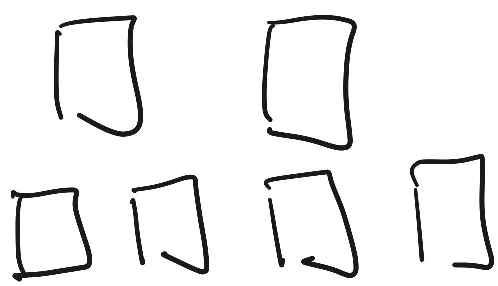
- [x] 1. Last time
- [x] 2. Process birth
- [x] 3. The shell, part I
- [x] 4. File descriptors
- [x] 5. The shell, part II
- [x] 6. Processes: the OS's view
- [x] 7. Threads
- [ ] 8. Intro to Concurrency

---

## 2. Process birth

```
fork();
```

```
for (i = 0; i < 10; i++) {
    fork();
}
while (1) {}
```



---

## 3. The shell

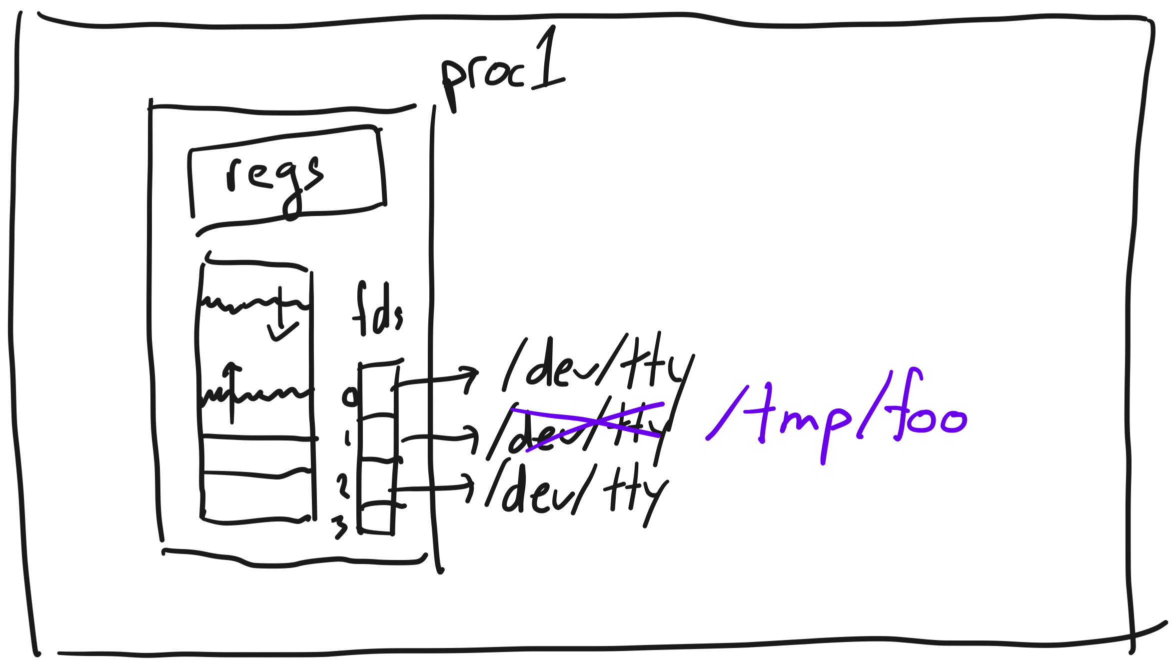
- A program that creates processes
- The human's interface to the computer

Eg:

