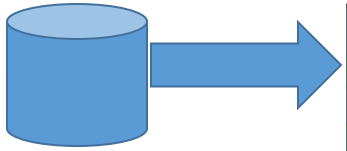
User runs $ ./a.out. OS loads from disk.

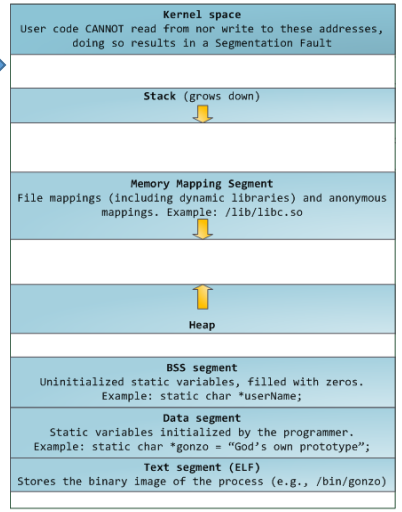




execl("/bin/ls") causes OS to load new program. Replacing the process memory.







Fork and execl, our first example

- This program creates a child process using the fork() system call.
- The child runs "ls"
- The parent coordinates with wait(), then prints the outcome

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
int main()
{
   int x = fork();
   if (x == -1) {
      perror("fork  failed");
      return(1);
   } else if (x == 0) {
      /* child */
