CS205

C / C++

Stéphane Faroult
faroult@sustc.edu.cn

Wang Wei (Vivian) vivian2017@aliyun.com

Back to System Calls and processes

We have seen some examples with signals; let's see why catching them can be useful and then let's move further into the exploration of processes.

Why catch e.g. SIGINT?

There are some excellent reasons while you would want to catch something such as Ctrl-C or any other intimation to stop, and not necessarily to try to resist such an order.

"Orderly termination"
"Clean shutdown"

- flush buffers and close files
- possibly delete temporary work files
- release resources

If a program suddenly quits, it may leave behind corrupted files, because by default everything isn't written instantly to files. Catching the signal allows you to close files before leaving. You may also want to remove some temporary work files, or release system resources, or close a network connection. Most "server type" programs try to put things in order before stopping.

How can we get the process id of an unrelated process ?

Now, we have said that we need the pid to deliver a signal, and that the only two pids we could get easily were our own (no need for a signal here) and our parent process pid.
What about other programs?

The system knows everything but you can't have direct access to sensitive data. Some systems may provide more facilities than others, but you can't rely on that.

Some programs call `getpid()`, then write their pid to a special file, e.g.

`/run/var/prog.pid`

Other programs can read this file, and signal them. This isn't a general practice.

The best all-round option probably remains `ps`

but how to automate other than shell scripting?

run_ps.c

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

int main() {
    system("ps -o pid,ppid,time,comm");
    return 0;
}
```

There is better with `popen()` (from `stdio.h`)

popen_ps.c

```c
#define LINELEN   500

int main() {
    FILE *p;
    char *s;
    char  line[LINELEN];
    int   len;

    if ((p = popen("ps -o pid,ppid,time,comm", "r")) != NULL) {
       while (fgets(line, LINELEN, p)) {
          len = strlen(line);
          while (len && isspace(line[len-1])) {
             len--;
          }
          line[len] = '\0';
          printf("read: %s\n", line);
       }
       pclose(p);
    }
    return 0;
}
```

popen() (pipe open) allows you to open a command as if it were a file, and to directly read from its standard output.

```
$ ./popen_ps
read:    PID  PPID      TIME COMM
read:    539   538   0:00.65 -bash
read:   1084  1083   0:00.14 -bash
read:   1177  1084   0:00.00 ./popen_ps
$
```

ps data becomes available inside the program.

To check a single program

```
ps -e -o pid,comm | grep prog
```

only shows the program name

You can refine (in that case the command to pass to `popen()` must be prepared first with say `sprintf()`) and pass the output through a filter (`grep`) that looks for 'prog' on the line before returning it to your program.

To check a single program

```
ps -e -o pid,command | grep prog | grep -v grep
```

also shows command line arguments

Another variant of `ps` allows you to get the full line with the arguments that were passed (may be useful) but in that case "`grep prog`" will also contain "prog" and will also be returned and another filter must be added to omit lines that contain "grep".

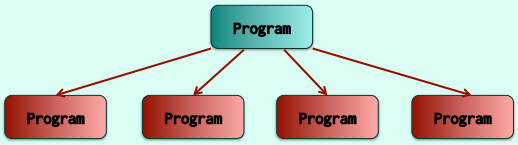
Starting subprocesses

`system()`	serializes
`popen()`	strong link

`system()` executes a command, but waits for its completion before returning and serializes operations. `popen()` is a way to have different programs running concurrently, but as one reads from the standard output (or writes to the standard input) of the other, they are strongly entertwined.

There is often a need for starting and controlling programs that work independently of each other (load simulator, for instance) or collaboratively (e.g. processing massive amounts of data)

Starting subprocesses

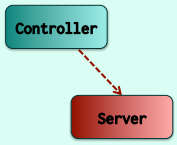


Simulations

Parallel processing

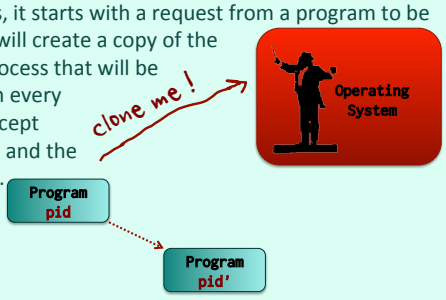
Another common case is when you want to start a server (HTTP server, FTP server, database server ...) that will have a life of its own after you have logged out. Such a program is called a daemon.

Starting subprocesses



Daemon (session independent)

In all cases, it starts with a request from a program to be cloned. It will create a copy of the original process that will be identical in every respect except for the pid and the parent pid.



Hits the ground running!

The duplicate process runs immediately.

The request to be cloned is performed with the `fork()` call.

clone me = **fork()**

