

1. Explore how processes are handled by the cpu

Using your linux virtual machine, install the following: htop, make, code from ostep.

Install htop:

First things first, I run `sudo apt update` and `sudo apt upgrade` to make sure all packages are up to date. Next, we can install htop by running `sudo apt install build-essential manpages-dev htop strace`.

```
[b@ubuntu ~]$ htop --version
htop 3.3.0
```

Looks great, htop is installed with version 3.3.0.

Install make:

Next, I'll run `sudo apt install make`.

```
[b@ubuntu ~]$ make --version
GNU Make 4.3
Built for x86_64-pc-linux-gnu
Copyright (C) 1988-2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
```

Perfect, we're all set with make as well. Onto ostep.

Install ostep code:

We have to install the ostep code from a github repo, so I'll need to git as well. I already have git installed:

```
[b@ubuntu ~]$ git --version
git version 2.43.0
```

Next, I'll navigate to my Github folder, and clone the ostep repo.

```
[b@ubuntu GitHub]$ git clone https://github.com/remzi-arpacidusseau/ostep-code.git
Cloning into 'ostep-code'...
remote: Enumerating objects: 316, done.
remote: Counting objects: 100% (155/155), done.
remote: Compressing objects: 100% (54/54), done.
remote: Total 316 (delta 110), reused 101 (delta 101), pack-reused 161 (from 1)
Receiving objects: 100% (316/316), 54.66 KiB | 4.20 MiB/s, done.
Resolving deltas: 100% (148/148), done.
```

Now that we've cloned the repo, I'll cd into /ostep/intro and `make` to compile the intro c programs.

```
[b@ubuntu intro]$ make
gcc -o cpu cpu.c -Wall
gcc -o mem mem.c -Wall
gcc -o threads threads.c -Wall -pthread
gcc -o io io.c -Wall
```

Great, we can see multiple programs compiled. The next step is to read through the intro chapter of the book and try out the code listed.

./cpu

The first program is `cpu.c`. Let's examine the contents of this program with `nano cpu.c`:

```
#include <stdio.h>
#include <stdlib.h>
#include "common.h"

int main(int argc, char *argv[])
{
    if (argc != 2) {
        fprintf(stderr, "usage: cpu <string>\n");
        exit(1);
    }
    char *str = argv[1];

    while (1) {
        printf("%s\n", str);
        Spin(1);
    }
    return 0;
}
```

This program spins up a cpu that repeats the following:

- Check if 1 second has passed

- If one second has passed, print argument

First, let's test it with `./cpu ben` :

```
[b@ubuntu intro]$ ./cpu ben
ben
ben
ben
ben
ben
^C
```

The program prints my name repeatedly until I kill it with `ctrl+c`.

Next, let's try running multiple programs at once with `./cpu A & ./cpu B` . & will send the process to the background.

```
[b@ubuntu intro]$ ./cpu A & ./cpu B
[1] 8216
B
A
B
A
```

This program repeatedly prints B, then A. This shows how virtualizing a cpu creates the illusion of the cpu doing multiple tasks at once.

While running, we can use `jobs` to see ids of process running and their states.

If I run `ctrl+c` , then `jobs` , we can see `./cpu A` is still running:

```
jobs
[1]+  Running                  ./cpu A &
```

This process isn't killed by `ctrl+c` because it's in the background. To kill it, we must bring it to foreground with `fg 1` .

```
fg 1
./cpu A
```

Now, we can kill it with `ctrl+c`.

Question: How many programs can run in the foreground?

A: With one cpu, only one program can run in the foreground.