      execl("/bin/ls",  "ls", (char *)NULL);
      perror("/bin/ls");
      return(1);
   } else {
      /* parent  */
      int status, pid;
      pid = wait(&status);
      printf("pid  %d exit status %d\n", pid, WEXITSTATUS(status));
      return(0);
}}
```

Important helpers: wait() and waitpid()

- wait() will block until a child process ”ends” meaning:
  - It calls exit();
  - It returns from main
  - It receives a signal (from the OS or another process) whose default action is to terminate. (such as ctrl-c, ctrl-x etc)
- Returns immediately if child ended first

- waitpid() is similar, but sometimes we like one child more than the others!

But who actually runs next?

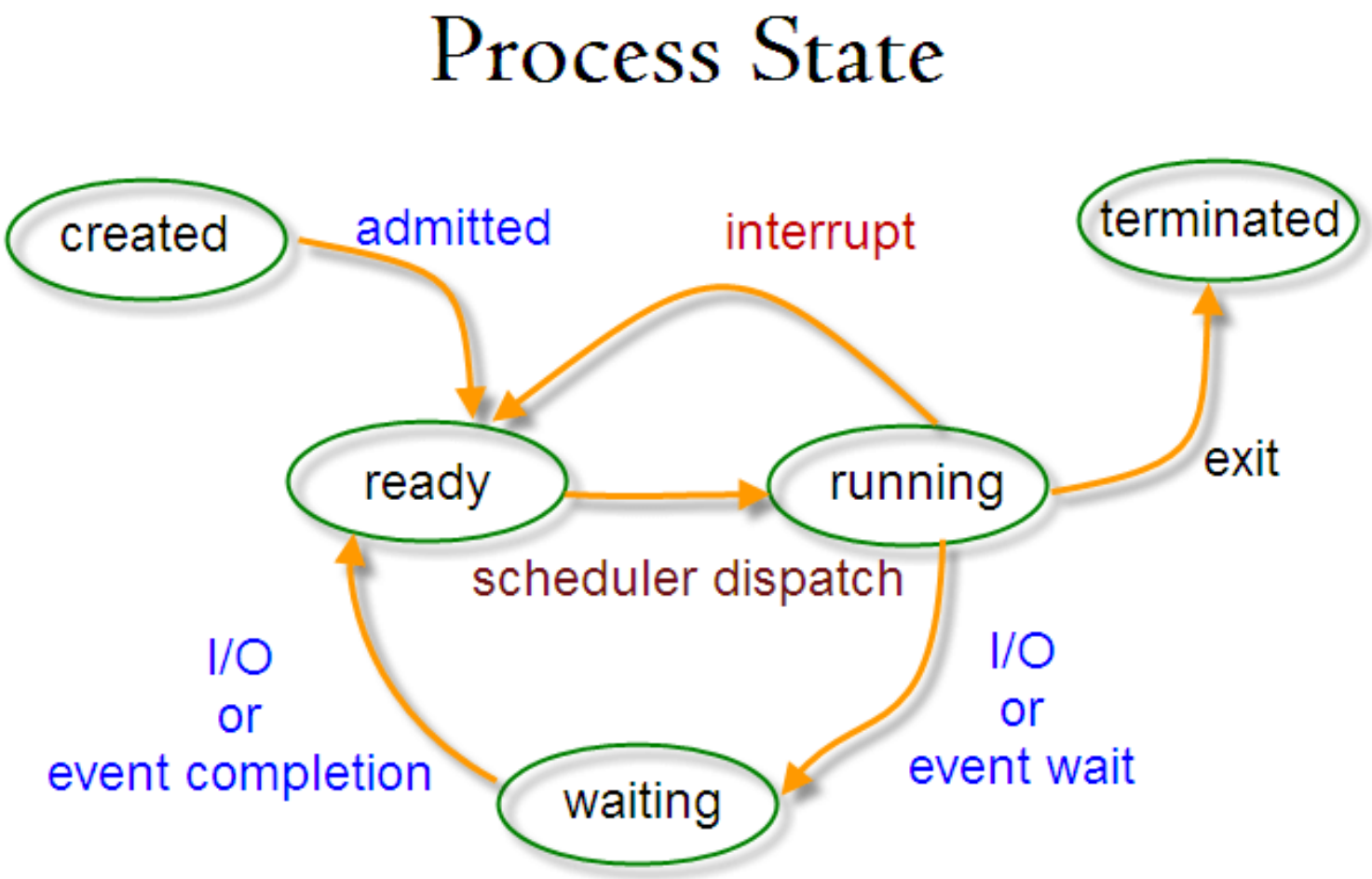
- This program creates a child process using the fork() system call.
- Both processes print from 1 to 200

- Will the parent or child finish first?
- Will one process always print to 200 prior to the other?

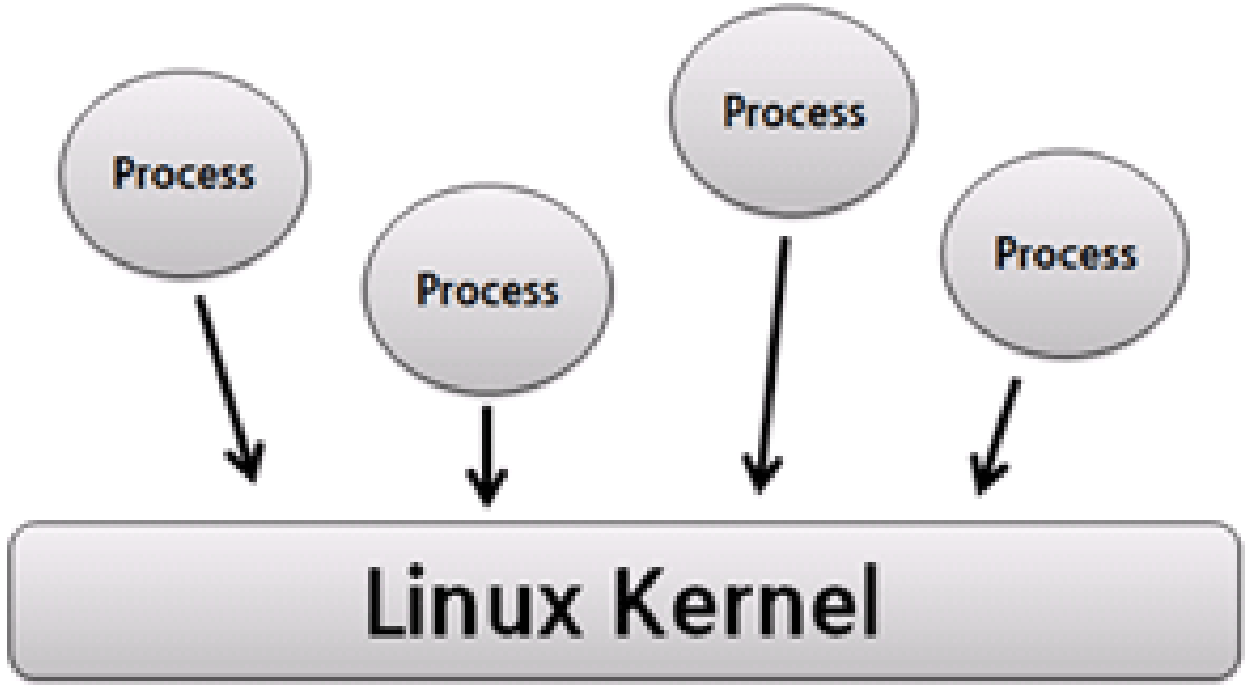
```
#include  <stdio.h>
#include  <string.h>
#include <unistd.h>
#include  <sys/types.h>
#define   MAX_COUNT  200
#define   BUF_SIZE   100
void  main(void)
{
    pid_t  pid;
    int    i;
    char   buf[BUF_SIZE];

    fork();