```
#include <unistd.h>
```

fork() takes no arguments

returns a pid_t

-1 if error

0 if child

child pid if parent

As the duplicate program runs immediately, when you return from `fork()` you can be in either program. The return value tells you which one.

DEMO

You'll find the code of demos in file `fork_examples.zip`.

Inherited:

As already said, the forked process will be identical in every respect, except for pid and parent pid. It will run the same code, write to the same files and the same terminal.

- files
  - including stdin/cin, stdout/cout, stderr/cerr

"attached" to a terminal window

in the same way as the parent process.

What happens when one of the processes terminates?

Child dies:

- parent receives `SIGCHLD`

Parent dies:

- process #1 becomes foster parent

A parent process is expected to

wait for the completion of the child process.

The reason is that a process returns a status (the `int` return value) and that this status is supposed to be at least acknowledged by the parent process. As long as the status isn't acknowledged, the system cannot quite cleanup everything related to the completed process (except, apparently, the last versions of Mac OSX)

A parent process is expected to call a `wait()` function.

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

pid_t wait(int *stat_loc);

pid_t waitpid(pid_t  pid,
              int   *stat_loc,
              int    options);
```

Wait for any child (its pid is returned)

Wait for a specific child.

Several other functions

Process #1, the foster parent of all orphaned processes, does nothing but waiting.

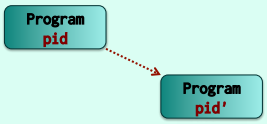
A not-waited for process

becomes a **ZOMBIE**

If the parent process is still running, but doesn't wait for a terminated child process, the child process becomes a zombie, unable to find eternal rest before the parent waits for it. When the parent dies, then all is fine because process #1 will wait.
Although zombie processes (which no longer run) take no resources, system engineers don't like them; they often indicate sloppy programming.

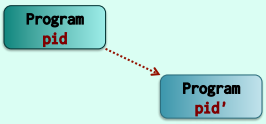
What about starting a different program?

If you write a generic program launcher, you may want to run any program, not necessarily a second copy of the code of your launcher itself.
This is performed in two steps.



First the program clones itself. This provides a "shell".

exec()



Then the child program calls the "exec()" function that loads and starts running the code for a completely different program.

Most useful ones IMHO

Same as path

others are "command-line" arguments

```
int execl(const char *path, const char *arg0, ... , NULL);

int execv(const char *path, char *const argv[]);
```

First element = path, terminated by NULL

"exec()" is a generic name for a whole family of functions with differents parameters (and names, because this is C and functions cannot be overloaded, so different parameters mean a different name)

Command path

PATH

The exec() functions use the value of PATH that is currently defined in the environment to locate the executable file to load and run.

```
char *getenv(const char *name);
```

As an aside, getenv() is the function that allows you to retrieve the value of any environment variable.

Launching an independent program

(daemon)

When you run a simulation, for instance, a parent process forks a large number of subprocesses who will `exec()` a few different programs, then waits for them to complete before exiting. Different story when you start a daemon or system service, as you start it, don't wait, and can even log out.

Problems to solve:

- not attached to a terminal
- parent won't wait for it
- environment (directories, etc.)

The last point deserves some comments: when you open a terminal, you have an environment. You have a home directory. You may be in a given directory. All of this has very little meaning for a system service such as a web server; it has a "root directory", but it's unrelated to a location when it was started (it's read from a configuration file)

Creating a daemon

Many systems implement a `daemon()` function (called by the child process) that turns a process into a system service. The Mac compiler doesn't like it but it works nevertheless.

```
#include <stdlib.h>

int daemon(int nochdir, int noclose);
```

if 0 moves current directory to /

if 0 redirects stdin, stdout and stderr to /dev/null (aka "black hole")

Creating a daemon

This is a brief summary of what to do if you have no `daemon()` function. Not THAT easy to do well.

If not available:

- Need to fork TWICE
- Create a new session (`setid()`)
- Deal with opened file descriptors

Search the web!

