Today

• Fork
• Exec
• Wait
• GDB
• Valgrind
• Splint
Processes in Unix

What’s a process?
- A program in execution

Difference between a process and a program?

Different states of a process:
1) Created (New)
2) Ready/Waiting (In main memory, awaiting execution [via context switch])
3) Running
4) Blocked (Waiting on an event)
5) Terminated

Diagram:
- New
  - Admitted
    - Selected to run
      - Ready/Waiting
        - Quantum Expired
          - Running
            - Exit
              - IO Complete or Event Complete
              - IO Request or Event Wait
                - Blocked
                  - Terminated
Process pool

- How do you find out what processes your system is running currently?
  - `ps -af`
  - `top` *(gives an interactive list of processes along various others stats)*
  - Look at the man pages of `ps` and `top`
fork()

- fork() – Creates a new process
- Parent process executes fork and creates an almost identical copy of itself
- Child process inherits parent’s state and context:
  - Code, data, open files
  - Program counter and stack
- #include <unistd.h>
- If fork() fails, it returns -1 and sets a errno to EAGAIN
- If fork() succeeds, it returns 0 to the child and the child’s pid to the parent.
- Potential pitfalls:
  - duplicate memory can provide intermixed output (try stdout)
fork()

Parent process (pid = 1000)

child_pid=fork()

child_pid=2000
Parent Process (pid=1000)

printf("I am a parent");

child_pid=0
Child Process (pid=2000)

printf("I am a child");

End

child_pid=-1
Error!
forking.c

```c
pid_t childpid;

childpid = fork();
if (childpid == -1)
{
    perror("fork() failed");
    return 1;
}
if (childpid == 0)
    printf("I am a child with id %ld\n", (long)getpid());
else
    printf("I am a parent with id %ld\n", (long)getppid());
return 0;
```
wait()

- When a process creates a child, both parent and child proceed execution from the point of \texttt{fork()}
- The parent can execute \texttt{wait()} or \texttt{waitpid()} to block until the child executes
- \texttt{wait()} : waits for the termination of one of the children
- \texttt{waitpid()} : waits for the termination for specified child process
pid_t childpid;

int status;
childpid = fork();

if(childpid==1)
{
    perror("fork");
    exit(0);
}
else if(childpid==0){
    printf("I am a child\n");
    exit(3);
}
else {
    wait(&status);
    if(WIFEXITED(status)) {
        printf("child exited with status %d\n", WEXITSTATUS(status));
    }
}
exec()

- exec – execute a shell command or program
- Int execv(const char *path, char *const argv[]);
- Six of them – execl, execlp and execle form one family while execv, execvp and execve form the other
- *man them all* – On your own time!
execl.c

pid_t childpid;
    childpid = fork();
    if(childpid==-1){
        perror("Failed to fork");
        return 1;
    }
    //child code
    if(childpid == 0){
        execl("/bin/ps", "ps", "-af", NULL);
        perror("child failed to exec all_ids");
        return 1;
    }
    if(childpid != wait(NULL)){
        perror("parent failed to wait due to signal or error");
        return 1;
    }
}
execve.c

Notice the environment variable

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

int main(int argc, char **argv) {
    char *args[] = {"ps", "-af", 0, 0};

    char *const env[] = { "PATH=/bin",0,0};

    execve("/bin/ps", args, env);
    perror("execve");       /* if we get here, execve failed */
    exit(1);
}
```
Environment Variables

- These define the environment that the program would be running in.
- env prints all the environment variables on most of the shells.
- setenv helps you define new environment variables on shells like tcsh
- The bash equivalent of setenv is export
- cat /proc/<PID>/environ to see the environment variables of a particular process
Why care about env variables when they are already inherited by the child?

- Child inherits environment of its parent, but `execve()` called in the child process replaces the inherited code and data of the child with new program you specify. Hence the inherited env variables are lost.
- To overcome this issue, `execve()` takes a const pointer to the array of env variables as an argument

```c
char *args[] = {"ps", "-af", 0, 0};
char *const env[] = { "PATH=/bin",0,0};
execve("/bin/ps", args, env);
```
execvp.c

Default Environment is taken

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

int main(int argc, char **argv) {
    char *args[] = {"ps", "-af", 0, 0};

    printf("About to run ls\n");
    execvp("ps", args);
    perror("execvp");       /* if we get here, execvp failed */
    exit(1);
}
```
Debugging using GDB

• GDB (GNU Debugger) is the standard command-line debugger for the GNU operating system

<table>
<thead>
<tr>
<th>How do I?</th>
<th>GDB commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compile with debugging symbols</td>
<td>gcc –g –o helloWorld helloWorld.c</td>
</tr>
<tr>
<td>Run programs in GDB</td>
<td>gdb ./helloWorld</td>
</tr>
<tr>
<td>Restart program in debugger</td>
<td>kill run</td>
</tr>
<tr>
<td>Exit debugger</td>
<td>quit</td>
</tr>
</tbody>
</table>

• Compile source files with –g option to enable debugging
  - -g option tells compiler to put debug information in the object file
  - cc –g –o HelloWorld HelloWorld.c
  - gdb ./HelloWorld
Executing GDB

- GDB allows –
  - To execute and stop your program at specified points
  - Examine what has happened and inspect your program after stop-point
  - Make changes to variables and run

<table>
<thead>
<tr>
<th>How do I?</th>
<th>Gdb Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>Restart execution</td>
<td>continue</td>
</tr>
<tr>
<td>See where the program stopped</td>
<td>list</td>
</tr>
<tr>
<td>Step through code line-by-line</td>
<td>Next step</td>
</tr>
<tr>
<td>Examine variables</td>
<td>print</td>
</tr>
</tbody>
</table>
**GDB Stack-breakpoints-watchpoints**

<table>
<thead>
<tr>
<th>How do I?</th>
<th>Gdb Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get a backtrace</td>
<td>backtrace</td>
</tr>
<tr>
<td>Change stack frame</td>
<td>frame</td>
</tr>
<tr>
<td>Set breakpoint on line/function</td>
<td>Break line_number/function_name</td>
</tr>
<tr>
<td>Set watchpoint on variable</td>
<td>Watch variable</td>
</tr>
<tr>
<td>Get list of breakpoints/watchpoints</td>
<td>Info breakpoints</td>
</tr>
<tr>
<td>Disable breakpoints</td>
<td>Disable/clear breakpoints</td>
</tr>
</tbody>
</table>

To debug programs with multiple processes, use `set follow-fork-mode mode` where mode can be child or parent.
Valgrind

http://www.valgrind.org/downloads/current.html

It's a Linux toolset generally used for memory debugging

Sample Usage:
valgrind --tool=memcheck program_name

Splint

http://www.splint.org/download.html

This works on both Unix and non-Unix platforms

Splint is a tool for statically checking C programs for coding mistakes.
QUESTIONS?