Processes

What is a Processes?

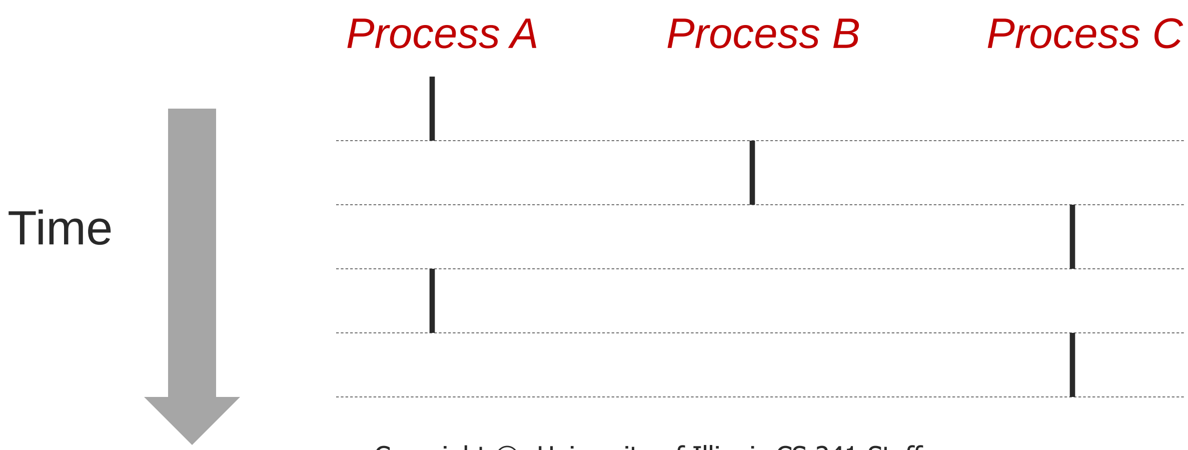
- Definition: A process is an instance of a running program
  - One of the most profound ideas in computer science
  - Not the same as “program” or “processor”
- Process provides each program with two key abstractions
  - Logical control flow
    - Each program seems to have exclusive use of the CPU
  - Private virtual address space
    - Each program seems to have exclusive use of main memory

What is a Processes?

- How are these illusions maintained?
  - Process executions interleaved (multitasking) or run on separate cores
  - Address spaces managed by virtual memory system

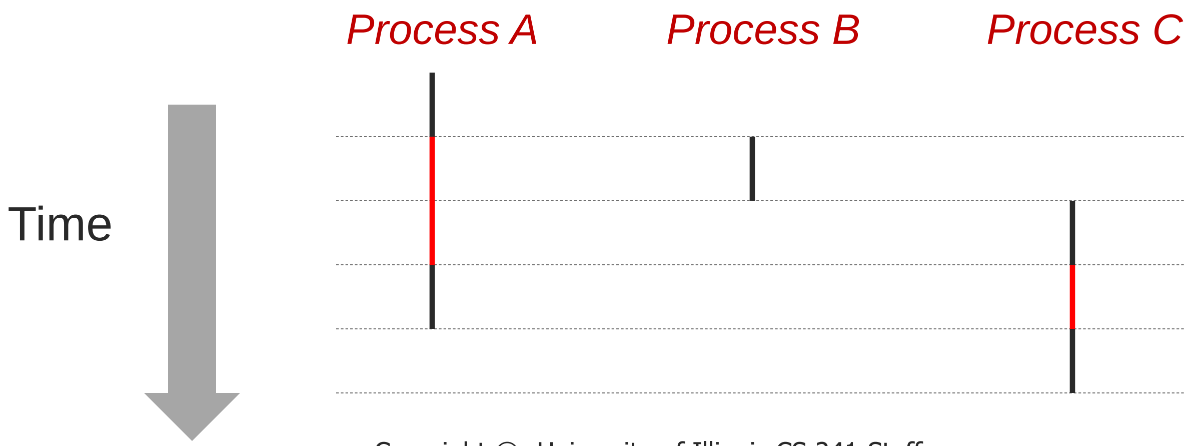
Concurrent Processes

- Two processes run concurrently (are concurrent) if their flows overlap in time
  - Otherwise, they are sequential
- Examples (running on single core)
  - Concurrent: A & B, A & C
  - Sequential: B & C



