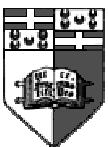




A Unix Process

- We have examined the memory layout of a UNIX process before.
- In this section we will see more in detail about how each process executes within the UNIX environment.
- Each process is identified by a unique integer known as a process identifier (PID).
- Each process identifier is associated with a process descriptor inside the operating system scheduler.
- A list of PIDs can be obtained by typing ‘ps -uax’ at the command prompt and information about each process can be found in /proc/pidn which is a directory corresponding to each PID.

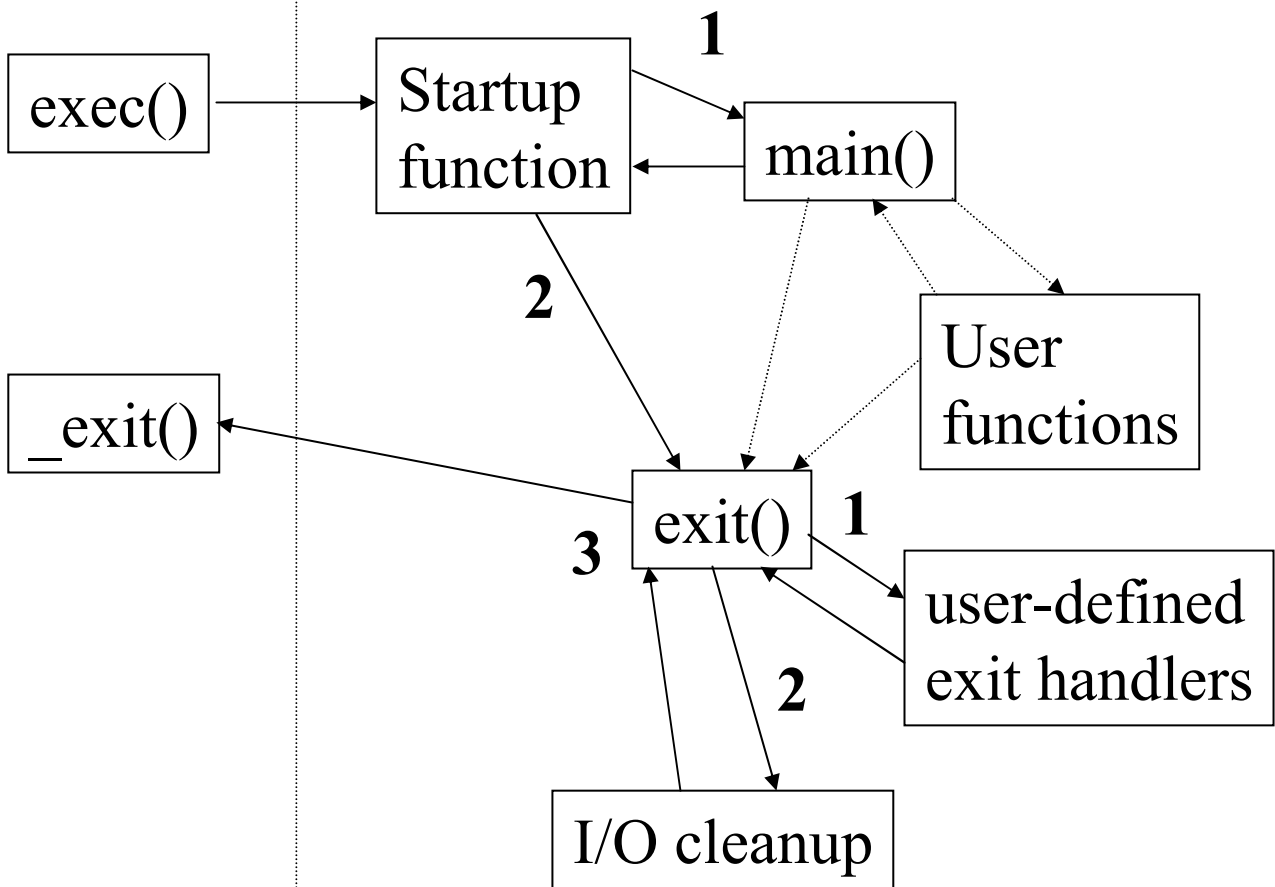




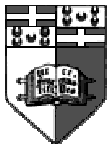
Process Start and Termination

KERNEL

USER SPACE



..... Optional
n Order of execution

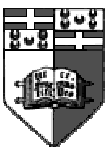




Process Startup and Termination (cont)

```
#include <stdlib.h>
#include <unistd.h>
int main(int argc, char *argv[]);
void exit(int status);
void _exit(int status);
```

- *main()* is passed the command line arguments and should *return* the termination status (0 if ok, -1 on error). If there is no *return* instruction, the return value is undefined example on fall-off.