Message: subtler than it looks

IPC
Inter-Process Communication

Now that we know how to create a herd of little galloping subprocesses, let's see how we can shepherd them, and primarily how we can communicate with them or let them communicate between themselves. We have already talked about using files (synchronization issues), sockets (heavy-handed on one computer, need to listen) and signals; signals are great for interrupting, but signals are like shouting "hey!" or waving furiously: the amount of information they can transmit is limited.

IPC

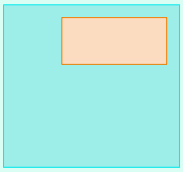
The AT&T System V version of Unix brought to the Unix world IPCs, which are three mechanisms. The first two ones can really be used for transmitting information, the third one is for synchronization.

- Messages
- Shared Memory
- Semaphores

IPC

Messages are basically a shared queue, where processes can add and remove messages.

- Messages



- Shared Memory

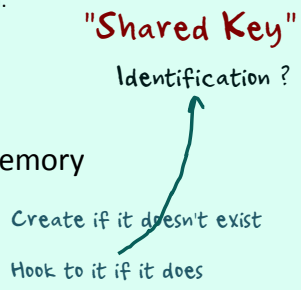
Shared memory allows mapping in the private address space of a process a memory area that simultaneously appears in the address spaces of several processes;

The use of signals can then be limited to notification, eg "I've just posted a message for you".

IPC

Both messages and shared memory follow the same pattern.

- Messages
- Shared Memory



SHARING a message queue or memory area requires that several processes use the same identifier for it.

IPC

The identifier will be an integer but here we have a problem. We cannot decide on our own that THIS number will identify our message queue; another program might select the same number. As it's coded in programs, it cannot be random either.

key_t → integer

"Shared Key"

cannot be random

uniqueness issue

The following function is usually called to generate a suitable shared key

```
key_t ftok(const char *path, int id);
```

The idea is to build the key out of the identification of a file that is only used by your program (or software suite). A second parameter (often a letter) allows creating several keys that are truly yours.

```
#include <sys/msg.h>

key = ftok("/etc/supersoft.conf", 'q');
msqid = msgget(key, 0666 | IPC_CREAT);
```

rw-rw-rw-

With the key (static) you obtain a message queue id (dynamically allocated by the system, like sockets or file identifiers) which will be used for subsequent operations. A flag specifies access rights and says to create the queue if it doesn't exist already.

```
struct msgbuf {
    long mtype;  // Must be positive
    char mtext[1];
};
```

dummy

```
int msgsnd(int msqid, const void *msgp,
           size_t msgsz, int msgflg);
```

msgsz excludes mtype

a `struct msgbuf` must contain a long as first member, but the second member can be anything: it can be an array of chars of any length, it can be a structure ... anything that will allow you to communicate what you want, the system only knows a number of bytes starting at a given address.

```
struct msgbuf {
    long mtype;  // Must be positive
    char mtext[1];
};
```

Contrary to the (very similar) `send()` and `recv()` functions for sockets, `msgrcv()` takes an additional parameter.

```
int msgrcv(int msqid, void *msgp,
           size_t msgsz,
           long msgtyp, int msgflg);
```

0 : get next message

>0 : get next message of this msgtyp

<0 : get next message with msgtyp < abs(arg)

Deleting a message queue

```
#include <sys/msg.h>


msgctl(msqid, IPC_RMID, NULL);
```

When you are done with a message queue, you must delete it (it can also be done in a console with command ipcrm, but it's FAR better to do it in a program). This is usually done by the parent process when all subprocesses have terminated. Kind of switching the lights off before leaving the room.

Shared Memory

```
key_t key;
int   shmid;
char *data;
```

Very similar mechanisms with shared memory. Key required, you request a shared memory segment of a given size and get a dynamic identifier, then `shmat()` returns a pointer to it, `malloc()` style.

```
key = ftok("/etc/supersoft.conf", 'm');
shmid = shmget(key, 1024, 0644 | IPC_CREAT);
data = shmat(shmid, (void *)0, 0);
```

```
(void)shmdt(data);
```

When you are done you unmap (detach) shared memory from your space.

You can pass a not NULL pointer if you want the shared memory to be mapped to a specific address. I have never found any use for this feature so far.

Shared Memory

When all processes have detached with `shmdt()` from the shared memory area, the segment is still there (`shmdt()` undoes what `shmat()` does, you need to undo the original `shmget()`)

```
shmctl(shmid, IPC_RMID, NULL);
```

Same type of call as with message queues.