User View of Concurrent Processes

- Control flows for concurrent processes are physically disjoint in time
- However, we can think of concurrent processes as running in parallel with each other

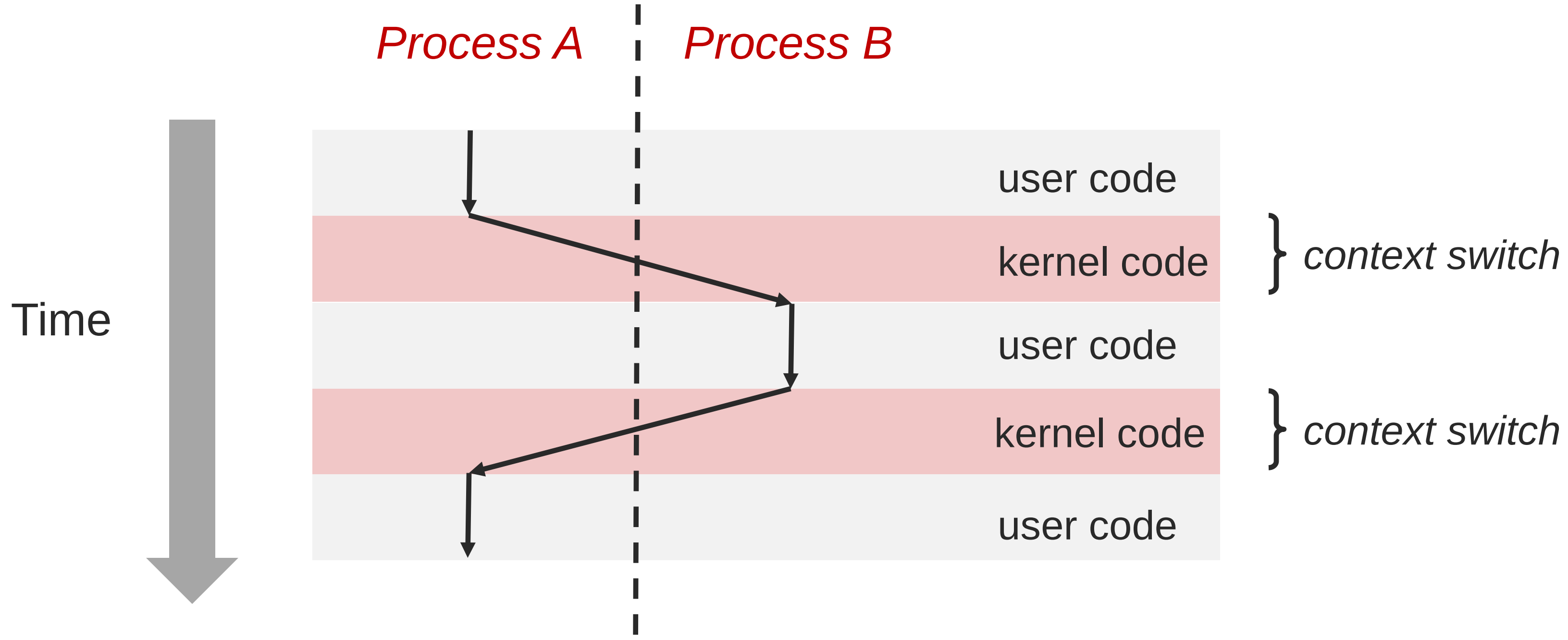


Program or Process?