Ex

$ ls
>

## 4. File descriptors



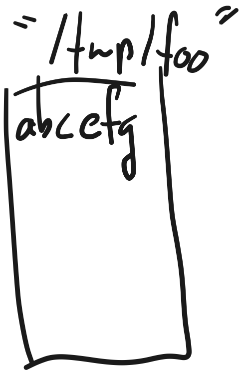

fd = open( ) - - -

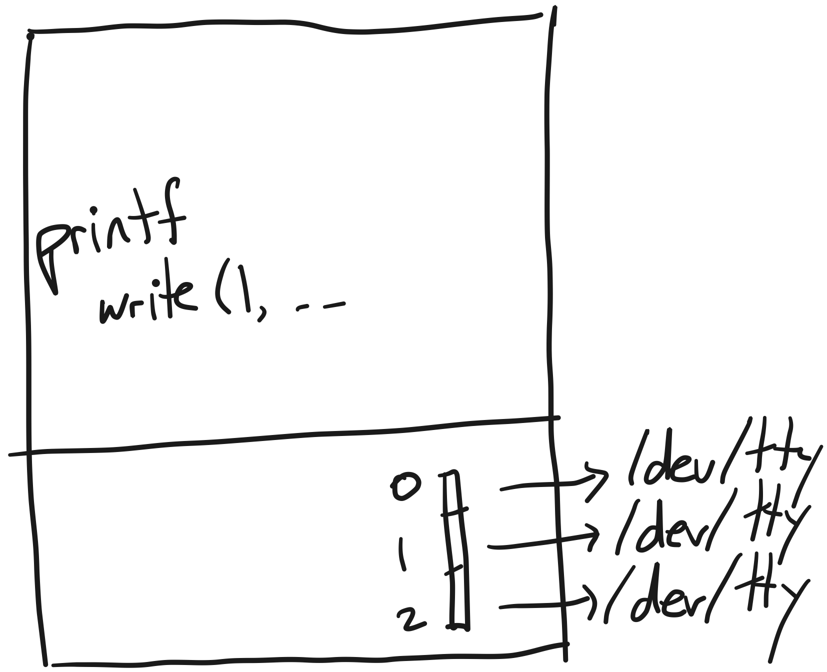
0 stdin
1 : stdout
2 : stderr

fprintf(stderr, . - - - )

## 5. Shell part II

$ ./first3 abcd efgh > /tmp/foo



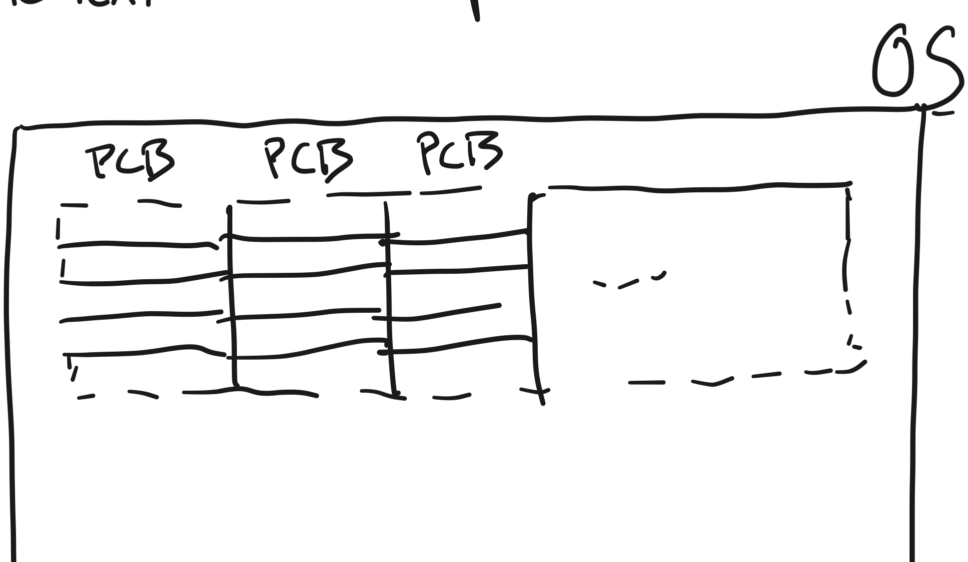



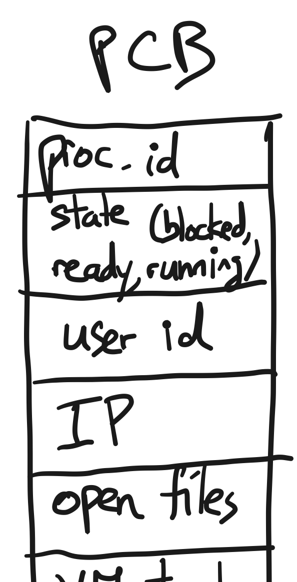

CreateProcess (name, commandline, security_attr, ..., ..., ..., ..., ..., ...);

$ :(){:|:&};:

