2- processes

program = countable set of ordered and sequential instructions that needs to be executed to accomplish a task

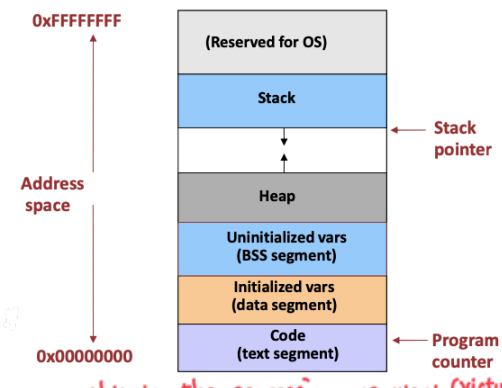
process = execution part of a program, provides programs two abstractions (CPU and mem)

① logical control flow = illusion of having own CPU → with interleaved executions of processes (multitasking)

② private virtual address = illusion of having own main memory → with managing address space by virtual memory system

process =

address space
code, vars, heap
libraries, stack
memory that process can access



+ processor/CPU state
general purpose registers, program counter, stack pointer.. (%rax, %rip...)
the CPU registers associated with the running process

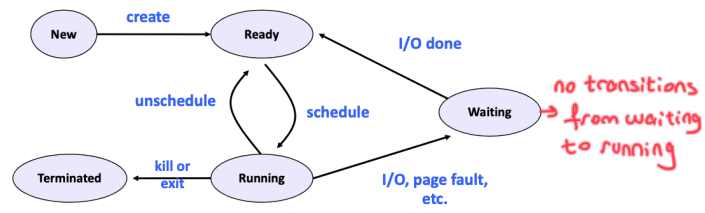
+ OS resources
open files, network sockets
OS states associated with the process

- this is the process's own view (virtual mem illusion)
- physical mem much more less

execution states of a process = running = currently using CPU

↳ ready = waiting to be assigned to CPU

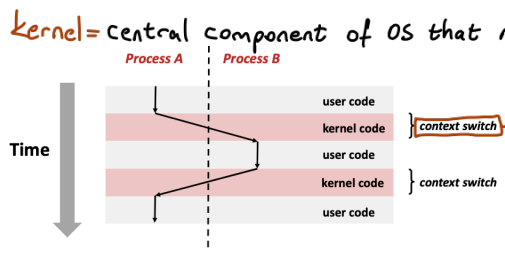
↳ waiting/sleeping = waiting for an event (I/O, timer)



process control block = OS maintains a pcb for each process

↳ process id + user id + execution state + saved cpu state + OS resources + mem management info + scheduling priority + accounting information
↳ total cpu time, mem usage ...

kernel = central component of OS that manages operations of computer and hardware → processes are managed by kernel



→ passing control flow from one process to another

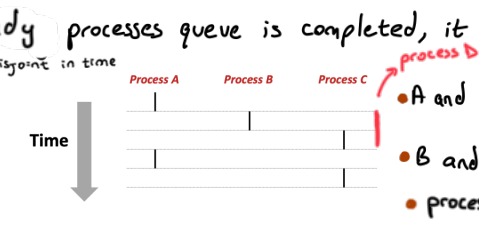
- uses many clock cycles and memory

state queues = OS maintains separate queues for ready and each kind of waiting processes (ex: I/O queue)

↳ pcbs move between these queues, when ready processes queue is completed, it moves to waiting queue to ready queue

concurrent processes = if two processes' flows overlap in time (but they are disjoint in time)

↳ otherwise they are sequential



- A and B, A and C are concurrent
- B and C are sequential
- process D and B are parallel → cannot do parallel if there is a single core

creating processes

↳ parent process creates a new running child process by calling fork(system call), they run concurrently

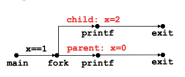
int fork (void) = returns 0 to child process

↳ returns child's pid to parent process (returns -1 if failed) pid = process id

- child get an identical but separate copy of the parents virtual address space and parent's open file descriptors → copy of address space + cpu state + OS resources
  ↳ they share the same physical memory, until one of them makes a change
- called ones but returns twice

```c
int main(int argc, char** argv)
{
    pid_t pid;
    int x = 1;

    pid = fork();
    if (pid == 0) {  /* Child */
        printf("child : x=%d\n", ++x);
        return 0;
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    return 0;
}
```
fork.c

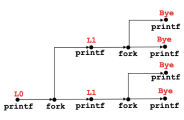
child: 2
parent: 0
or
parent: 0
child: 2



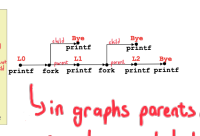

```c
int main(int argc, char** argv)
{
    pid_t pid;
    int x = 1;

    pid = fork();
    if (pid == 0) {  /* Child */
        printf("child : x=%d\n", ++x);
        printf("child : x=%d\n", ++x);
        return 0;
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    printf("parent: x=%d\n", --x);
    return 0;
}
```
fork.c

one of orders ↓
parent: 0
child : 2
child : 3
parent: -1

```c
void fork2()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```
forks.c

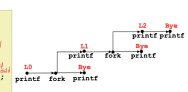