- Process
  - A process is the *context* (the information/data) maintained for an executing program
    - An executable instance of a program
  - A program can have many processes
  - Each process has a unique identifier
- Unix processes
  - Process #1 is known as the 'init' process (root of the process hierarchy)

Context Switching

- Processes are managed by the kernel
  - Important: the kernel is not a separate process, but rather runs as part of some user process
- Control passes from one process to another via a context switch



What makes up a process?

- Program code
- Machine registers
- Global data
- Stack
- Open files
- An environment

Process Context

- Process ID (**`pid`**) — unique integer
- Parent process ID (**`ppid`**) — unique integer
- Current directory
- File descriptor table
- Environment — **`VAR=VALUE`** pairs
- Pointer to program code
- Pointer to data — Mem for global vars
- Pointer to stack — Mem for local vars
- Pointer to heap — Dynamically allocated memory
- Execution priority
- Signal information

Unix Processes

- Address space
  - The address space is a section of memory that contains the code to execute as well as the process stack
- Set of data structures in the kernel to keep track of that process
  - Address space map
  - Current status of the process
  - Execution priority of the process
  - Resource usage of the process
  - Current signal mask
  - Owner of the process

Process Lifetime

- Some processes run from system boot to shutdown
  - Servers & Daemons
    (e.g. Apache httpd server)
- Most processes come and go rapidly, as tasks start and complete
  - 'unit of work' on a modern computer
- A process can die a premature, even horrible death (say, due to a crash)

Know your process

- Each process has a unique identifier

```
int myid = getpid()
```

What is wrong with this?

Know your process

- better…

pid_t myid = getpid()

  - **pid_t: int** in linux,
  - **pid_t: long** in other systems

- Know your parent

pid_t myparentid = getppid()

Process Creation

- On creation, process needs resources
  - CPU, memory, files, I/O devices
- Get resources from the OS or from the parent process
  - Child process is restricted to a subset of parent resources
  - Prevents many processes from overloading system

Process Creation

- Execution options
  - Parent continues concurrently with child
  - Parent waits until child has terminated
- Address space options
  - Child process is duplicate of parent process
  - Child process has a new program loaded into it

Creating a Process – `fork()`

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

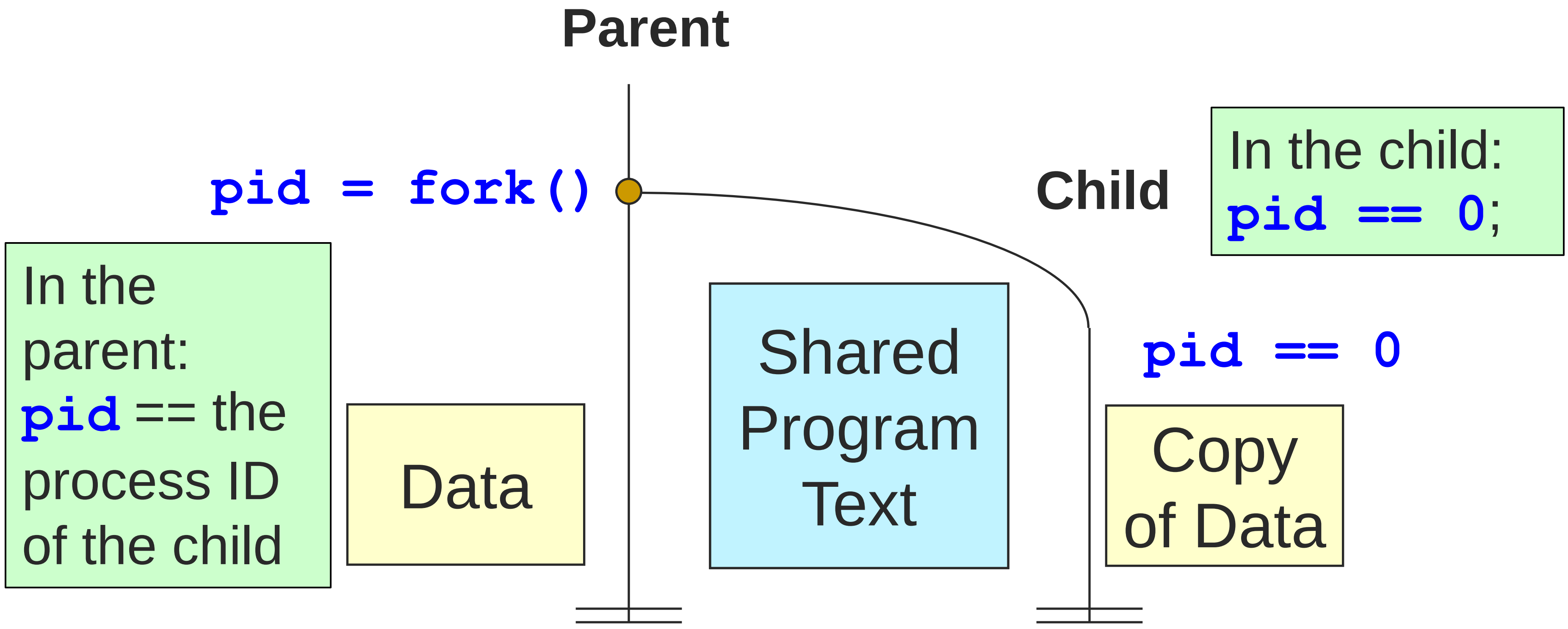
- Create a child process
  - The child is an (almost) exact copy of the parent
  - The new process and the old process both continue in parallel from the statement that follows the `fork()`
- Returns:
  - To child
    - 0 on success
  - To parent
    - process ID of the child process
    - -1 on error, sets `errno`

