The basic model of time control is the idea of virtualizing the processor. This is the idea of using the hardware to build more virtual versions of itself, and has the advantage of time sharing. We can use it to convince a program it is running with control of the entire computer, whilst also keeping certain procedures protected.

We do these by modeling a program running on a virtual processor as a *process*. These have their own ALU, registers, and (virtual) memory.

We prevent hackers from exploiting these by limiting access to system calls. Whilst it is slow, the time cost is not on the same order of I/O and is thus irrelevant.

How do we run a program?

1. load its data and code to memory/address space in executable format.
   This can be done *eagerly* or *lazily*
   - eagerly := load all data before the program runs
   - lazily := load data only as needed
2. allocate stack and heap
3. initialize stdin, stdout, stderr file descriptors
4. jump to main and transfer the control to the process to be run

C provides an api for this:

```
pid_t fork(void);
// creates a child process and returns a PID (0 in child, child PID in parent)
// the child runs the same code as the parent starting from after the fork
int execvp(const char *file, char *const argv[]);
// run a process
// argv is an array of strings that represent the argv for the call
// all data is copied except for the new data from the earlier section
```

The protocol for running a child program is:

- fork() — new virtual machine, same program
- exec() — same virtual machine, new program

This can lead to errors, so the parent is responsible for "baby proofing" the program. A bootstrap booter, on the other hand, calls exec() but not fork(), since we do not want to be returned to.

This can introduce a few error:

```
// we can have only the child exit with:
if (!fork()) ... exit(0);
// but what if the parent never calls wait? or worse:
if (fork()) ... exit(0);
// now the child is left as an orphan!
while(true) fork();
// a fork bomb takes up ALL available memory with a "dud" processor!
```

The data copied by fork and replaced by exec can be large and includes all program data except:

- PID and parent PID
- accumulated execution time
- file locks

Another abstraction we use is called a *thread*. Within a process, we may have multiple threads. A thread has its own program counter and registers. A thread switch does not require a change in address space; a thread only has its own stack/heap, referred to as thread-local storage, which includes unique copies of things like errno. Processes have an ID, SP, PC, and page-map-address (PMAR); a thread has virtual versions of all of these.

But why even use threads?

Process	Thread
fork()	pthread_create()
exec()	pthread_join()
waitpid()	pthread_join()

1. exploiting parallelism
2. overlap to avoid blocking inefficiencies (within the same program)
3. latency control
4. exploit the multiprocessor

Processes, on the other hand, would make it hard to share data!

How do we use threads?

1. load program text and data
2. allocate thread and run

Both of these are done by the *thread manager*.

We identify threads similarly to how we identify processes: Both types give a *handle*:

- Processes give an identifier (pid_t) in the form of an integer. This is *opaque*, and therefore restricts direct access to the processor that granted the ID.
- Threads give an identifier (pthread_t) in the form of a pointer. This is *transparent*, and therefore allows direct access and sharing of data structures.

Contrary to what its name would have you believe, kill() is not the actual process killer, since a parent still holds on to the process. exit() does not kill either, as an exiting child can be ignored and left as a zombie. Therefore it must be waitpid()!

We actually have a few additional ways to interrupt processes and threads:

```
pid_t waitpid(pid_t pid, int *status, int options);
// recombine the threads upon completion
// note a call to waitpid with options=-1 is equivelant to wait()
// Upon which the parent thread catches all the completed threads
void exit(int status);
// pass the return value
// uses jmp, so it never actually returns
void _noreturn _exit(int status);
// does not clear bufferx etc -> faster
// raise a signal on another thread
int kill(pid_t pid, int sig);
// if sig < 0, send to all decendents as well
// if pid = getpid(), we can use:
int raise(int sig);
// these only work if the sender and receiver share a user
unsigned int alarm(unsigned int seconds);
// a timer to guarantee the child does not run forever
// but the child could end the alarm with alarm(0)!
```

That being said, threads may still turn into orphans or zombies, so a process with index 1 takes in all orphaned threads by periodically calling

```
int waitpid(-1, status, UNOHAND);
// does not wait for children to complete, but does catch all completed children
```

