Part I Introduction

General Information

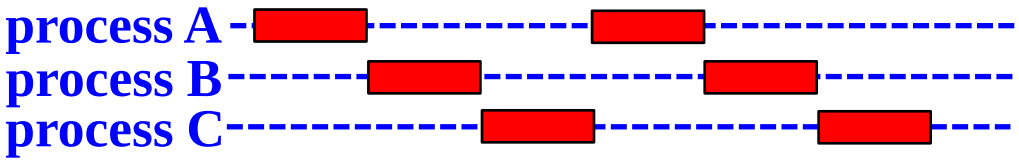
It takes a really bad school to ruin a good student
and
a really fantastic school to rescue a bad student.

Spring 2021

Dennis J. Frailey

Concurrent vs. Parallel: 1/3

- The execution of processes is said to be
  - **interleaved** , if all processes are *in progress* but not all of them are *running*;



  - **parallel** , if they are all *running* at the same time;

  - **Concurrent**, if it is interleaved or parallel.

Concurrent vs. Parallel: 2/3

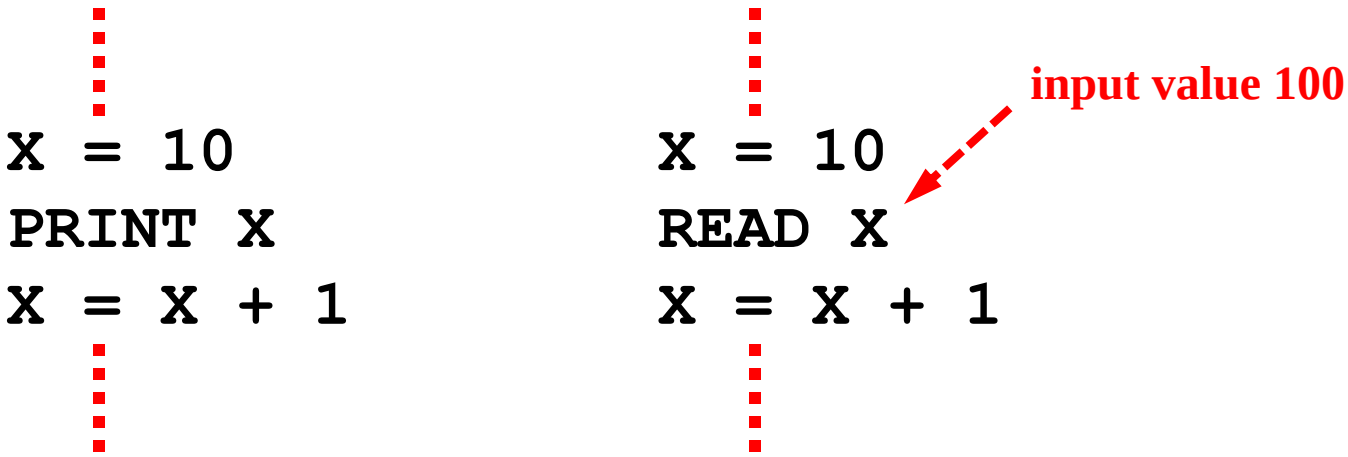
- A **parallel** program may have a number of processes or threads, each of which runs on a CPU/core. All execute at the same time.
- A parallel program needs multiple CPU/cores; but, a concurrent program may only need **ONE**.
- **Concurrent** is more general than **Parallel**!
- **Why?** We may write a concurrent program with multiple processes or threads. If we have enough number of CPUs, all processes or threads can run in parallel. Otherwise, the OS assigns each process/thread some CPU time in turn so that they can finish their job bit-by-bit.

Concurrent vs. Parallel: 3/3

- **This world is actually very concurrent.**
- **You are eating while watching TV.**
- **You listen to music, send e-mail, talk to your buddy at the same time.**
- **You text and drive at the same time. This is a dangerous concurrent system because due to interleaved execution you won't be able to pay attention to the traffic all the time.**
- **Think about some concurrent stuffs in a computer system……**

Pause and Think ……

- **Will your program's execution be suspended when it is doing input or output?**



Some Historical Remarks: 1/6

- It all started with operating systems design.
- If your program is doing input and output, it cannot run until the I/O completes. **Why?**
- If there is only one program in the system, the CPU is idle when this program is doing I/O.
- Decades ago, I/O devices were very slow although CPUs were not fast either (but very expensive).
- Therefore, once the program starts doing I/O, we are wasting our money!