Understanding `fork()`

- Fork is interesting (and often confusing) because it is called once but returns twice

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

Creating a Process – `fork()`



A program can use this **`pid`** difference to do different things in the parent and child

How does `fork()` work?

- Parent

```
pid_t pid = fork();
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```

pid = m

```
hello from parent
```

- Child

```
pid_t pid = fork();
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```

pid = 0

```
hello from child
```

Which one is output first?

`fork()` Example #1

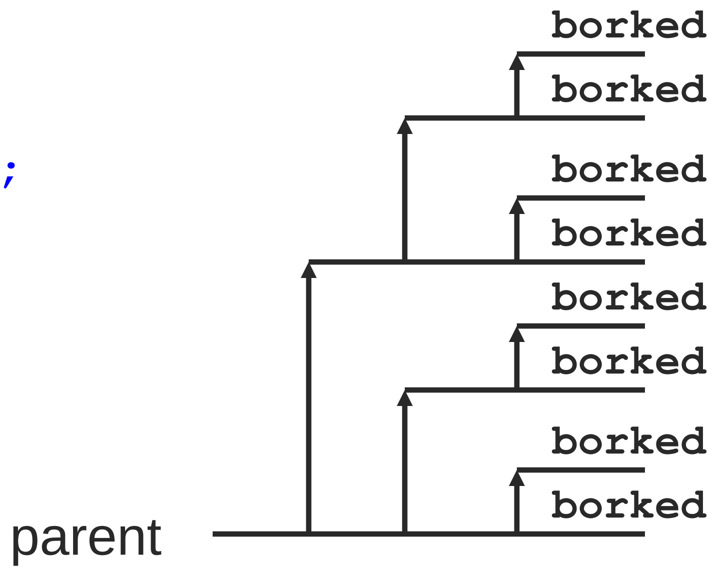
```
void fork1() {
    int x = 1;
    pid_t pid = fork();
    if (pid == 0) {
        printf("Child has x = %d\n", ++x);
    } else {
        printf("Parent has x = %d\n", --x);
    }
    printf("Bye from process %d with x = %d\n",
           getpid(), x);
}
```