We can actually simplify this process and reduce overhead by combining the fork() and exec() system calls, but it often ends up being more work than it is worth. That being said, C provides a procedure call for this:

```
int posix_spawn(pid_t *pid, const char *path,
                const posix_spawn_file_actions_t *file_actions,
                const posix_spawnattr_t *attrp,
```

```
                    char *const argv[], char *const envp[]);
// pid: pointer to process
// file: executable
// actions: fds for the child
// attrp: other attributes for the child
// envp: environment pointer
// "restrict" means that there exist no other pointers pointing to these objects
```

0.1 Inter-Process Communication

UNIX has a command called 'date' of the form:

Wed Jan 14 13:34:03 PST 2015

We attempt to emulate it in C with a status of the form:

```
bool printdate(void) {
  if ((pid_t p = fork()) < 0) return false;
  // have a child do the call!
  if (p == 0) {
    execvp("usr/bin/date", (char*[]){"date", "-u", 0});
    // this is only reachable on exec failure
    exit(127);
  }
  // if we get here, we are in the parent
  int status;
  if(waitpid(p, &status, 0) < 0) return false;
  // WIFEXITED tells us if the child exited (as opposed to signaled)
  // WEXITSTATUS gives us the exit value
  return (WIFEXITED(status) && WEXITSTATUS(status) == 0);
}
```

Unsurprisingly, there are other ways to transfer data between isolated processes. Processes (mostly) run in isolation for:

- ease of debugging
- security

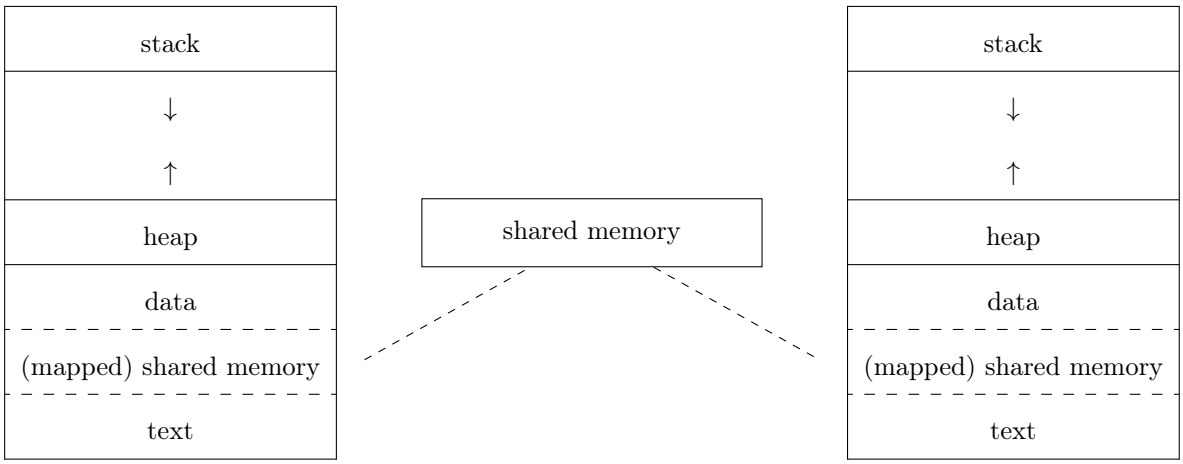
But there are a few notable exceptions:

- fork() conveys the child pid to the parent
- wait() conveys exit info to the parent
- kill() conveys info to everyone

Isolation is a good thing, but it makes it hard to communicate big data; how can we?

1. Storing in a file
   - First program must finish before the seconds can access it
   - I/O is SLOW
2. Message Passing
   - We send parts of a large structure in messages
   - This also involves slow copying
3. Shared Memory
   - give up isolation in pursuit of efficiency

If we require isolation, we must still use the first two methods.


stack
↓
↑
heap
data
(mapped) shared memory
text
shared memory
stack
↓
↑
heap
data
(mapped) shared memory
text