    pid = getpid();
    for (i = 1; i <= MAX_COUNT; i++) {
        sprintf(buf,
                "This line is from pid %d, value = %d\n", pid, i)
        write(1, buf, strlen(buf));
    }
}
```

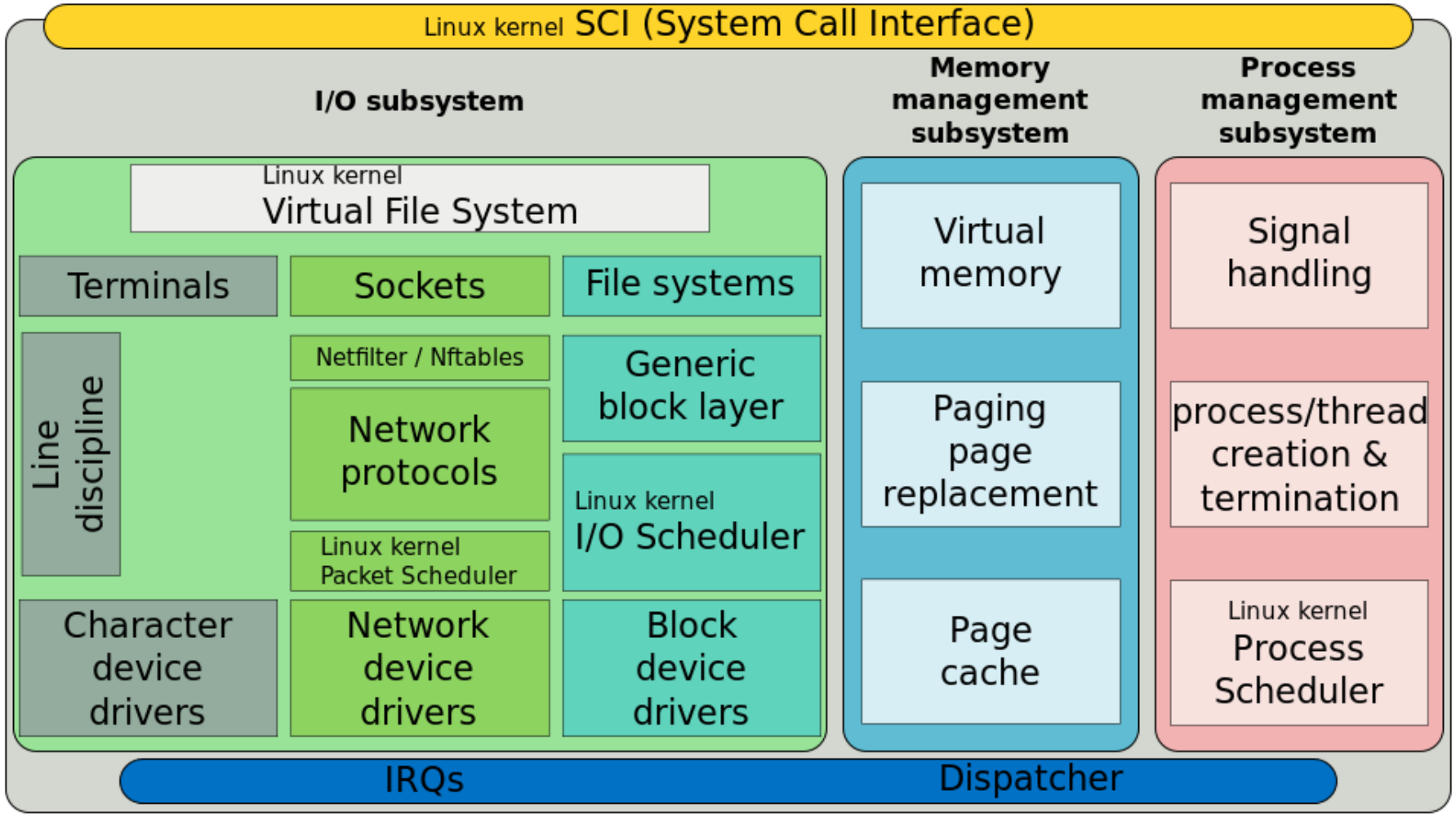
A process does not simply "run"



The operating system manages each process' access to CPU



Linux: scheduler lives within the kernel



Concepts in scheduling

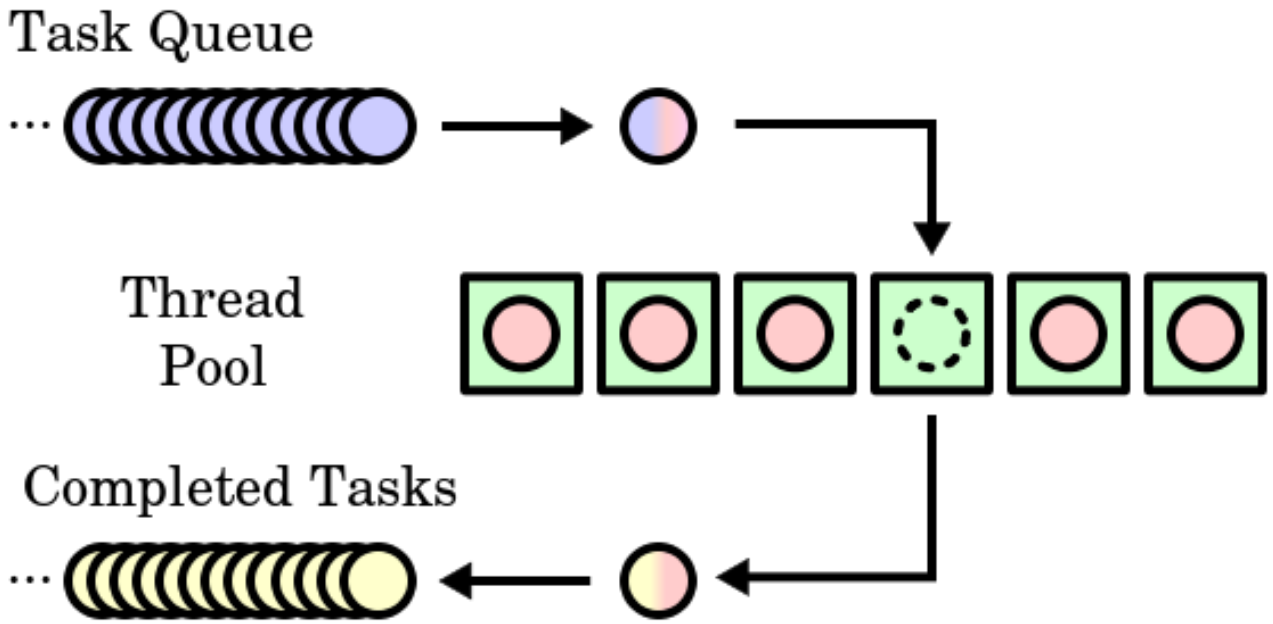
- **Core idea**: We have only 1 (or a few) CPUs per machine, so many processes are "waiting", but if we switch them fast enough, user feels like "it all happens at once".