---

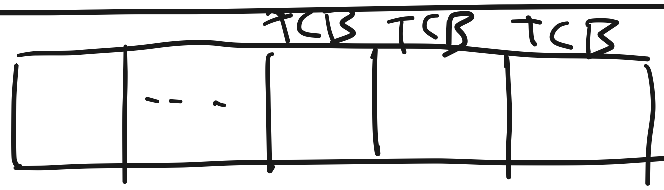
# 6. Implementation of processes

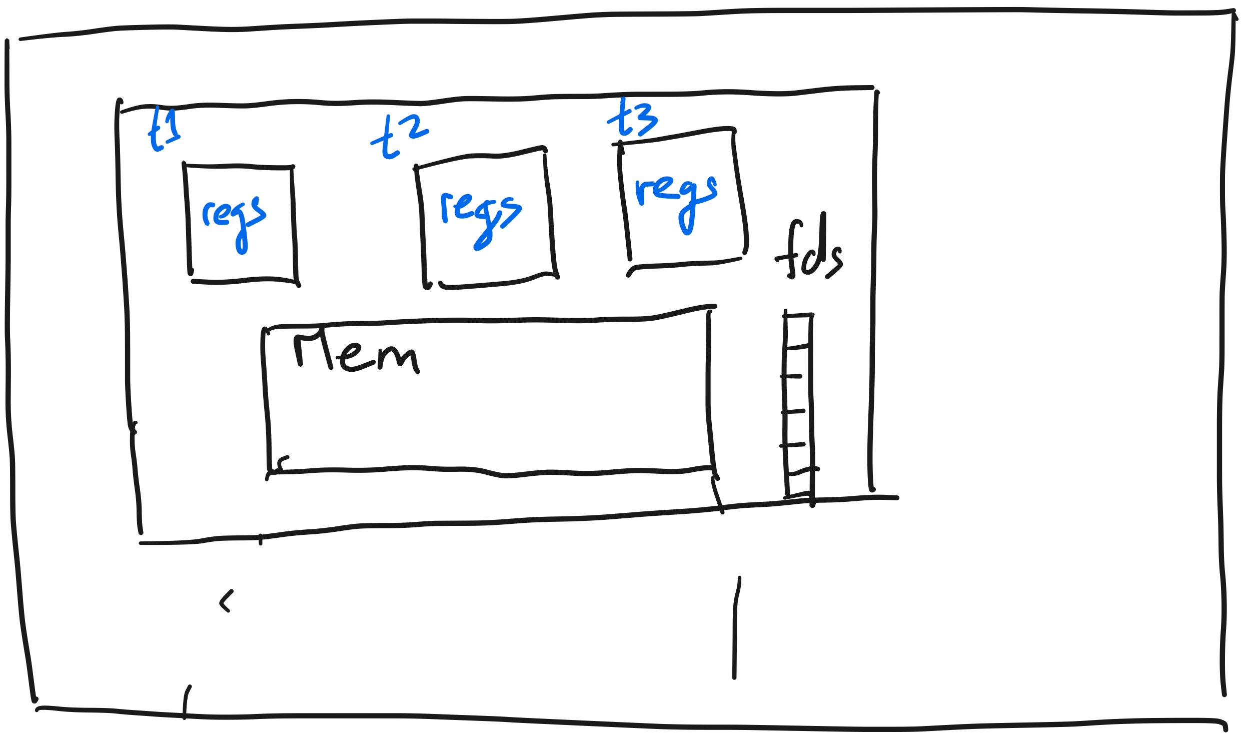


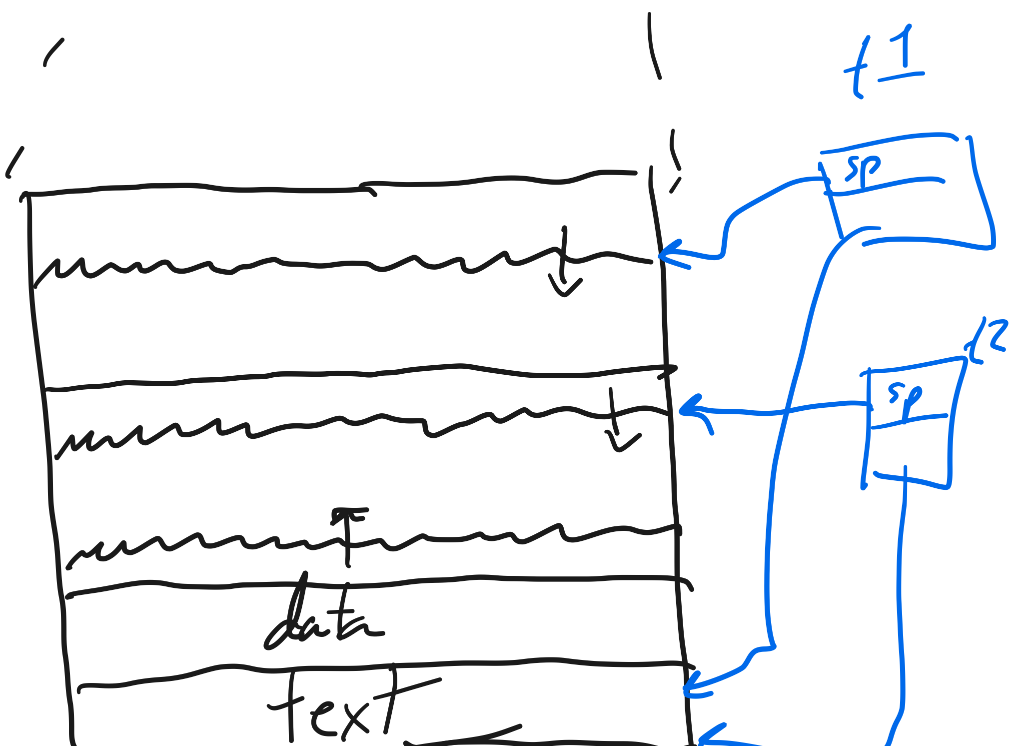




regs

# 7. Threads


TCB
TCB
TCB


t1
regs
t2
regs
t3
regs
fds
Mem


t1
sp
t2
sp
data
text

## interface

, int, int

```
tid  thread_create(void (*fn)(void*), void*)
void thread_exit();
void thread_join(tid thr);
```

+

lots of synchronization primitives

```
CS 202, Fall 2021
Handout 2 (Class 3)

The handout is meant to:

    --illustrate how the shell itself uses syscalls

    --communicate the power of the fork()/exec() separation

    --give an example of how small, modular pieces (file descriptors,
    pipes, fork(), exec()) can be combined to achieve complex behavior
    far beyond what any single application designer could or would have
    specified at design time. (We will not cover pipes in lecture today.)

1. Pseudocode for a very simple shell
```

"ls"   $_ ls&

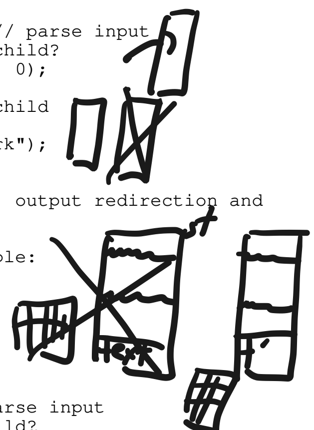
```
        while (1) {
                write(1, "$ ", 2);
                readcommand(command, args); // parse input
                if ((pid = fork()) == 0) // child?
                        execve(command, args, 0);
                else if (pid > 0) // parent?
                        wait(0); //wait for child
                else
                        perror("failed to fork");
        }

2. Now add two features to this simple shell: output redirection and
   backgrounding

    By output redirection, we mean, for example:
        $ ls > list.txt
    By backgrounding, we mean, for example:
        $ myprog &
