5. Parallelism

The basic model of time control is the idea of VIRTUALIZING THE PROCESSOR

- the hardware build more virtual versions of itself
- this also has the advantage of time sharing

We can use it to convince a program it is running with control of the entire computer

To do this, we model a program running on a virtual processor as a process

Processes

- have their own (virtual) memory
- have their own ALU & registers

Hackers could exploit these, so we limit I/O access to system calls

  - this is slow, but in terms of order it does not add much to the time cost of I/O

How do we run a program?

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

Isn't this approach time & memory inefficient?

~ it would be, if the OS didn't copy the data lazily

There is 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
```

Fork copies & exec replaces all data except:

- pid & parent pid
- accumulated execution time
- pending signals
- file locks

```
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 doesn't run forever
// but the child could end the alarm with alarm(0)!
```

Protocol
1. fork() ~ new virtual machine, same program
2. exec() ~ same virtual machine, new program

NOTES:
~ this can lead to errors, so the parent is responsible for “baby-proofing” the program
~ a bootstrap booter calls exec but not fork, since we do not want it to be returned to

This can introduce a few errors!

```
// 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!
```

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 also doesn’t require a change in address space;
  - a thread only has its own stack/heap, referred to as **thread-local storage**
    - this includes unique copies of things like errno
- Processes have an ID, SP, PC, & **page-map-address (PMAR)**;
  - a thread has virtual versions of all of these
- Why even use threads?
  i) exploiting **parallelism**
  ii) **overlap** to avoid **blocking** inefficiencies (within the same program)
  iii) latency control
  iv) exploit the multiprocessor

Processes would make it hard to share data!

How to use threads?

i) load program text and data
ii) allocate thread and run

This is all done by the **thread manager**

We identify these threads similarly to how we identify processes:

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

- Processes give a pid_t, like an integer
- Threads give a pthread_t, which functions like a pointer
- These are both called handles

pthread_t is transparent & therefore allows direct access & sharing of structures
pid_t is opaque and restricts direct access to the processor that granted the ID

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
—> it must be waitpid()!

To deal with orphans and zombies, a process 1 takes in all orphaned threads
This process occasionally calls;

```
int waitpid(-1, status, UNOHAND);
```

which doesn't wait for children to complete, but does catch all completed children

We can actually combine the fork() and exec() system calls to lower overhead
~ BUT it ends up being more work than it is worth

```
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
```

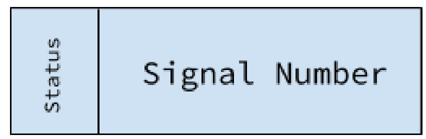
Let’s see if we can put all this code to good use:

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:



Form of status

```
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);
}
```

Since processes are distinct things and work in isolation from one another, this is the only way to transfer data, right?

~ OF COURSE NOT

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

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