N. Rochester, The computer and its peripheral equipment, *Proc. Eastern Joint Computer Conf.*, Boston, MA, pp. 64-69, November, 1955.

Some Historical Remarks: 2/6

- Then, why don't we run a second program while the first one is doing I/O?
- Thus, program 1 does I/O & program 2 computes, program 2 does I/O & program 1 computes, etc.
- The system has **TWO** programs running, each of which has some progress at any time. **Isn't this the concept of concurrency**?
- If we can run two programs, why don't we run more? But, wait a minute! The power of a CPU is limited and is not able to run too many programs!

Some Historical Remarks: 3/6

- In the 1960's, operating systems could run multiple programs (i.e., multiprogramming).
- If a system could run several programs at the same time, why don't we split a program into multiple pieces (i.e., processes or threads) so that they run concurrently.
- So, systems can run multiple programs and programs can have multiple processes/threads.
- Concurrent programming is born.
- This is what we are going to talk about in this semester.

Some Historical Remarks: 4/6

- In the early 1960's, a number of higher-level programming languages supported concurrency.
- PL/I F and ALGOL 68 were among the first.
- Then, we had Modula 2, followed by Modula 3, Ada, Concurrent Euclid, Turing Plus, etc. No, I did not forget Java; but, it is a late comer. The new C++ standard also supports concurrency.
- Systems also provided system calls and libraries to support concurrent programming.
- Concurrent programming was booming in the 1990's.

Some Historical Remarks: 5/6

- **Programmers may be excited about concurrency. But, the picture is not that rosy because splitting a program into multiple processes or threads is easily said than done.**
- **Processes and threads must communicate with each other to get the job done. Once there are communications there are troubles. (What if I missed your call asking for some data, to continue or not to continue becomes your big question.)**
- **This is synchronization. I am sure many of you will hate me when we talk about it.**

Some Historical Remarks: 6/6

- Not only synchronization is a headache, splitting a program improperly would just make the program more inefficient.
- This requires a new mindset to design good concurrent programs. **So, be prepared.**
- The behavior of concurrent programs is **dynamic**. A bug may not surface until our grader is grading your programs. Even if it appears at this time, it may not occur when you run the program again. Or it may never occur!
- No debugger can catch dynamic bugs completely.

First Taste of Concurrency: 1/7

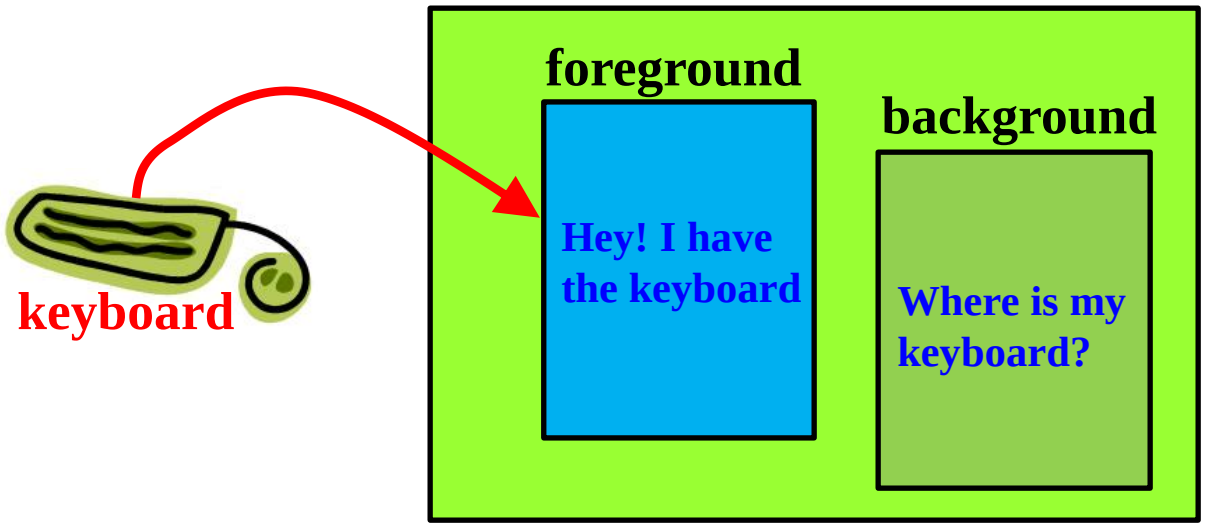
- **Actually, you know it and do it every day.**
- **You sit down in front of a computer and open multiple windows in each of which you run an application (i.e., web browser, editor, e-mail).**
- **This is concurrency!**
- **Let us look at a very simple example.**

First Taste of Concurrency: 2/7

- Consider the Unix command line operator `&`.
- Is `&` the bit-wise and operator? No, it is not! It runs a program in the background.
- When your program runs, it becomes a process. Don't worry about its actual meaning as we will explain it in great detail very soon.
- This process takes its input from the keyboard and waits until the input becomes available.
- You won't be able to issue shell commands because the keyboard is now the `stdin` of your program.