- *exit()* takes the termination status and causes any files to be properly closed and releases any memory, etc
- *_exit()* terminates the process immediately closing all open descriptors but not cleanly.

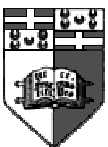




User defined exit functions

- The *exit()* function calls any registered user exit functions before calling the termination functions.
- Exit functions are registered using *atexit()* which takes a pointer to function that does not return anything and takes no arguments.
- Calling *_exit()* directly allows us to skip these registered *atexit* functions.

```
#include <stdlib.h>
int atexit(void (*func) (void));
        returns 0 if OK, -1 on error
```

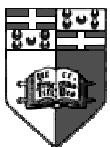
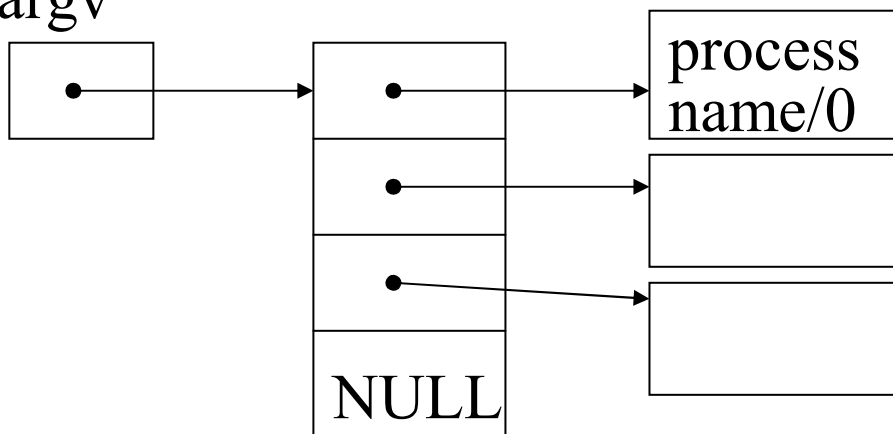




Command Line Arguments

- *main()* is passed two arguments. These are called the command line arguments which are passed from the caller of our program.
- *argc* is the number of arguments present in *argv[]*. It always has the value 1 or more.
- *argv[]* is an array of pointers with element 0 always pointing to a string storing the process name and the last value equal to **NULL**. *argv* has type equivalent to *char ***.