        $

        while (1) {
            write(1, "$ ", 2);
            readcommand(command, args); // parse input
            if ((pid = fork()) == 0) { // child?
                if (output_redirected) {
                    close(1);
                    open(redirect_file, O_CREAT | O_TRUNC | O_WRONLY, 0666);
                }
                // when command runs, fd 1 will refer to the redirected file
                execve(command, args, 0);
            } else if (pid > 0) { // parent?
                if (foreground_process) {
                    wait(0);  //wait for child
                }
            } else {
                    perror("failed to fork");
            }
        }
```

"/tmp/foo"






```
3. Another syscall example: pipe()

    The pipe() syscall is used by the shell to implement pipelines, such as
        $ ls | sort | head -4
     We will see this in a moment; for now, here is an example use of
     pipes.

        // C fragment with simple use of pipes

        int fdarray[2];
        char buf[512];
        int n;

        pipe(fdarray);
        write(fdarray[1], "hello", 5);
        n = read(fdarray[0], buf, sizeof(buf));
        // buf[] now contains 'h', 'e', 'l', 'l', 'o'

4. File descriptors are inherited across fork

        // C fragment showing how two processes can communicate over a pipe

        int fdarray[2];
        char buf[512];
        int n, pid;

        pipe(fdarray);
        pid = fork();
        if(pid > 0){
          write(fdarray[1], "hello", 5);
        } else {
          n = read(fdarray[0], buf, sizeof(buf));
        }
```