First Taste of Concurrency: 3/7

- By running a process in the **background**, it means the window from which you run the program is detached from the process, and command line input becomes available.
- The process has the command line input is said to be in the **foreground**.



First Taste of Concurrency: 4/7

- Running a program with `&` puts that program in the background.

```
a.out &
```

- The above runs `a.out` in the background. The command line is available immediately.
- You may use `&` as many times as possible. The program before each `&` runs in the background.

```
a.out & dumb-prog & smart
```

- `a.out` and `dumb-prog` are in the background, while `smart` is in the foreground.
- So, **they run concurrently**!

First Taste of Concurrency: 5/7

procA.c

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

#define  LIMIT  (20)    // run this number of iterations

int main(void)
{
     int i, j, x, y;

     srand(time(NULL));  // plant a random number seed
     for (j = 1; j <= LIMIT; j++) {
        x = rand()/10; // get a random number and scale
         for (i = 1; i <= x; i++)
             y = rand();  // just waste CPU time, :o)
         printf("Hi, A here!  Random number = %d\n", x);
     }
     printf("A completes\n");
}
```

waste some CPU time

First Taste of Concurrency: 6/7

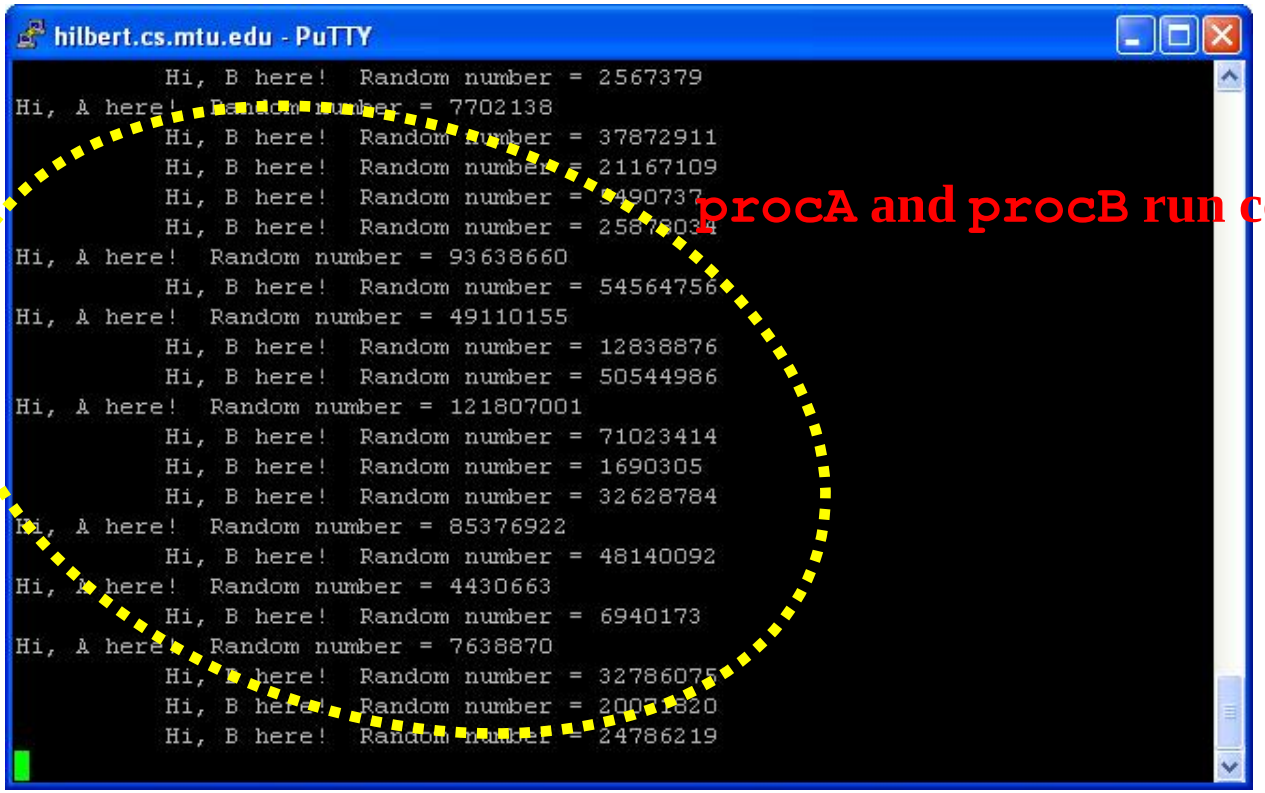
procB.c