- Both processes start with same state
  - Each has private copy
  - Including shared output file descriptor
- Relative ordering of their print statements (and so variable manipulations) is undefined

fork() Example #2

Three consecutive forks

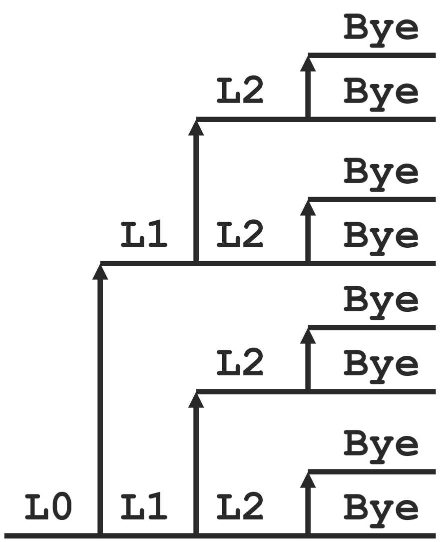
```
#define bork fork

void forkbork()
{
    bork(); bork(); bork();
    printf("borked\n");
}
```



fork() Example #3

```
void fork3()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("L2\n");
    fork();
    printf("Bye\n");
}
```

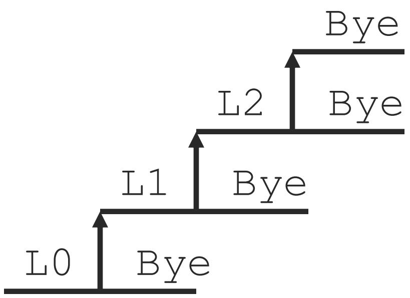
Three consecutive forks



fork() Example #5

Nested forks in children

```
void fork5()
{
    printf("L0\n");
    if (fork() == 0) {
       printf("L1\n");
       if (fork() == 0) {
           printf("L2\n");
           fork();
       }
    }
    printf("Bye\n");
}
```



How does `fork` really work?

- Parent
  `mypid = 4, myppid = 1`

```
int forked_pid , wait_pid;
int status = 0;

if (forked_pid = fork()) {
    /* parent */
    …..
    wait_pid = wait(&status);
} else {
    /* child */
    …..
    exit(status);
}
```

- Child
  `mypid = 6, myppid = 4`

```
int forked_pid , wait_pid;
int status = 0;
```

Copy-on-Write!

Another Example

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main() {
    pid_t pid;
    int i;
    pid = fork();
        if( pid > 0 ) {   /* parent */
        for( i=0; i < 1000; i++ )
        printf("\t\t\tPARENT %d\n", i);
    } else { /* child */
        for(i=0; i < 1000; i++)
                printf( "CHILD %d\n", i );
    }
    return 0;
}
```

What will the output be?

Possible Output

- `i` is copied between parent and child
- Switching between parent and child depends on many factors
  - Machine load, system process scheduling
- I/O buffering effects amount of output shown
- Output interleaving is nondeterministic
  - Cannot determine output by looking at code

When good processes die

Process Termination

- Upon completion of last statement
  - A process automatically asks the OS to delete it
  - All of the child's resources are de-allocated
  - Child process may return output to parent process
- Other termination possibilities: Aborted by parent process
  - Child has exceeded its usage of some resources
  - Task assigned to child is no longer required
  - Parent is exiting and OS does not allow child to continue without parent

Process Termination

- Voluntary termination
  - Normal exit
    - End of `main()`
  - Error exit
    - `exit(2)`
- Involuntary termination
  - Fatal error
    - Divide by 0, core dump / seg fault
  - Killed by another process
    - `kill` procID, end task

exit() Example

void exit(int status)

- Exits a process
- Normally return with status 0

atexit()

- Registers functions to be executed upon exit