./mem

Let's open up mem.c:

```
GNU nano 7.2 mem.c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include "common.h"

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "usage: mem <value>\n");
        exit(1);
    }
    int *p;
    p = malloc(sizeof(int));
    assert(p != NULL);
    printf("(%d) addr pointed to by p: %p\n", (int) getpid(), p);
    *p = atoi(argv[1]); // assign value to addr stored in p
    while (1) {
        Spin(1);
        *p = *p + 1;
        printf("(%d) value of p: %d\n", getpid(), *p);
    }
    return 0;
}
```

The mem.c program uses the `malloc` function to manually allocate memory. I've seen this function previously in my Assembly/Computer Architecture course. When running mem.c, the program:

- Allocates memory
- Prints address of allocated memory
- Puts zero into the first slot of the allocated memory.

This loops every second, and increments the value stored at the saved address, as well as

printing the PID of itself.

Let's test it with `./mem 15` :

```
📦 [b@ubuntu intro]$ ./mem 15
(8271) addr pointed to by p: 0x5620bdc522a0
(8271) value of p: 16
(8271) value of p: 17
(8271) value of p: 18
(8271) value of p: 19
^C
```

We can see the address of the memory we allocated, as well as watch the value of p increment, starting at 15.

What about with `./mem 15 & ./mem 5` ?

```
📦 [b@ubuntu intro]$ ./mem 15 & ./mem 5
[1] 8274
(8275) addr pointed to by p: 0x55d35b1b32a0
(8274) addr pointed to by p: 0x55a6f99162a0
(8275) value of p: 6
(8274) value of p: 16
(8275) value of p: 7
(8274) value of p: 17
(8275) value of p: 8
(8274) value of p: 18
^C
```

We can see each instance of mem running has allocated to separate memory, and takes turns incrementing their value.

htop

The first part of this section is to "start up a few cpus and mem in the background, then run htop in another terminal."

Because I'm running ubuntu in a container, I have to use a terminal multiplexer to have multiple terminal sessions. I'll be using tmux. With tmux, I can create sessions with `ctrl+b` then `c` , and cycle through sessions with `ctrl+b` and `n` for next / `p` for previous. It's pretty cool!

Start up a few cpus and mem in the background

I'll run `./mem 2 & ./cpu A`, then jump into `htop` in my other session.

```
[b@ubuntu intro]$ ./mem 2 & ./cpu A
[1] 8436
A
(8436) addr pointed to by p: 0x55afa48ce2a0
A
(8436) value of p: 3
```

`htop` opens us into a TUI that is a bit overwhelming at first glance:

The screenshot shows the htop interface with system statistics at the top and a list of processes below. The processes list includes `./cpu A` (PID 8437) and `./mem 2` (PID 8436) at the top, indicating they are consuming most of the system resources.

PID	USER	PRI	NI	VIRT	RES	SHR	S	CPU%	MEM%	TIME+	Command
8437	b	20	0	2692	1400	1300	R	99.8	0.0	1:01.18	./cpu A
8436	b	20	0	2692	1668	1572	R	99.2	0.0	1:01.05	./mem 2

We can see `./mem` and `./cpu` are sitting at the top of our process list.

Main		I/O									
PID	USER	PRI	NI	VIRT	RES	SHR	S	CPU%	MEM%	TIME+	Command
8436	b	20	0	2692	1668	1572	R	99.7	0.0	2:11.05	./mem 2
8437	b	20	0	2692	1400	1300	R	99.7	0.0	2:11.15	./cpu A

Let's try a few functions:

sort (f6)