Shared Memory

Your program doesn't have to worry too much with message queues, concurrency issues are handled by the system. However, with shared memory, you cannot write at an address in a shared segment without ensuring that another process is not, at the same time, overwriting YOUR bytes with what is, from your point of view, garbage.

Concurrency?

Here is the solution :

Semaphores

Semaphores

You are usually asking for *nsems* distinct semaphores, each one protecting a distinct variable or structure in shared memory.

```
#include <sys/sem.h>

int semget(key_t key, int nsems, int semflg);
```

Must be initialized with a call to `semctl()`

You are requesting, in the same fashion as with other IPCs, a set of semaphores which can be seen as counters. You must initialize them, let's say that you set all of them to one to mean that all resources which are protected are initially accessible.

When a program needs access to a resource, it will specify the id of the semaphore set, then the number of the specific semaphore associated with the resource, and say it wants to decrement the counter by 1. If the counter is one, this will be possible, the operation is guaranteed to be one-process-at-a-time, the process will grab the resource and the counter will become 0. Another process doing the same request a fraction of a second later will block, as the counter is 0. When the first process is done, it releases the resource by adding one to the semaphore. Then the second process is unblocked and can grab it, and so forth. Sometimes a bit difficult to tune, especially deciding what are the right initial values.

Semaphores

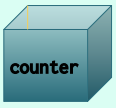
```
struct sembuf {
    ushort sem_num;
    short  sem_op;
    short  sem_flg;
};
```

>0 add to semaphore counter (release)

<0 block until it can be subtracted from semaphore counter (grab)

0 block till counter is zero

```
int semop(int            semid,
          struct sembuf *sops,
          unsigned int   nsops);
```



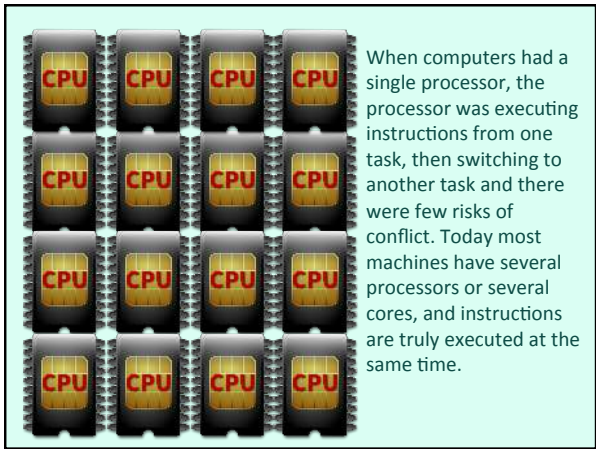
semop() is the blocking call.

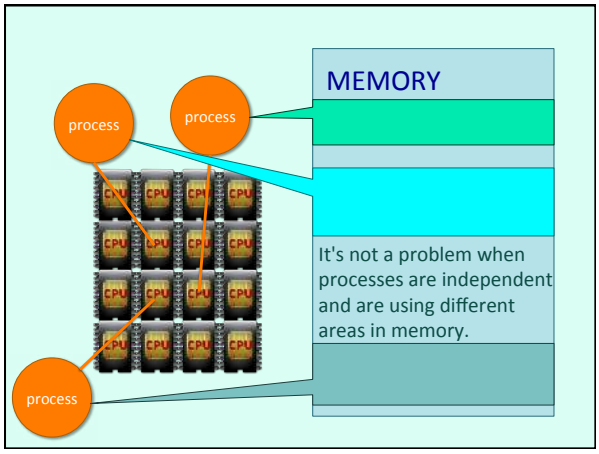
Semaphores

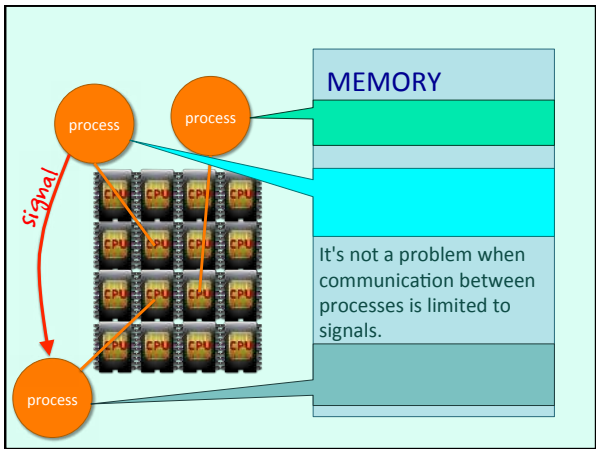
As always, semaphore sets must be destroyed when no longer used. Note that here `IPC_RMID` is the third parameter, the second one is a semaphore number only used for other operations.