```
void cleanup(void) {
   printf("cleaning up\n");
}

void fork6() {
   atexit(cleanup);
   fork();
   exit(0);
}
```

Zombies

- What happens on termination?
  - When process terminates, still consumes system resources
  - Entries in various tables & info maintained by OS
- Called a “zombie”
  - Living corpse, half alive and half dead

Zombies

- Reaping
  - Performed by parent on terminated child (using **`wait`** or **`waitpid`**)
  - Parent is given exit status information
  - Kernel discards process
- What if parent doesn't reap?
  - If any parent terminates without reaping a child, then child will be reaped by **`init`** process (**`pid == 1`**)
  - So, only need explicit reaping in long-running processes
    - e.g., shells and servers

Zombie Example

```
void fork7()
{
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID = %d\n",
               getpid());
        exit(0);
    } else {
        printf("Running Parent, PID = %d\n",
               getpid());
        while (1)
            ; /* Infinite loop */
    }
}
```

Zombie Example

```
void fork7() {
    if (fork() == 0) {
        /* Child */
        printf("Terminating
               getpid());
        exit(0);
    } else {
        printf("Running Pare
               getpid());
        while (1)
            ; /* Infinite lo
    }
}
```

```
Linux> ./forktest 7 &
[1] 8992
Terminating Child, PID = 8993
Running Parent, PID = 8992
Linux> ps
  PID TTY          TIME CMD
 8992 pts/1    00:00:06 forktest
 8993 pts/1    00:00:00 forktest <defunct>
 8994 pts/1    00:00:00 ps
29160 pts/1    00:00:00 bash
Linux> kill 8992
[1]+  Terminated              ./forktest 7
Linux> ps
  PID TTY          TIME CMD
 9004 pts/1    00:00:00 ps
29160 pts/1    00:00:00 bash
```

- **`ps`** shows child process as “defunct”
- Killing parent allows child to be reaped by **`init`**

Orphan Example

```
void fork8() {
    if (fork() == 0) {
        /* Child */
        printf("Running Child, PID = %d\n",
               getpid());
        while (1)
            ; /* Infinite loop */
    } else {
        printf("Terminating Parent, PID = %d\n",
               getpid());
        exit(0);
    }
}
```

Orphan Example

```
void fork8() {
    if (fork() == 0) {
        /* Child */
        printf("Running Chil
               getpid());
        while (1)
            ; /* Infinite lo
    } else {
        printf("Terminating
               getpid());
        exit(0);
    }
}
```

```
Linux> ./forktest 8
Running Child, PID = 9413
Terminating Parent, PID = 9412
Linux> ps
  PID TTY          TIME CMD
 9413 pts/1    00:00:07 forktest
 9416 pts/1    00:00:00 ps
29160 pts/1    00:00:00 bash
Linux> kill 9413
Linux> ps
  PID TTY          TIME CMD
 9422 pts/1    00:00:00 ps
29160 pts/1    00:00:00 bash
```

- Child process still active even though parent has terminated
- Must kill explicitly, or else will keep running indefinitely

Waiting for a child to finish – `wait()`

```
#include <sys/types.h>
#include <wait.h>
pid_t wait(int *status);
```

- Suspend calling process until child has finished
- Allow parent to reap child
- Returns:
  - Process ID of terminated child on success
  - -1 on error, sets **`errno`**
- Parameters:
  - **`status`**: status information set by **`wait`** and evaluated using specific macros defined for **`wait`**.

Waiting for a child to finish – `wait()`

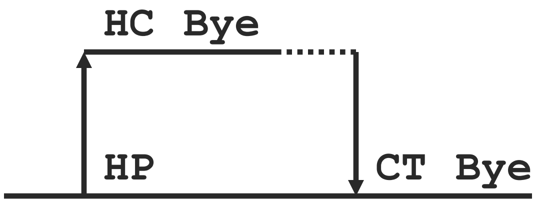
```
#include <sys/types.h>
#include <wait.h>
pid_t wait(int *status);
```

- Suspend calling process until child has finished
- Allow parent to re
- Returns:
  - Process ID of ter
  - -1 on error, sets
- Parameters:
  - **`status`**: status information set by **`wait`** and evaluated using specific macros defined for **`wait`**.

Professional code uses signal handler (later lecture) for signal **`SIGCHLD`** which issues a **`wait()`** call

Waiting for a child to finish