	Main	I/O										
Sort by	PID	USER	PRI	NI	VIRT	RES	SHR	S	CPU%	MEM%	TIME+	Command
PID	8436	b	20	0	2692	1668	1572	R	100.0	0.0	3:31.81	./mem 2
USER	8437	b	20	0	2692	1400	1300	R	100.0	0.0	3:31.86	./cpu A
PRIORITY	1	root	20	0	24596	16060	9720	S	0.0	0.1	0:00.65	/usr/lib/systemd/systemd --system --deserializ
NICE	5550	root	20	0	2724	1932	1816	S	0.0	0.0	0:00.00	script -c cat /var/console /dev/null
M_VIRT	5552	root	20	0	2712	1484	1388	S	0.0	0.0	0:00.00	cat /var/console
M_RESIDENT	5623	root	20	0	66744	17204	16160	S	0.0	0.1	0:00.07	/usr/lib/systemd/systemd-journald
M_SHARE	5638	avahi	20	0	8704	4644	4204	S	0.0	0.0	0:00.13	avahi-daemon: running [bluefin.local]
STATE	5639	messagebus	20	0	9560	5272	4616	S	0.0	0.0	0:00.05	@dbus-daemon --system --address=systemd: --nof
PERCENT_CPU	5641	root	20	0	18108	8556	7448	S	0.0	0.1	0:00.15	/usr/lib/systemd/systemd-logind
PERCENT_MEM	5644	avahi	20	0	8432	1532	1172	S	0.0	0.0	0:00.00	avahi-daemon: chroot helper
TIME	5662	b	20	0	22284	13536	9516	S	0.0	0.1	0:00.12	/usr/lib/systemd/systemd --user
Command	5663	b	20	0	21036	3508	1780	S	0.0	0.0	0:00.00	(sd-pam)

We can see a nice menu of presets to sort by. Personally, I like keeping it on PERCENT_CPU.

tree (f5)

Main	I/O										
PID	USER	PRI	NI	VIRT	RES	SHR	S	CPU%	MEM%	TIME+	Command
1	root	20	0	24596	16060	9720	S	0.0	0.1	0:00.65	/usr/lib/systemd/systemd --system --deserialize=24 --log-targ
5550	root	20	0	2724	1932	1816	S	0.0	0.0	0:00.00	script -c cat /var/console /dev/null
5552	root	20	0	2712	1484	1388	S	0.0	0.0	0:00.00	cat /var/console
5623	root	20	0	66744	17204	16160	S	0.0	0.1	0:00.07	/usr/lib/systemd/systemd-journald
5638	avahi	20	0	8704	4644	4204	S	0.0	0.0	0:00.13	avahi-daemon: running [bluefin.local]
5644	avahi	20	0	8432	1532	1172	S	0.0	0.0	0:00.00	avahi-daemon: chroot helper
5639	messagebus	20	0	9560	5272	4616	S	0.0	0.0	0:00.05	@dbus-daemon --system --address=systemd: --nofork --nopidf
5641	root	20	0	18108	8556	7448	S	0.0	0.1	0:00.15	/usr/lib/systemd/systemd-logind
5662	b	20	0	22284	13536	9516	S	0.0	0.1	0:00.12	/usr/lib/systemd/systemd --user
5663	b	20	0	21036	3508	1780	S	0.0	0.0	0:00.00	(sd-pam)
5681	b	20	0	22284	13424	9404	S	0.0	0.1	0:00.12	/usr/lib/systemd/systemd --user
5682	b	20	0	21040	3612	1880	S	0.0	0.0	0:00.00	(sd-pam)
5775	b	20	0	9440	5132	4684	S	0.0	0.0	0:00.00	/usr/bin/dbus-daemon --session --address=systemd: --nof

tree shows us a parent/child tree of which processes spawned which.

filter (f4)

Main

I/O

PID	USER	PRI	NI	VIRT	RES	SHR	S	CPU%	MEM%	TIME+	Command
8437	b	20	0	2692	1400	1300	R	99.8	0.0	10:57.04	└─ ./cpu A

Enter

Done

Esc

Clear

Filter: cpu

Next, I use `Kill (f9)` to send `SIGTERM` 15s to both `cpu` and `mem`, killing the programs without messing with `fg` and `jobs` .

./Fork

Now, we'll take a look at `fork.c`.

```
GNU nano 7.2                                     fork.c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main() {
    pid_t pid = fork();

    if (pid < 0) {
        perror("fork failed");
        return 1;
    } else if (pid == 0) {
        // Child
        printf("Child: PID = %d, Parent PID = %d\n", getpid(), getppid());
    } else {
        // Parent
        printf("Parent: PID = %d, Child PID = %d\n", getpid(), pid);
    }
    return 0;
}
```

We can see this is a simple program to fork a process, and display child/parent PIDs.