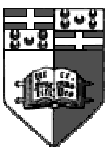
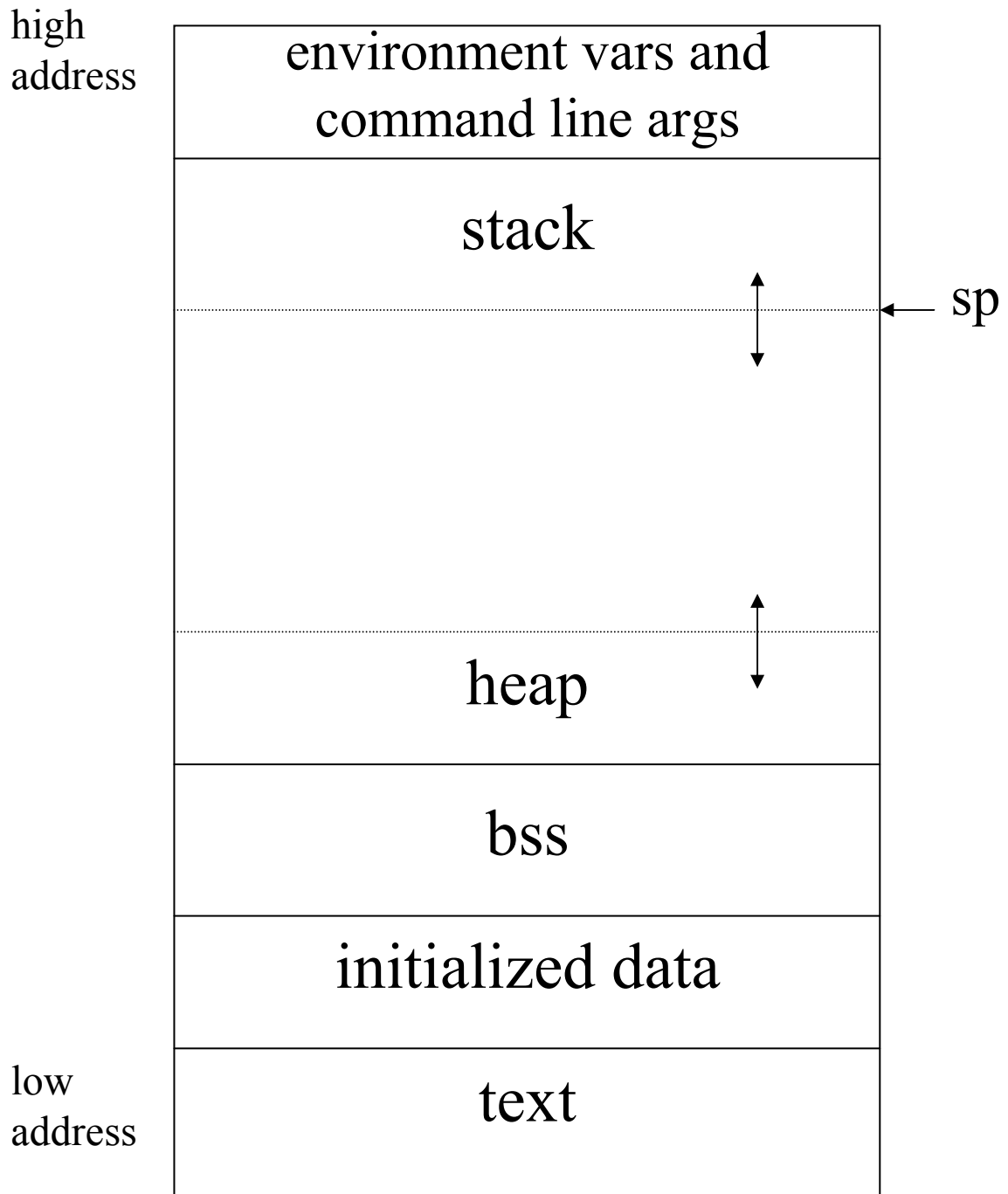
argv