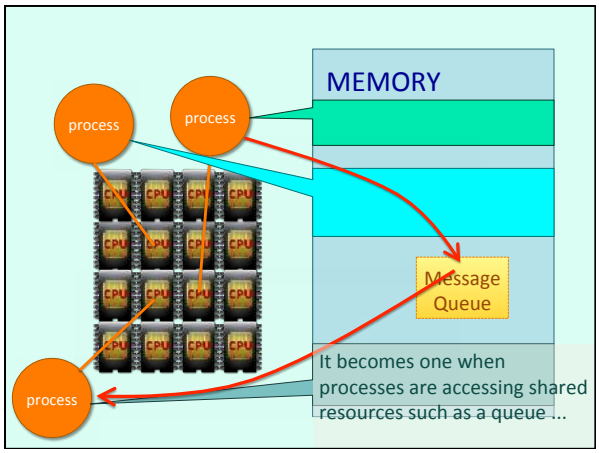
```
semctl(semid, 0, IPC_RMID);
```

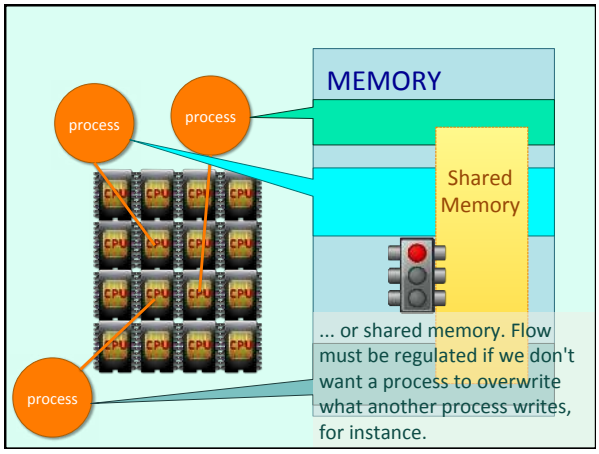
semnum (ignored)


When computers had a single processor, the processor was executing instructions from one task, then switching to another task and there were few risks of conflict. Today most machines have several processors or several cores, and instructions are truly executed at the same time.

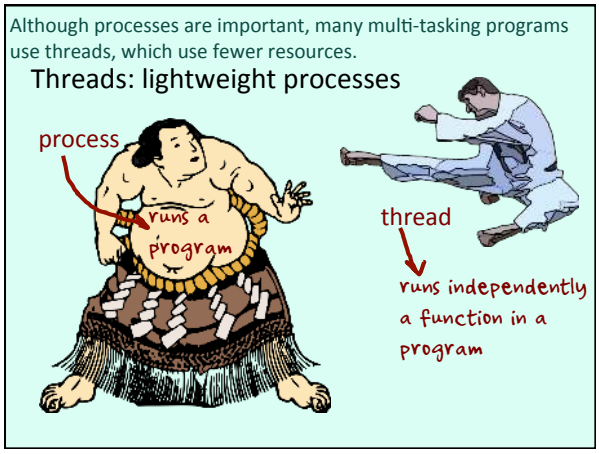

MEMORY
process
process
process
It's not a problem when processes are independent and are using different areas in memory.


MEMORY
process
process
signal
process
It's not a problem when communication between processes is limited to signals.


MEMORY
process
process
Message Queue
process
It becomes one when processes are accessing shared resources such as a queue ...



Although processes are important, many multi-tasking programs use threads, which use fewer resources.

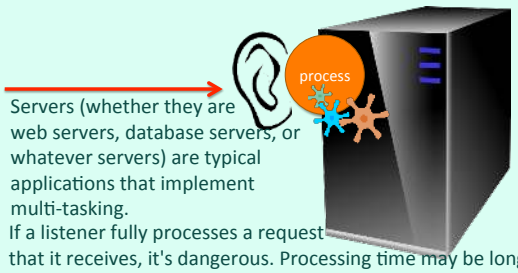


Threads: lightweight processes

In theory. They are sometimes implemented as processes.