`gcc fork.c -o fork && ./fork`

```
📦[b@ubuntu Downloads]$ gcc fork.c -o fork && ./fork
Parent: PID = 8500, Child PID = 8501
Child: PID = 8501, Parent PID = 8500
📦[b@ubuntu Downloads]$ gcc fork.c -o fork && ./fork
Parent: PID = 8507, Child PID = 8508
Child: PID = 8508, Parent PID = 1
📦[b@ubuntu Downloads]$ gcc fork.c -o fork && ./fork
Parent: PID = 8514, Child PID = 8515
Child: PID = 8515, Parent PID = 1
```

We can see after the first run, the children show a Parent PID of 1, meaning the parent has been terminated.

./fork_exec

Let's take a look at `fork_exec`.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    pid_t pid = fork();

    if (pid == 0) {
        // Child replaces itself with ls
        printf("Child (PID %d) about to exec ls...\n", getpid());
        execl("/bin/ls", "ls", "-l", NULL);
        perror("execl failed"); // only reached if exec fails
        _exit(1);
    } else if (pid > 0) {
        wait(NULL); // parent waits for child
        printf("Parent: child finished.\n");
    }
    return 0;
}
```

We can see this will repeatedly spawn a child that calls `ls -l`, and the parent prints `Parent: child finished` once `ls -l` runs successfully.

Let's use `zombie.c` to look for zombie processes in `htop`.

```
[b@ubuntu Downloads]$ gcc Zombie.c -o Zombie
[b@ubuntu Downloads]$ ./Zombie
Parent (PID 8549) sleeping for 60 seconds...
Child (PID 8550, parent 8549) exiting now...
```

In `htop`:

Main		I/O											
PID	USER	PRI	NI	VIRT	RES	SHR	S	CPU%	MEM%	TIME+	Command		
1	root	20	0	24596	16060	9720	S	0.0	0.1	0:00.66	/usr/lib/systemd/systemd -		
8551	b	20	0	2692	1668	1576	S	0.0	0.0	0:00.00	└─ ./Zombie		
8552	b	20	0	0	0	0	Z	0.0	0.0	0:00.00	└─ Zombie		

We can see that the child has a "Z" state, showing that it's a zombie child. Eventually, the zombie process gets cleaned up.

Now, let's observe an orphan process get reparented:

```
Child starting (PID 8561, initial parent 8560)
[b@ubuntu Downloads]$ Child waking up (PID 8561, new parent now 1)
Child exiting.
```

Main		I/O													
PID	USER	PRI	NI	VIRT	RES	SHR	S	CPU%	MEM%	TIME+				Command	
5639	messagebus	20	0	9560	5272	4616	S	0.0	0.0	0:00.05				@dbus-daemon --system --address=systemd: --nofork --nopidf	
5775	b	20	0	9440	5132	4684	S	0.0	0.0	0:00.00				/usr/bin/dbus-daemon --session --address=systemd: --nof	
8561	b	20	0	2692	1004	904	S	0.0	0.0	0:00.00				./orphan	

./cpu_burn

- Detects how many CPU cores you have
- Spawns one pthread per core
- Each thread runs a tight infinite loop doing math