```
#include <stdio.h>
#include <stdlib.h>pro

#define  LIMIT (20)

int main(void)
{
     int i, j, x, y;

     srand(time(NULL));
     for (j = 1; j <= LIMIT; j++) {
          x = rand()/30;    // scaled differently
          for (i = 1; i <= x; i++)
               y = rand();
          printf("  Hi, B here!  Random number = %d\n", x);
     }
     printf("           B completes\n");
}
```

Random number `x` is scaled differently (only 1/3 of the one in `procA.c`)
Thus, `procB` prints faster

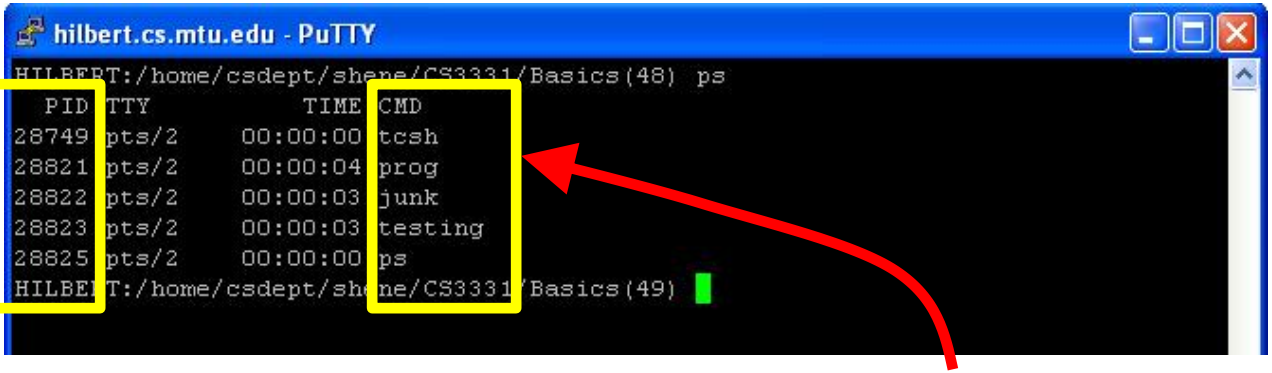
First Taste of Concurrency: 7/7

- Run them with `procA & procB`
- Which one is in the foreground/background?



Let Us Investigate Further

- Since programs become processes when they run, how many processes do I have?
- The Unix `ps` command reports process status.
- `ps` without arguments reports **your** processes.
- `ps` may use other arguments to get a full report that includes **all** processes running concurrently.

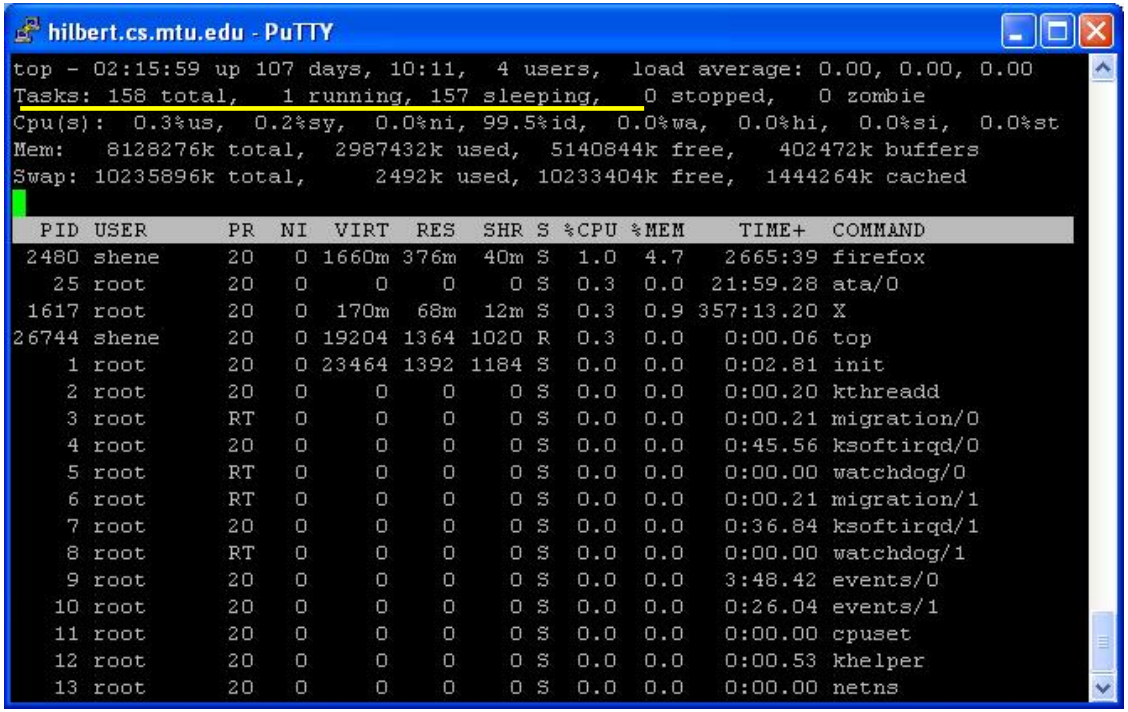


These are the assigned process IDs

These are the program names

Who Is the Top CPU Hog?

- The `top` command is a system monitor tool, which shows and frequently updates system resource usage, usually sorted by percentage of CPU usage.