  - We have to balance long running vs slow running jobs carefully
  - Operations like accessing the disk are long compared to running processor instruction, so we have a chance to be efficient

- A good scheduler is one that provides a high quality of service for each process, measured in one of several ways:
  - **Wait time**: duration between requesting to start and getting first service
  - **Latency**: time until completion
  - **Throughput**: how many resources are being used per second

- Additionally, we may think about fairness (worst-served process) or overall performance (average of all processes)

A first scheduling algorithm: First in First Out

- Known as FIFO, this is a trivial scheduler to implement:
  - Every time the CPU is free, run the task at the beginning of the queue
  - Every time a new task requests to run, place it at the end of the queue



FIFO Discussion

- Pros and Cons?

- How would we rank FIFO on our 3 criteria:
  - Wait time
  - Latency
  - Throughput

FIFO Discussion

- Pros:
  - It is very fair. All tasks get to run eventually and the order of being ready is a human understandable way to make the decision (e.g.: your bank)
  - It is trivial to implement, so the kernel itself will be simple (Linux design principle!)
- Cons:
  - A single long running task can delay many shorter tasks
  - Each task has to run completely in one session, so no chance to exploit IO breaks etc.

Another possibility: Round Robin

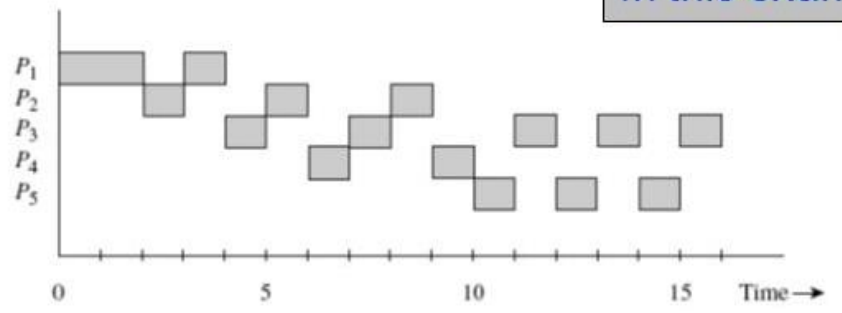
- Concept: break time up into slices that are about the size of the shortest process you expect, run each task for the slice time only (not until completion), cycle through circular task queue sequence repeatedly

Time of scheduling		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Position of	P_1	1	1	2	1												
Processes in	P_2			1	3	2	1	3	2	1							
ready queue	P_3				2	1	3	2	1	4	3	2	1	2	1	2	1
(1 implies	P_4					3	2	1	3	2	1						
head of queue)	P_5									3	2	1	2	1	2	1	
Process scheduled		P_1	P_1	P_2	P_1	P_3	P_2	P_4	P_3	P_2	P_4	P_5	P_3	P_5	P_3	P_5	P_3

c	t_a	w
4	4	1.33
9	7	2.33
16	13	2.60
10	6	3.00
15	7	2.33

$\overline{t_a}$ = 7.4 seconds, $\overline{w}$ = 2.32
c: completion time of a process

In this example, $\delta = 1$



Round robin Discussion

- Pros and cons?

- Again, what about the performance measures:
  - Wait time
  - Latency
  - Throughput

Round robin Discussion

- Pros:
  - Fast tasks finish fast (they might be less than one time slice)
  - Retains fairness, everyone gets an equal slice
- Cons:
  - Imagine many equally-sized tasks starting all at once. They all finish around the time when the *latest* task would finish under FIFO
  - There is more switching between tasks, as we pre-empt. If this is an expensive operation, we waste time.