```
[b@ubuntu Downloads]$ gcc cpu_burn.c -o cpu_burn -pthread -lm
[b@ubuntu Downloads]$ ./cpu_burn
Starting 12 CPU burner threads
```

Here we go....

We can see almost immediately, my CPU usage hits ~100% for each process for ./cpu_burn

Thu Feb 5 4:29 PM

b@bluefin: ~/Documents/GitHub/ostep-code/intro

0[|||||97.0%] 3[|||||96.9%] 6[|||||97.7%] 9[|||||97.8%]
1[|||||96.9%] 4[|||||97.8%] 7[|||||97.7%] 10[|||||97.7%]
2[|||||97.7%] 5[|||||97.7%] 8[|||||98.5%] 11[|||||97.7%]
Mem[|||||5.616/15.36] Tasks: 23, 12 thr, 0 kthr; 12 running
Swp[| 272K/8.006] Load average: 6.70 2.08 0.89
Uptime: 06:21:58

Main		I/O													
PID	USER	PRI	NI	VIRT	RES	SHR	S	CPU%	MEM%	TIME+				Command	
5761	b	20	0	2812	1876	1768	S	0.0	0.0	0:00.00				/bin/sh -c \$(getent passwd 'b' cut -f 7 -d :) -l	
5765	b	20	0	13208	6272	4444	S	0.0	0.0	0:00.10				/usr/bin/bash -l	
5775	b	20	0	9440	5132	4684	S	0.0	0.0	0:00.00				/usr/bin/dbus-daemon --session --address=systemd: --nofork --	
8406	b	20	0	14516	5320	4612	S	0.0	0.0	0:00.00				tmux	
8408	b	20	0	16416	6284	3696	S	0.0	0.0	0:01.25				tmux	
8409	b	20	0	12308	6128	3760	S	0.0	0.0	0:00.07				-bash	
8421	b	20	0	11660	5448	3728	S	0.0	0.0	0:00.00				-bash	
8523	b	20	0	11316	4796	3788	R	0.0	0.0	0:01.47				htop	
8568	b	20	0	99.5M	2124	1912	S	0.0	0.0	0:00.00				./cpu_burn	
8569	b	20	0	99.5M	2124	1912	R	98.7	0.0	0:48.70				./cpu_burn	
8570	b	20	0	99.5M	2124	1912	R	100.3	0.0	0:48.70				./cpu_burn	
8571	b	20	0	99.5M	2124	1912	R	101.8	0.0	0:48.72				./cpu_burn	
8572	b	20	0	99.5M	2124	1912	R	101.8	0.0	0:48.74				./cpu_burn	
8573	b	20	0	99.5M	2124	1912	R	101.8	0.0	0:48.68				./cpu_burn	
8574	b	20	0	99.5M	2124	1912	R	101.0	0.0	0:48.70				./cpu_burn	
8576	b	20	0	99.5M	2124	1912	S	101.0	0.0	0:48.82				./cpu_burn	
8577	b	20	0	99.5M	2124	1912	R	100.3	0.0	0:48.77				./cpu_burn	
8575	b	20	0	99.5M	2124	1912	R	100.3	0.0	0:48.75				./cpu_burn	
8578	b	20	0	99.5M	2124	1912	R	101.0	0.0	0:48.69				./cpu_burn	
8579	b	20	0	99.5M	2124	1912	R	100.3	0.0	0:48.74				./cpu_burn	
8580	b	20	0	99.5M	2124	1912	R	101.8	0.0	0:48.72				./cpu_burn	

F1Help F2Setup F3Search F4Filter F5Tree F6SortBy F7Nice -F8Nice +F9Kill F10Quit

[0] 0:./cpu_burn- 1:htop*

"bluefin" 21:29 05-Feb-26

After letting cpu_burn run for a bit, I sent SIGTERM 15s to ./cpu_burn, and my usage returned to normal.

```

Thu Feb 5 4:30 PM
b@bluefin:~/Documents/GitHub/ostep-code/intro
~/Documents/GitHub/ostep-code/intro

0[| 0.6%] 3[ 0.0%] 6[| 1.7%] 9[ 0.0%]
1[| 0.6%] 4[| 2.3%] 7[ 0.0%] 10[| 0.6%]
2[| 4.6%] 5[| 1.8%] 8[| 1.8%] 11[ 0.0%]
Mem[|||||] 5.67G/15.3G Tasks: 21, 0 thr, 0 kthr; 1 running
Swp[| 272K/8.00G] Load average: 3.51 2.05 0.95
Uptime: 06:22:53