User Space

Memory Layout of a Process



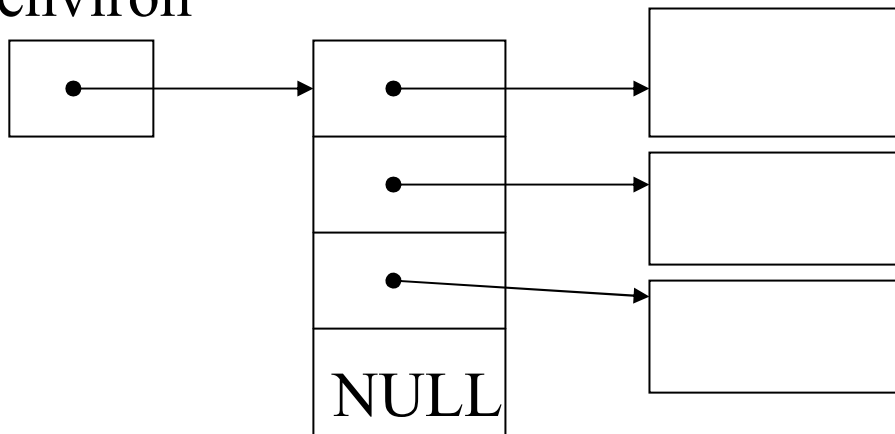


Environment Variables

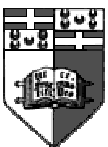
- Each process inherits from the system a list of environment variables stored inside a global variable

```
char **environ;
```

environ



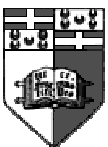
- Each string has the format “*NAME=value*”
ex. “HOME=/home/jcorl”
- When adding values or modifying new values, space might be *malloc*'ed on the heap and the appropriate values updated.



Unit Environment Variables (cont)

```
#include <stdlib.h>
char *getenv(const char *name);
        return pointer to value, NULL on error
int putenv(const char *str);
int setenv(const char*name, const char *value,
        int overwrite)
        return 0 of OK, -1 on error
void unsetenv(const char *name);
```

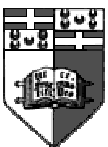
- *putenv* creates new entries or replaces a value if *NAME* exists.
- If *overwrite* is 0 and *name* exists then *setenv()* does nothing and returns with a 0 return value.
- *unsetenv()* fixes all the pointers to conform to the expected convention.





Resource Limits

- Each process has a limited number of resources which it can use.
- A hard limit defines the upper bound of the resource which only the **superuser** can change.
- The soft limit is the current resource limit assigned to the process.
- A user may lower its hard limit but this operation is usually irreversible.
- A user may raise or lower its soft limit as long as it stays lower than the hard limit.





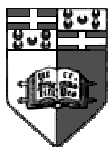
Resource Limits (cont)

```
#include <sys/time.h>
#include <sys/resource.h>
int getrlimit(int resource, struct rlimit *rlptr);
int setrlimit(int resource,
              const struct rlimit *rlptr);
              return 0 if OK, -1 on error
```

```
struct rlimit {
    rlim_t rlim_cur;
    rlim_t rlim_max;
}
```

Resource Limits

```
RLIMIT_CORE
RLIMIT_CPU
RLIMIT_DATA
RLIMIT_FSIZE
RLIMIT_MEMLOCK
RLIMIT_NOFILE
RLIMIT_NPROC
RLIMIT_RSS
RLIMIT_STACK
RLIMIT_VMEM
```

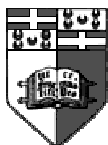


Unix

setjmp() and *longjmp()*

- When we want to exit from a deeply nested situation we can use *goto*, yet the label cannot reside in a different function than *goto*.
- *setjmp()* records the current position together with the current stack pointers and returns 0.
- *longjmp()* takes you to the *setjmp*'ed position, restores the stack pointers and returns *val* (inside *setjmp()*).
- One can build a thread scheduler like this !!!

```
#include <setjmp.h>
int setjmp(jmp_buf env);
    return 0 when called first,
    non-zero after longjmp
void longjmp(jmp_buf env, int val);
    val must be non zero
    returns in setjmp() !!!
```





Exercises

- Write your own ‘echo’ command.
- Write an exit function which displays a goodbye greeting when the process terminates.
- Display all the environment variables inside your process. If you change the shell prompt, do these change (C shell , Bourne shell, Korn shell, etc)?
- Experiment with `setjmp` and `longjmp`. After a `longjmp`, are the variables the same as when you executed the `setjmp`, even if these variables changed after the `setjmp` instruction?