```
void fork9() {
   int child_status;

   if (fork() == 0) {
      printf("HC: hello from child\n");
   }
   else {
      printf("HP: hello from parent\n");
      wait(&child_status);
      printf("CT: child has terminated\n");
   }
   printf("Bye\n");
   exit();
}
```



Waiting for any child to finish

If multiple children complete, they are taken in an arbitrary order

Can use macros **WIFEXITED** and **WEXITSTATUS** to get information about exit status

```
void fork10() {
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
   if ((pid[i] = fork()) == 0)
        exit(100+i); /* Child */
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
   if (WIFEXITED(child_status))
        printf("Child %d terminated with exit status %d\n",
           wpid, WEXITSTATUS(child_status));
   else
        printf("Child %d terminate abnormally\n", wpid);
    }
}
```

wait() Function

- Allows parent process to wait (block) until child finishes
- Causes the caller to suspend execution until child's status is available

errno	cause
ECHILD	Caller has no unwaited-for children
EINTR	Function was interrupted by signal
EINVAL	Options parameter of waitpid was invalid

Waiting for a child to finish – `waitpid()`

```
#include <sys/types.h>
#include <wait.h>
pid_t waitpid(pid_t pid, int *status, int
   options);
```

- Suspend calling process until child specified by **`pid`** has finished
- Returns:
  - Process ID of terminated child on success
  - 0 if **`WNOHANG`** and no child available, sets **`errno`**
  - -1 on error, sets **`errno`**
- Parameters:
  - **`status`**: status information set by **`wait`** and evaluated using specific macros defined for **`wait`**.

Waiting for a child to finish – `waitpid()`

```
#include <sys/types.h>
#include <wait.h>
pid_t waitpid(pid_t pid, int *status, int
   options);
```

- Suspend calling process until child specified by `pid` has finished
- Parameters:
  - `pid`:
    - < -1: wait for any child process whose process group ID is equal to the absolute value of `pid`.
    - -1 wait for any child process (same as `wait`)
    - 0 wait for any child process whose process group ID is equal to that of the calling process.
    - > 0 wait for the child whose process ID is equal to the value of `pid`.

Waiting for a child to finish – `waitpid()`

```
#include <sys/types.h>
#include <wait.h>
pid_t waitpid(pid_t pid, int *status, int
   options);
```

- Suspend calling process until child specified by **`pid`** has finished
- Parameters:
  - **`options`**:
    - **`WNOHANG`**: return immediately if no child has exited.
    - **`WUNTRACED`**: return for children that are stopped, and whose status has not been reported.

Waiting for a child to finish – **waitpid()**

```
void fork11() {
    pid_t pid[N];
    int i;
    int child_status;

    for (i = 0; i < N; i++)
    if ((pid[i] = fork()) == 0)
        exit(100+i); /* Child */
    for (i = N-1; i >= 0; i--) {
        pid_t wpid = waitpid(pid[i], &child_status, 0);
    if (WIFEXITED(child_status))
        printf("Child %d terminated with exit status %d\n",
            wpid, WEXITSTATUS(child_status));
    else
        printf("Child %d terminated abnormally\n", wpid);
    }
}
```

How to List all Processes?

- On Windows: run Windows task manager
  - Hit Control+ALT+delete
  - Click on the “processes” tab
- On UNIX
  - **`> ps -e`** also, **`pstree`**
  - Try “**`man ps`**”

Example – fork()

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main() {
    pid_t pid;              /* could be int */
    int i;
    pid = fork();
    if( pid > 0 ) {         /* parent */
        for( i=0; i < 1000; i++ )
        printf("\t\t\tPARENT %d\n", i);
    }
    else { /* child */
        for(i=0; i < 1000; i++)
                printf( "CHILD %d\n", i );
    }
    return 0;
}
```

How can you use **ps** to see the processes that are created?