Main I/O
PID USER PRI NI VIRT RES SHR S CPU%MEM% TIME+ Command
1 root 20 0 24596 16060 9720 S 0.0 0.1 0:00.67 /usr/lib/systemd/systemd --system --deserialize=24 --log-targ
5550 root 20 0 2724 1932 1816 S 0.0 0.0 0:00.00 script -c cat /var/console /dev/null
5552 root 20 0 2712 1484 1388 S 0.0 0.0 0:00.00 cat /var/console
5638 avahi 20 0 8704 4644 4204 S 0.0 0.0 0:00.18 avahi-daemon: running [bluefin.local]
5639 messagebus 20 0 9560 5272 4616 S 0.0 0.0 0:00.05 @dbus-daemon --system --address=systemd: --nofork --nopidfile
5641 root 20 0 18108 8556 7448 S 0.0 0.1 0:00.15 /usr/lib/systemd/systemd-logind
5644 avahi 20 0 8432 1532 1172 S 0.0 0.0 0:00.00 avahi-daemon: chroot helper
5662 b 20 0 22284 13536 9516 S 0.0 0.1 0:00.12 /usr/lib/systemd/systemd --user
5663 b 20 0 21036 3508 1780 S 0.0 0.0 0:00.00 (sd-pam)
5681 b 20 0 22284 13424 9404 S 0.0 0.1 0:00.12 /usr/lib/systemd/systemd --user
5682 b 20 0 21040 3612 1880 S 0.0 0.0 0:00.00 (sd-pam)
5759 root 20 0 11996 4732 4264 S 0.0 0.0 0:00.24 su b -m --pty -s /bin/sh -c "$@" "$@" -- /bin/sh -c $(getent
5760 b 20 0 2812 1848 1744 S 0.0 0.0 0:00.00 sh -c "$@" "$@" /bin/sh -c $(getent passwd 'b' | cut -f 7 -d
5761 b 20 0 2812 1876 1768 S 0.0 0.0 0:00.00 /bin/sh -c $(getent passwd 'b' | cut -f 7 -d :) -l
5765 b 20 0 13208 6272 4444 S 0.0 0.0 0:00.10 /usr/bin/bash -l
5775 b 20 0 9440 5132 4684 S 0.0 0.0 0:00.00 /usr/bin/dbus-daemon --session --address=systemd: --nofork --
8406 b 20 0 14516 5320 4612 S 0.0 0.0 0:00.00 tmux
8421 b 20 0 11660 5448 3728 S 0.0 0.0 0:00.00 -bash
8523 b 20 0 11240 4796 3788 R 0.0 0.0 0:01.53 htop
5623 root 20 0 66744 17204 16160 S 0.0 0.1 0:00.08 /usr/lib/systemd/systemd-journald
8408 b 20 0 16276 6200 3696 S 0.0 0.0 0:01.27 tmux
F1Help F2Setup F3Search F4Filter F5Tree F6SortBy F7Nice -F8Nice +F9Kill F10Quit
[0] 1:htop* "bluefin" 21:30 05-Feb-26

```

Now modify the code to run with:

- Less threads than cores
- Number varies based on your vm setup, ex. You have 5 cores and set to 2
- An overload of threads
- `int cores = sysconf(SC_NPROCESSORS_ONLN)`

Let's open up `cpu_burn.c`:

```

GNU nano 7.2                                                                    cpu_b
int main() {
    int cores = sysconf(_SC_NPROCESSORS_ONLN);
    printf("Starting %d CPU burner threads\n", cores);

    pthread_t* threads = malloc(sizeof(pthread_t) * cores);

    for (int i = 0; i < cores; i++) {
        pthread_create(&threads[i], NULL, burn, NULL);
    }

    // Prevent main thread from exiting
    for (int i = 0; i < cores; i++) {
        pthread_join(threads[i], NULL);
    }

    free(threads);
    return 0;
}

```

For *less threads than cores*, I'll simply change the for loop to execute 4 less times than the amount of cores we have.