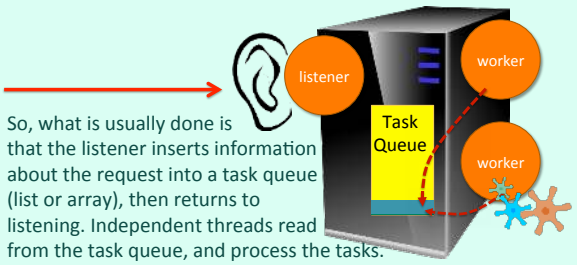
It's the case in Linux systems. Note also that a system such as Windows only uses threads, and not full-blown processes.
I'll use threads to discuss synchronization issues.

Typical Application



Servers (whether they are web servers, database servers, or whatever servers) are typical applications that implement multi-tasking.
If a listener fully processes a request that it receives, it's dangerous. Processing time may be long (some database queries can run for several minutes, even hours). Meanwhile, it would no longer be listening for other requests.

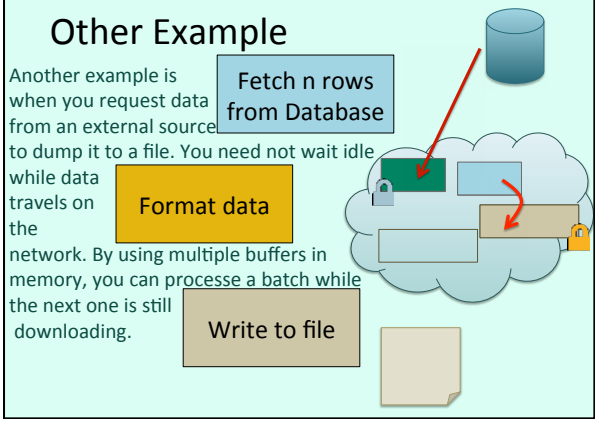
Typical Application



So, what is usually done is that the listener inserts information about the request into a task queue (list or array), then returns to listening. Independent threads read from the task queue, and process the tasks. With several such "worker threads" you make sure that a long task will not block everything (it's like multiple cashiers in a supermarket).

Other Example

Another example is when you request data from an external source to dump it to a file. You need not wait idle while data travels on the network. By using multiple buffers in memory, you can processe a batch while the next one is still downloading.



Synchronization requires rigor ...

The big issue is synchronization. If a thread is reading from memory while another one is writing to it, what is read may be corrupted. With a task queue, if a worker task notices that there is a new task and another worker tasks sees the task at the same time, there is a significant risk of seeing some tasks processed twice (it's known as "race condition"), which may have serious consequences (imagine that the task is buying or selling a huge number of equities ...). There is no built-in mechanism in C for synchronization, and you must be very rigorous.

Threads Basics

```
#include <pthread.h>

gcc -D_REENTRANT    ......-lpthread
```

To use threads in a C program, you must include pthread.h and link with libpthread. You must also declare the symbol _REENTRANT. Reentrant means that a function may be called recursively or be interrupted in the middle but will still give a correct result if called again (re-entered) later. An enthusiastic usage of static variables usually puts reentrancy at risk.

Threads Basics

`pthread_create()` is very similar to the `fork()`/`exec()` combination, except that execution in the created thread starts at the beginning of the `start_routine` function, to which `arg` is passed.

```
int pthread_create(pthread_t * thread,
    pthread_attr_t * attr,
    void * (* start_routine) (void *),
    void * arg);
```

detached/joinable

scheduling

stack

NULL = default

The optional attribute allows to say how independent a thread is, and to specify some deeply technical details.

Threads Basics

```
void pthread_exit(void *retval);
```

As a thread is part of a program it musn't call `exit()` but `pthread_exit()` when it terminates.

```
int pthread_join(pthread_t th,
                 void **thread_return);
```

Similar to wait()

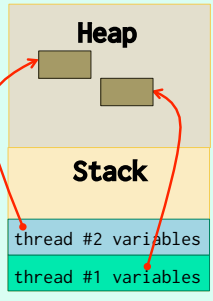
Unless the thread was detached (special attribute) the parent thread must call `pthread_join()` to get its return value (status).

Threads Basics

No problem with local variables

No problem with heap variables

Problem with static and global variables



Several threads may call the same function, there is no problem with local variables that pile up in the stack, nor with what is malloc'ed in the function. The problem is with shared variables.