158 processes
1 running
157 sleeping
Top CPU usage: firefox, 1%

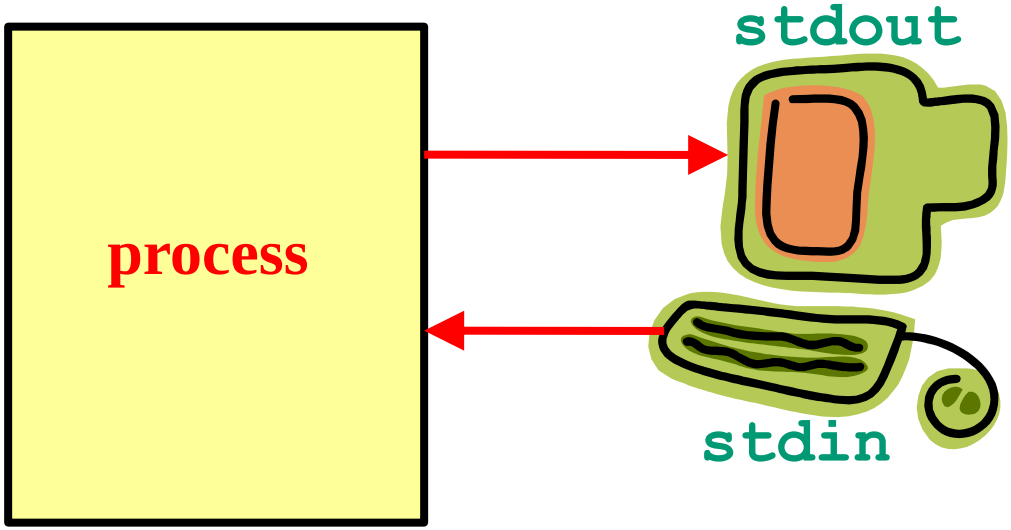
These processes are run concurrently

Some Cooperation: 1/10

- Previous examples have processes running **independent of each other** (i.e., they do not need any help from each other).
- They are **independent processes**.
- If processes must communicate with each other to complete a task, they become cooperative (i.e., **cooperating processes**).
- Independent processes are easy to handle; however, cooperating processes require a careful planning for **synchronization**.

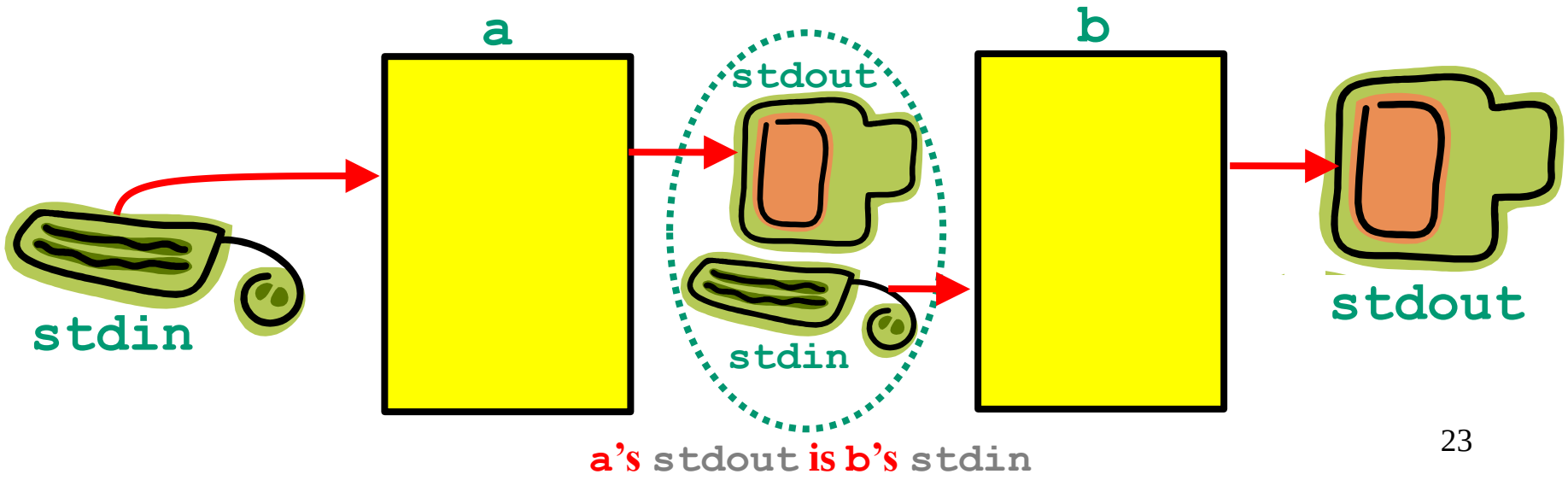
Some Cooperation: 2/10

- Here is a very simple cooperating processes example.
- This is the Unix pipe operator `|`.
- A program has its default I/O: `stdin` (keyboard), `stdout` (screen), and `stderr`.



Some Cooperation: 3/10

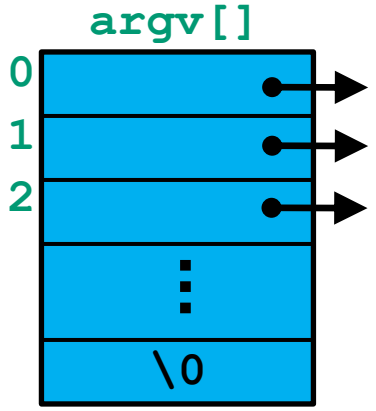
- If you use `a | b`, where `a` and `b` are two programs, the `stdout` of `a` becomes the `stdin` of `b`.
- In this way, `a` takes input from its `stdin`, sends output to `b`, and `b` prints to `b`'s `stdout`.



Some Cooperation: 4/10

pA.c