```

for (int i = 0; i < cores - 4; i++) {
    pthread_join(threads[i], NULL);
}

```

Next, for *overload of threads* i'll change:

```

int main() {
    int cores = sysconf(_SC_NPROCESSORS_ONLN) * 4;
}

```

So, we're creating more threads than cores we have available. Let's run `cpu_burn` again!

```
Thu Feb 5 4:44 PM
b@bluefin:~/Documents/GitHub/ostep-code/intro
~/Documents/GitHub/ostep-code/intro

0[|||||99.3%] 3[|||||99.3%] 6[|||||100.0%] 9[|||||100.0%]
1[|||||99.3%] 4[|||||100.0%] 7[|||||100.0%] 10[|||||100.0%]
2[|||||100.0%] 5[|||||100.0%] 8[|||||99.3%] 11[|||||100.0%]
Mem[|||||5.296/15.36] Tasks: 23, 44 thr, 0 kthr; 12 running
Swp[|||||272K/8.00G] Load average: 15.37 3.92 1.72
Uptime: 06:37:22

Main I/O
PID USER PRI NI VIRT RES SHR S CPU% MEM% TIME+ Command
8676 b 20 0 355M 2416 1940 R 27.4 0.0 0:05.70 ./cpu_burn
8679 b 20 0 355M 2416 1940 R 27.4 0.0 0:05.61 ./cpu_burn
8683 b 20 0 355M 2416 1940 R 26.7 0.0 0:05.66 ./cpu_burn
8684 b 20 0 355M 2416 1940 R 26.7 0.0 0:05.62 ./cpu_burn
8685 b 20 0 355M 2416 1940 R 27.4 0.0 0:05.65 ./cpu_burn
8696 b 20 0 355M 2416 1940 R 27.4 0.0 0:05.67 ./cpu_burn
8665 b 20 0 355M 2416 1940 R 26.7 0.0 0:05.65 ./cpu_burn
8668 b 20 0 355M 2416 1940 R 27.4 0.0 0:05.70 ./cpu_burn
8670 b 20 0 355M 2416 1940 R 27.4 0.0 0:05.77 ./cpu_burn
8671 b 20 0 355M 2416 1940 R 27.4 0.0 0:05.67 ./cpu_burn
8672 b 20 0 355M 2416 1940 R 26.7 0.0 0:05.69 ./cpu_burn
8673 b 20 0 355M 2416 1940 R 27.4 0.0 0:05.68 ./cpu_burn
8675 b 20 0 355M 2416 1940 R 27.4 0.0 0:05.71 ./cpu_burn
8677 b 20 0 355M 2416 1940 R 27.4 0.0 0:05.67 ./cpu_burn
8678 b 20 0 355M 2416 1940 R 26.7 0.0 0:05.65 ./cpu_burn
8681 b 20 0 355M 2416 1940 R 26.7 0.0 0:05.67 ./cpu_burn
8682 b 20 0 355M 2416 1940 R 27.4 0.0 0:05.64 ./cpu_burn
8687 b 20 0 355M 2416 1940 R 26.7 0.0 0:05.66 ./cpu_burn
8688 b 20 0 355M 2416 1940 R 27.4 0.0 0:05.69 ./cpu_burn
8689 b 20 0 355M 2416 1940 R 27.4 0.0 0:05.68 ./cpu_burn
8690 b 20 0 355M 2416 1940 R 26.7 0.0 0:05.66 ./cpu_burn
F1Help F2Setup F3Search F4Filter F5Tree F6SortBy F7Nice F8Nice F9Kill F10Quit
[0] 1:./cpu_burn- 2:htop* "bluefin" 21:44 05-Feb-26
```

This time, each instance of `./cpu_burn` is only taking up ~27% of CPU, but we can see at the top all 12 of my cores are maxed out with usage. This means that even if you distribute high workload across many threads, you can still burn through resources quickly if your cores are limited.

Conclusion

Overall, this was a great way to get some experience with `htop`, and get a glimpse into what goes on under the hood with processes, threads, and cores.