5. Putting it all together: implementing shell pipelines using fork(), exec(), and pipe().

```
 // Pseudocode for a Unix shell that can run processes in the
 // background, redirect the output of commands, and implement
 // two element pipelines, such as "ls | sort"

  void main_loop() {

     while (1) {
         write(1, "$ ", 2);
         readcommand(command, args); // parse input
         if ((pid = fork()) == 0) { // child?
             if (pipeline_requested) {
                 handle_pipeline(left_command, right_command)
             } else {
                 if (output_redirected) {
                     close(1);
                     open(redirect_file, O_CREAT | O_TRUNC | O_WRONLY, 0666);
                 }
                 exec(command, args, 0);
             }
         } else if (pid > 0) { // parent?
             if (foreground_process) {
                 wait(0);  // wait for child
             }
         } else {
                 perror("failed to fork");
         }
     }
 }

 void handle_pipeline(left_command, right_command) {

     int fdarray[2];

     if (pipe(fdarray) < 0) panic ("error");
     if ((pid = fork ()) == 0) {  // child (left end of pipe)

        dup2 (fdarray[1], 1);  // make fd 1 the same as fdarray[1],
                               // which is  the write end of the
                               // pipe. implies close (1).
        close (fdarray[0]);
        close (fdarray[1]);
        parse(command1, args1, left_command);
        exec (command1, args1, 0);

     } else if (pid > 0) {        // parent (right end of pipe)

        dup2 (fdarray[0], 0);  // make fd 0 the same as fdarray[0],
                               // which is the read end of the pipe.
                               // implies close (0).
        close (fdarray[0]);
        close (fdarray[1]);
        parse(command2, args2, right_command);
        exec (command2, args2, 0);

     } else {
        printf ("Unable to fork\n");
     }
 }
```



6. Commentary

Why is this interesting? Because pipelines and output redirection are accomplished by manipulating the child's environment, not by asking a program author to implement a complex set of behaviors. That is, the *identical code* for "ls" can result in printing to the screen ("ls -l"), writing to a file ("ls -l > output.txt"), or getting ls's output formatted by a sorting program ("ls -l | sort").

This concept is powerful indeed. Consider what would be needed if it weren't for redirection: the author of ls would have had to anticipate every possible output mode and would have had to build in an interface by which the user could specify exactly how the output is treated.

What makes it work is that the author of ls expressed their code in terms of a file descriptor:

```
    write(1, "some output", byte_count);
```