```
#include <stdio.h>

int main(int argc, char **argv[])
{
      int   i, LIMIT;
      char  output[100];

      LIMIT = atoi(argv[1]);  // read command line argument
      printf("%d\n", LIMIT);  // print # of lines
      for (i = 1; i <= LIMIT; i++) {  // print the lines
            sprintf(output, "Printing %d from A", i);
            printf("%s\n", output);
      }
}
```



print to a character string

read an `int` from command line and print that number of lines

Some Cooperation: 5/10

pB.c

```
#include  <stdio.h>

int  main(void)
{
     int   i, LIMIT;
     char  input[100];

     gets(input);   // read a complete input line
     LIMIT = atoi(input); // convert to integer
     for (i = 1; i <= LIMIT; i++) {  // repeat
          gets(input); // read a complete input line
          printf("     From B: %s\n", input);
     }
}
```

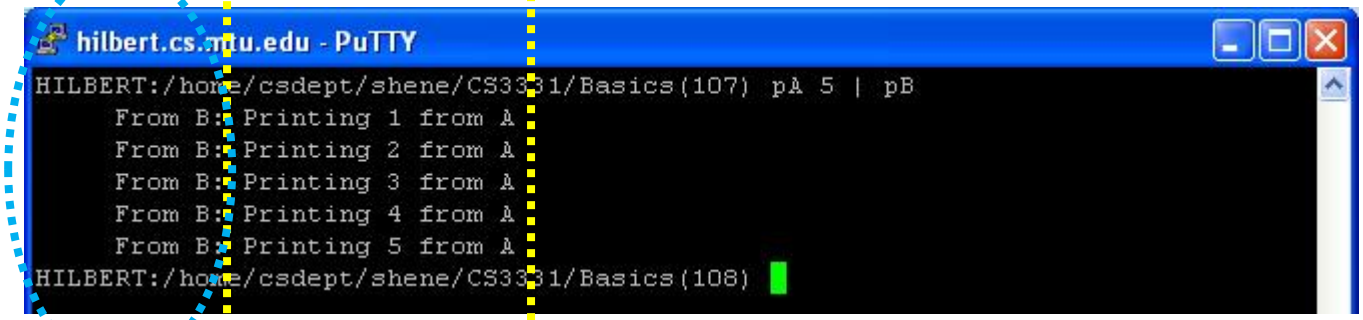
The first `int` gives the number of lines to be read

`gets()` is a bit risky as it does not have a bound.

Some Cooperation: 6/10

The following shows the result of `pA 5 | pB`

print 5 lines from `pA`



`pB` adds this portion

`pB` receives this portion from `pA`

Some Cooperation: 7/10

pC.c

```
#include  <stdio.h>

int  main(void)
{
     int   i, LIMIT = 100;
     char  input[100];