```c
void fork4()
{
    printf("L0\n");
    if (fork() != 0) {
        printf("L1\n");
        if (fork() != 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}
```
forks.c



↳ in graphs parents are always at bottom

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



terminating processes

↳ calling the exit func ⇒ void exit (int status) → status=0 normal, status≠0 error → exit(0); called once but never returns → because function that called is no longer exist

↳ returning from the main

↳ receiving a signal whose default action is to terminate ctrl+c

atexit(func_name) = the func pointed by atexit is automatically called without arguments when the program terminates normally

↳ returns zero if the function registration is successful, nonzero else

```c
void first()
{
    cout << "Exit first" << endl;
}

// Executed third
void second()
{
    cout << "Exit Second" << endl;
}

// Executed Second
void third()
{
    cout << "Exit Third" << endl;
}

// Executed first
void fourth()
{
    cout << "Exit Fourth" << endl;
}
// Driver Code
int main()
{
    int value1, value2, value3, value4;
    value1 = atexit(first);
    value2 = atexit(second);
    value3 = atexit(third);
    value4 = atexit(fourth);
    if ((value1 != 0) or (value2 != 0) or
        (value3 != 0) or (value4 != 0)) {
        cout << "atexit() function registration Failed" << endl;
        exit(1);
    }
    // Executed at the starting
    cout << "Registration successful" << endl;
    return 0;
}
```

output
Registration successful
Exit Fourth
Exit Third
Exit Second
Exit first
} stack order

- when process terminates it still consumes system resources (exit status, os tables) → zombie

```c
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

```c
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);
    }
}
```
non-terminating child

reaping child processes = performed by parent on terminated child using wait or waitpid (↳ waiting for specific), kernel then deletes child.

↳ child died but parent is still alive, child's data is not deleted

↳ if parent terminates without reaping a child, the child will be reaped by init process (pid=1)

↳ so only need explicit reaping in long-running processes → shells, servers

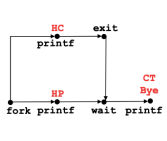
int wait (status *child_status) = suspends current parent process until one of its children terminates

- return value is pid of the child process that terminated
- if child_status != NULL, then the int it points to will be set to a val that indicates the reason the child terminated and the exit status

process completion status

- int WIFEXITED (int status)
  - returns a nonzero value if the child process terminated normally with exit or _exit.
- int WEXITSTATUS (int status)
  - If WIFEXITED is true of *status*, this macro returns the low-order 8 bits of the exit status value from the child process.
- int WIFSIGNALED (int status)
  - returns a nonzero value if the child process terminated because it received a signal that was not handled
- int WTERMSIG (int status)
  - If WIFSIGNALED is true of *status*, this macro returns the signal number of the signal that terminated the child process.
- int WCOREDUMP (int status)
  - Returns a nonzero value if the child process terminated and produced a core dump.
- int WIFSTOPPED (int status)
  - returns a nonzero value if the child process is stopped.
- int WSTOPSIG (int status)
  - If WIFSTOPPED is true of *status*, this macro returns the signal number of the signal that caused the child process to stop.

```c
void fork9() {
    int child_status;

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




```c
void fork10() {
    pid_t pid[N];
    int i, child_status;

    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0) {
            exit(100+i); /* Child */
        }
    for (i = 0; i < N; i++) { /* Parent */
        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);
    }
}
```
forks.c

→ all children exits arbitrary, but they are zombie until parent is terminated

→ parent access them in order

loading and running programs

loads and run in the current process

int execve (char * filename, char * argv[], char * envp[]) → called ones but never returns, if no error → returns -1

(base function; argv: argument list; envp: environment var list)

- argv[0] = filename by convension,
- envp: in the form "name=value" (ex: user=droh) putenv, getenv, printenv
- overwrites code, data, and stack → retains pid, open files and signal context
- does not fork a new process, it replaces the address space and CPU state of the current process
- loads the new address space from the executable file and starts it from main()
- ★ so start a new program, use fork() followed by execve()

int execl (char * path, char * arg0, char * arg1, ... , 0)

(arg0: name of the executable object file)

- environment taken from char **environ

environment variable = dynamic named val that can affect the way running process will behave on a computer

exec() func family

- The suffix's determine the arguments
  - l : arguments are passed as a list of strings to the main()
  - v : arguments are passed as an array of strings to the main()
  - p : path/s to search for the new running program
  - e : the environment can be specified by the caller
- One can mix them with different combinations
  - int execl(const char *path, const char *arg, ...);
  - int execlp(const char *file, const char *arg, ...);
  - int execle(const char *path, const char *arg, ..., char * const envp[]);
  - int execv(const char *path, char *const argv[]);
  - int execve(const char *path, char *const argv[], char *const envp[]);
  - int execvp(const char *file, char *const argv[]);
  - int execvpe(const char *file, char *const argv[], char *const envp[]);
- The initial argument is always the name of a file to be executed.

```c
int main(int argc, char **argv) {
    int rv;
    if (fork()) {   /* Parent process */
        wait(&rv);
    } else {        /* Child process */
        char *newargs[3];
        printf("Hello, I am the child process.\n");
        newargs[0] = "/bin/echo";   /* Convention! Not required!! */
        newargs[1] = "some random string";
        newargs[2] = NULL;          /* Indicate end of args array */
        if (execv("/bin/echo", newargs)) {
            printf("warning: execve returned an error.\n"); exit(-1);
        }
        printf("Child process should never get here\n");
        exit(42);
    }
}
```