What does Linux really do?

- It adds a concept of priority called "niceness". Each process has a niceness level, and the scheduler tries to give more time to less nice processes. This is important as practically all tasks are not equal:
  - For example, the user playing a game in one window while the entire disk is being backed up, or virus scanned in the "background".
- It estimates the running duration of each process, so that short tasks can pre-empt
- It interacts with sleeping/waiting processes (we'll see in a bit) and IO delays

- This is all called "Completely Fair Scheduler"

Take-aways from scheduling

- There is no single best ordering. New tasks arrive as we go and the OS is never sure what will come next (the user is un-predictable).

- Modern schedulers introduce delays that are out of our control as a user-space programmer. We must be ready for our process to be "pre-empted", and cannot assume the order we start programs is the order in which they will finish!

- This is already true on one machine, but becomes *very* true across networks.

Now that we can run multiple processes...

- With great power comes great responsibility. We can create software systems with 2, 3, … thousands of processes that interact to do a single job.

- To really get work done, the scheduler is not enough. As Software Systems programmers, we must actively organize our processes.

Recall, our concurrency disaster example

- Remember this BASH script called "add_ten.bash"
- Running "$ bash add_ten.bash" adds ten. Great!
- Running "$ bash add_ten.bash & add_ten.bash" (two simultaneous processes) usually adds only ten (why?)
- Stacking 4-6 simultaneous calls usually resets to 1 or 2 because writing to the file is not safe to parallelize:
  - We call this a ***critical*** operation

```
#!/bin/bash
for i in 0 1 2 3 4 5 6 7 8 9
do
        x=`cat file`
        expr $x + 1 > file
done
```

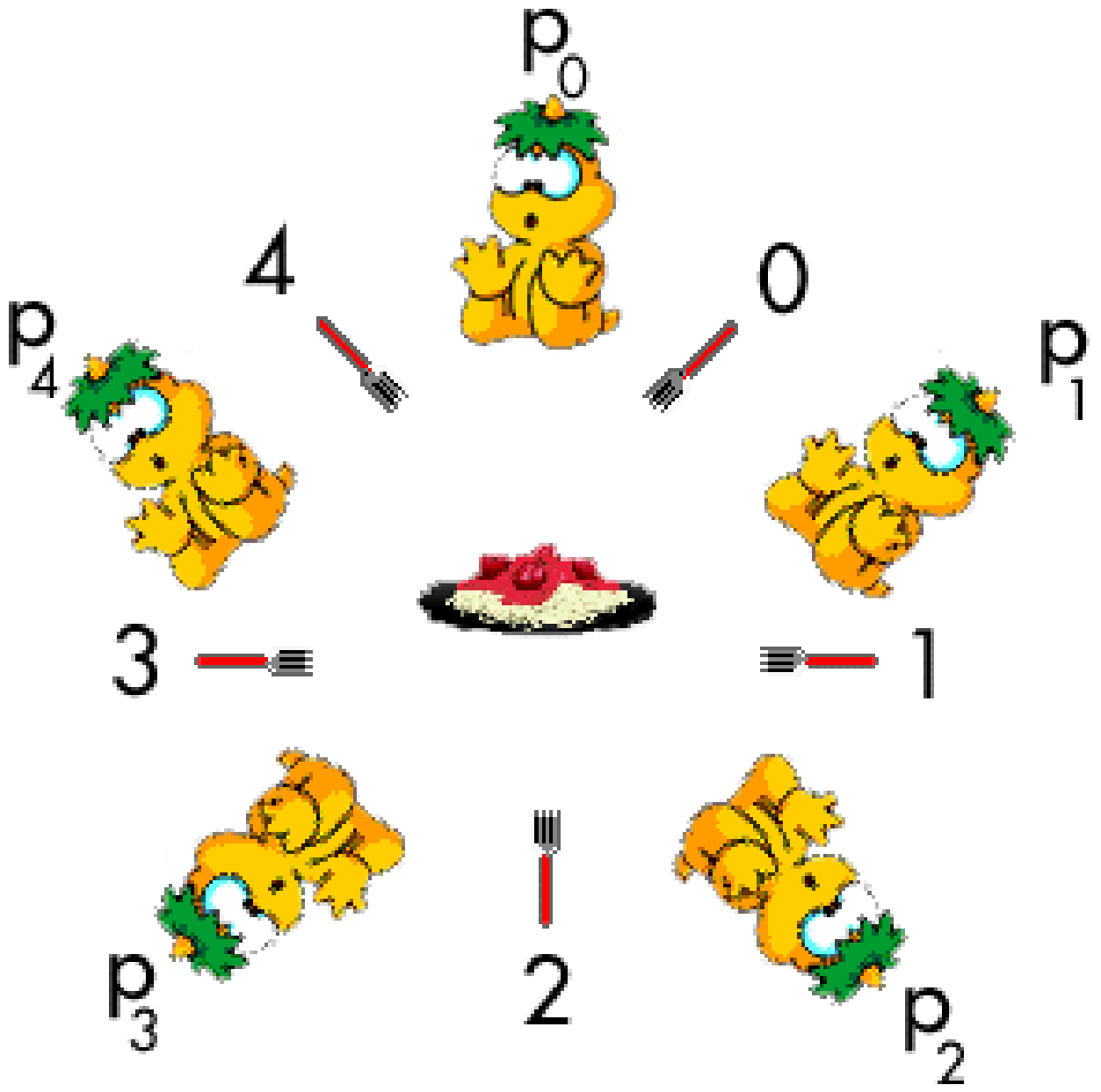
add_ten.bash problem

- The two instances of "add_ten" that are trying to read and write from the same file can do so in any order, and sometimes can read in the middle of another process' write.

- As we increase the number of processes, this happens more and more often. Eventually we always truncate to the first digit.

- So, an obvious fix is to restrict only one process to be able to open the file at a time.
  - Is this enough to solve all process concurrency?