     // now we use a better way: fgets()
     // keep reading until EOF
     while (fgets(input, LIMIT, stdin) != NULL)
          printf("From C: %s", input);
}
```

fgets() is safer than gets()

input char[]

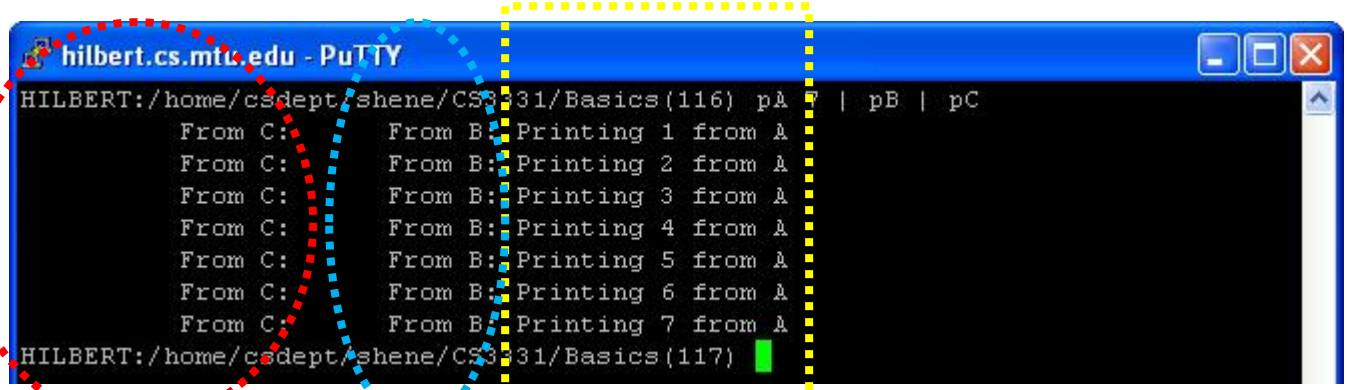
max length including \0

NULL means EOF

Some Cooperation: 8/10

The following shows the result of `pA 7 | pB | pC`

print 7 lines from `pA`



`pC` adds this portion

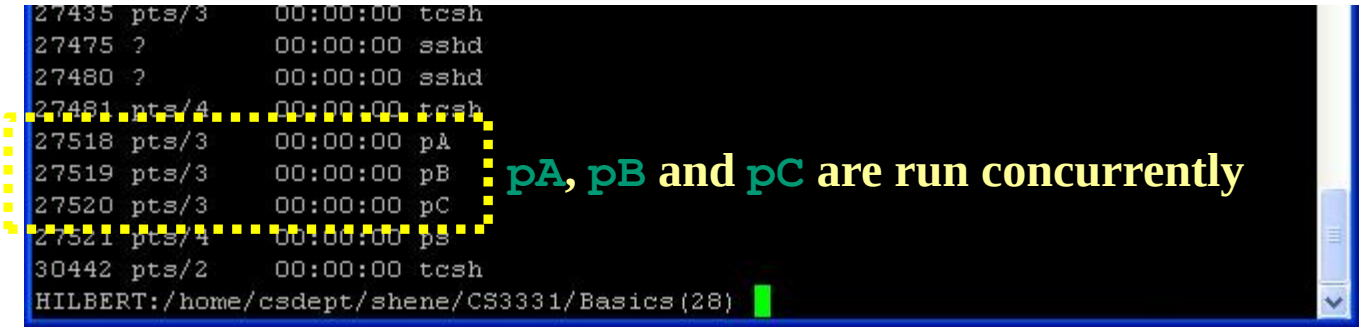
`pB` adds this portion

`pB` receives this portion from `pA`

Some Cooperation: 9/10

- Processes `pA`, `pB` and `pC` run concurrently.
- The following is a screenshot of the `ps` command `ps -A` while `pA 10000 | pB | pC` is in execution.

show all processes



Some Cooperation: 10/10

- Since `pB` depends on `pA`'s output, and `pC` depends on `pB`'s output, `pA`, `pB` and `pC` are cooperating processes, and they communicate via their "hooked" `stdin`s and `stdout`s, even though this type of communication is very simple.
- We will see more complex communication techniques among processes and threads soon.

Few Extra Unix Commands: 1/2

- **Ctrl-Z**: Suspend the foreground process and return to the shell (i.e., command line)
- `bg`: Run the most recently suspended process in the background.