This author does not, and cannot, know what the file descriptor will represent at runtime. Meanwhile, the shell has the opportunity, *in between fork() and exec()*, to arrange to have that file descriptor represent a pipe, a file to write to, the console, etc.

```c
/*
 * our_head.c -- a C program that prints the first L lines of its input,
 *   where L defaults to 10 but can be specified by the caller of the
 *   program.
 *
 *   (This program is inefficient and does not check its error
 *   conditions. It is meant to illustrate filters aka pipelines.)
 */
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char** argv)
{
    int i = 0;
    int nlines;
    char ch;
    int ret;

    if (argc == 2) {
        nlines = atoi(argv[1]);
    } else if (argc == 1) {
        nlines = 10;
    } else {
        fprintf(stderr, "usage: our_head [nlines]\n");
        exit(1);
    }

    for (i = 0; i < nlines; i++) {

        do {

            /* read in the first character from fd 0 */
            ret = read(0, &ch, 1);

            /* if there are no more characters to read, then exit */
            if (ret == 0) exit(0);

            write(1, &ch, 1);

        } while (ch != '\n');

    }

    exit(0);
}
```



```c
/*
 * our_yes.c -- a C program that prints its argument to the screen on a
 * new line every second.
 *
 */
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char** argv)
{
    char* repeated;
    int len;

    /* check to make sure the user gave us one argument */
    if (argc != 2) {
        fprintf(stderr, "usage: our_yes string_to_repeat\n");
        exit(1);
    }

    repeated = argv[1];

    len = strlen(repeated);

    /* loop forever */
    while (1) {

        write(1, repeated, len);

        write(1, "\n", 1);

        sleep(1);
    }

}
```

```
CS 202, Fall 2021
Handout 3 (Class 4)

1. Example to illustrate interleavings: say that thread A executes f()
and thread B executes g(). (Here, we are using the term "thread"
abstractly. This example applies to any of the approaches that fall
under the word "thread".)

    a. [this is pseudocode]

        int x;

        int main(int argc, char** argv) {

            tid tid1 = thread_create(f, NULL);
            tid tid2 = thread_create(g, NULL);

            thread_join(tid1);
            thread_join(tid2);

            printf("%d\n", x);
        }

        void f()
        {
            x = 1;
            thread_exit();
        }

        void g()
        {
            x = 2;
            thread_exit();
        }


        What are possible values of x after A has executed f() and B has
        executed g()? In other words, what are possible outputs of the
        program above?



    b.  Same question as above, but f() and g() are now defined as
    follows:

        int y = 12;

        f() { x = y + 1; }
        g() { y = y * 2; }

        What are the possible values of x?



    c. Same question as above, but f() and g() are now defined as
    follows:

        int x = 0;
        f() { x = x + 1; }
        g() { x = x + 2; }

        What are the possible values of x?
```



```
2. Linked list example

    struct List_elem {
        int data;
        struct List_elem* next;
    };

    List_elem* head = 0;

    insert(int data) {
        List_elem* l = new List_elem;
        l->data = data;
        l->next = head;
        head = l;
    }

    What happens if two threads execute insert() at once and we get the
    following interleaving?

    thread 1: l->next = head
    thread 2: l->next = head
    thread 2: head = l;
    thread 1: head = l;
```

```
3. Producer/consumer example:

    /*
    "buffer" stores BUFFER_SIZE items
    "count" is number of used slots. a variable that lives in memory
    "out" is next empty buffer slot to fill (if any)
    "in" is oldest filled slot to consume (if any)
    */

     void producer (void *ignored) {
         for (;;) {
             /* next line produces an item and puts it in nextProduced */
             nextProduced = means_of_production();
             while (count == BUFFER_SIZE)
                 ; // do nothing
             buffer [in] = nextProduced;
             in = (in + 1) % BUFFER_SIZE;
             count++;
         }
     }

     void consumer (void *ignored) {
         for (;;) {
             while (count == 0)
                ; // do nothing
             nextConsumed = buffer[out];
             out = (out + 1) % BUFFER_SIZE;
             count--;
             /* next line abstractly consumes the item */
             consume_item(nextConsumed);
         }
     }

    /*
       what count++ probably compiles to:
        reg1 <-- count      # load
        reg1 <-- reg1 + 1   # increment register
        count <-- reg1      # store

       what count-- could compile to:
        reg2 <-- count      # load
        reg2 <-- reg2 - 1   # decrement register
        count <-- reg2      # store
    */

    What happens if we get the following interleaving?

        reg1 <-- count
        reg1 <-- reg1 + 1
        reg2 <-- count
        reg2 <-- reg2 - 1
        count <-- reg1
        count <-- reg2
```



```
4. Some other examples. What is the point of these?

    [From S.V. Adve and K. Gharachorloo, IEEE Computer, December 1996,
    66-76. http://rsim.cs.uiuc.edu/~sadve/Publications/computer96.pdf]

    a. Can both "critical sections" run?

        int flag1 = 0, flag2 = 0;

        int main () {
            tid id = thread_create (p1, NULL);
            p2 (); thread_join (id);
        }

        void p1 (void *ignored) {
            flag1 = 1;
            if (!flag2) {
                critical_section_1 ();
            }
        }

        void p2 (void *ignored) {
            flag2 = 1;
            if (!flag1) {
                critical_section_2 ();
            }
        }

    b. Can use() be called with value 0, if p2 and p1 run concurrently?

        int data = 0, ready = 0;

        void p1 () {
            data = 2000;
            ready = 1;
        }
        int p2 () {
            while (!ready) {}
            use(data);
        }

    c. Can use() be called with value 0?

        int a = 0, b = 0;

        void p1 (void *ignored) { a = 1; }

        void p2 (void *ignored) {
          if (a == 1)
            b = 1;
        }

        void p3 (void *ignored) {
          if (b == 1)
            use (a);
        }
```