The Dining Philosophers Game: Volunteers Please!



Rules of the game

- Philosophers alternate between thinking for an un-predictable amount of time and eating. They must never starve.
- There is unlimited spaghetti (yay!)
- The problem: eating requires both chopsticks (classically forks, but why eat with 2 forks?), that are placed on the left and right
  - This means not everyone can eat at once: resource constraint!
- Philosophers are solitary people, and cannot communicate with each other in any way
  - The solution must be made only considering the current chop-stick states

Volunteers, please do the following

- Think until the left stick is available; when it is, pick it up;

Volunteers, please do the following

- Think until the left stick is available; when it is, pick it up;
- Think until the right stick is available; when it is, pick it up;
- When both sticks are held, eat for a fixed amount of time;
- Then, put the right stick down;
- Then, put the left stick down;
- Repeat from the beginning.

This solution suffers from deadlock

- Deadlock: The situation where resources are in competition, no progress is being made, but the algorithm does not proceed

Solution concept #2: Timeout

- Same as above, but if you hold only one chopstick and are blocked for 10 seconds, put it down and wait 10 seconds before trying again

- Does it work?

Solution #2 problem: Livelock can occur

- Livelock: means we are changing states (timeout always moves us), but the overall system may not succeed

- If all philosophers start trying to eat at the same moment, they are fully synchronized and we are no better off, just infinitely picking up and putting down forks (and getting hungry!)

The first working solution (by Turing award winner **Edsger Wybe Dijkstra)**

- Number the sticks

- Rather than picking up your left stick first always, pick up the lowest numbered stick (of your options).
  - This blocks at least one of the eaters, preventing them from taking the last stick. Their neighbor can then proceed.

- Since it gets exactly one person eating, each time the eater finishes and puts down their sticks, one other person can proceed.

Another solution

- Ask a waiter if you can eat, wait until they come to serve you to take any forks. Since only one waiter, only one eater tries to eat.

- This is the concept of a "mutex" (mutual exclusion) which we will continue with next time.

- Thanks for participating!