```
prog        // run program prog
Ctrl-Z      // suspend prog
bg          // resume prog in the background
```

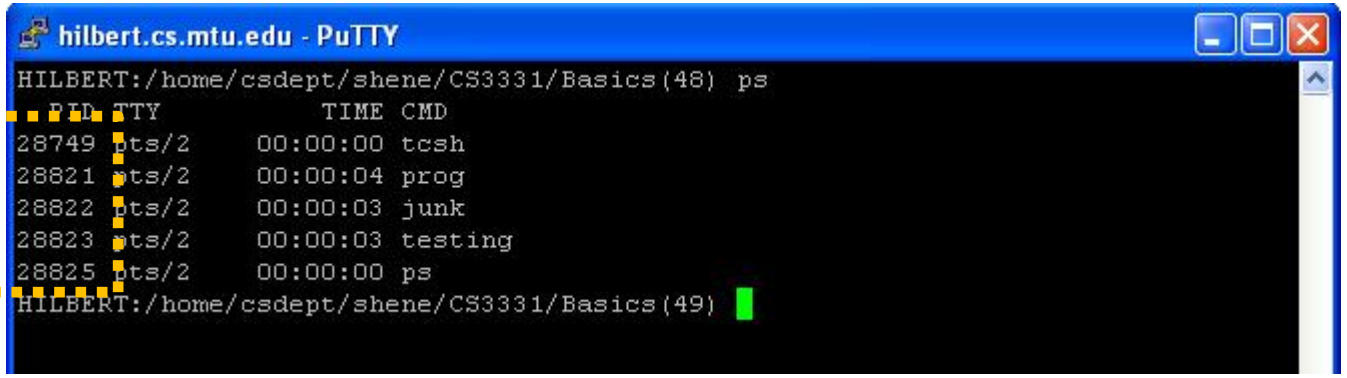
- `fg`: Bring the most recent background process to foreground.

```
fg          // prog  in the foreground
```

Few Extra Unix Commands: 2/2

- Each process has a process ID assigned by the OS.

process ID



- To terminate processes, use

  `kill -KILL pid1 pid2 … pid`*i*

- `kill -KILL 28821 28823` terminates `prog` and `testing`.
- Note that `KILL` is actually `9`. I prefer `KILL`.

I/O Redirection: 1/3

- A program may read input from a text file instead of `stdin` (i.e., redirecting input).
- Similarly, a program may print output to a text file instead of `stdout` (i.e., redirecting output).
- `prog < data`: program `prog` reads input from file `data`. File `data` must exist before `prog` is run.
- `prog > report`: program `prog` prints output to file `report`.

I/O Redirection: 2/3

- **What is the difference between the following:**

```
pA | pB
```

and

```
pA > temp-file
pB < temp-file
```

- **The first version has `pA` and `pB` running concurrently, while the second is not.**
- **Since input file `temp-file` of `pB` must exist before `pB` runs, `pB` must wait until `pA` finishes its work.**

I/O Redirection: 3/3

- Both `stdin` and `stdout` may be redirected.
- In the following, `prog` reads input from file `data` and prints output to file `report`:

```
prog < data > report
```

The End