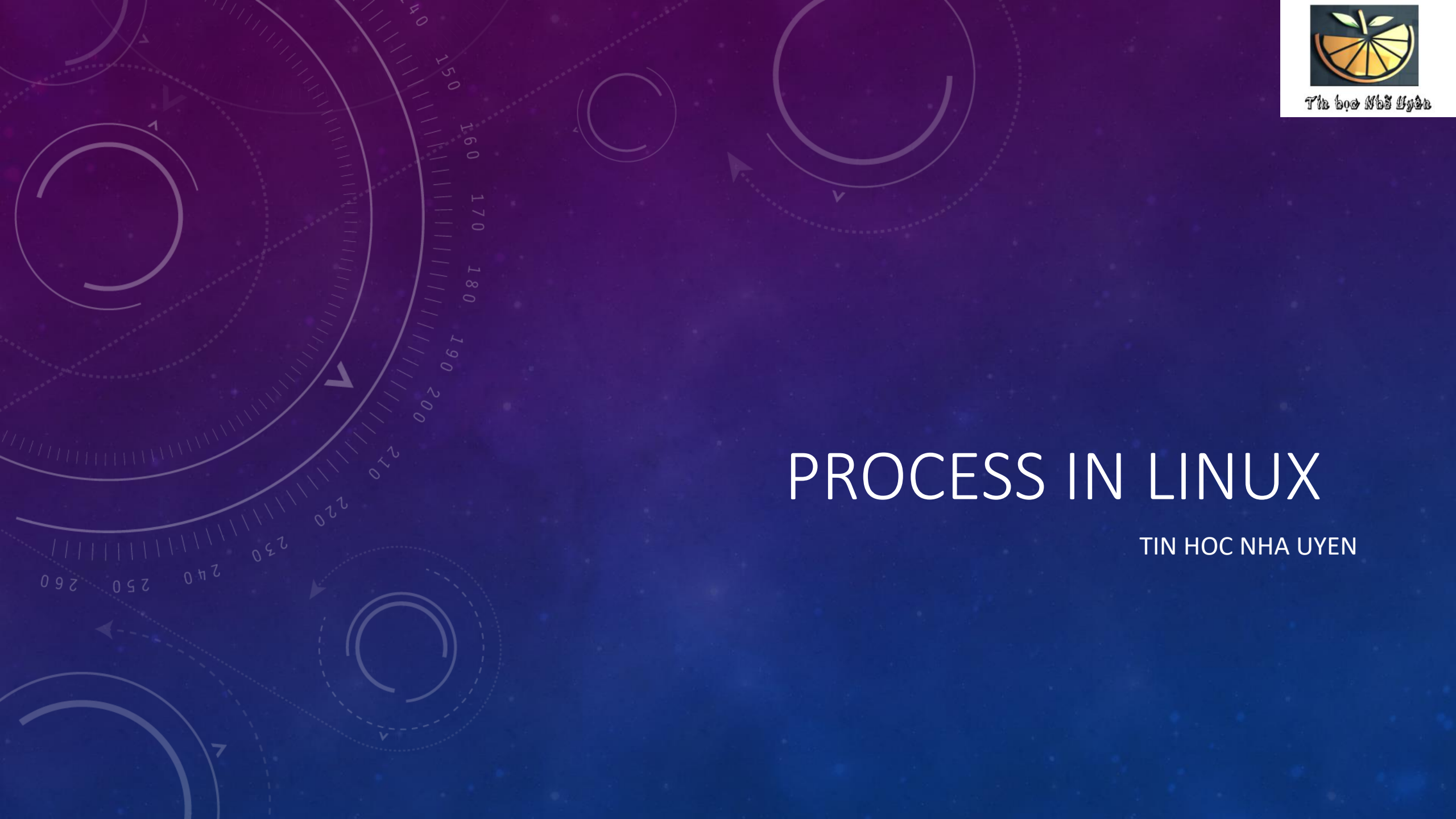


PROCESS IN LINUX

TIN HOC NHA UYEN

CONTENT

- PROCESS ID
- PROCESS MEMORY LAYOUT
- VIRTUAL MEMORY MENAGEMENT
- STACK AND STACK FRAME
- COMMAND LINE ARGUMENT
- ENVIRONMENT VARIABLES
- PROCESS CREATION
- PROCESS TERMINATION
- MONITORING CHILD PROCESS
- PROGRAM EXECUTION

PROCESS ID AND PARENT PROCESS ID

- Process ID
  - Uniquely identifies the process on the system
  - Used by varieties of system call like kill
  - Limited to 32767
- Parent Process ID
  - Process create child process
  - pstree

```
pid_t getpid(void);
```

```
pid_t getppid(void);
```

```
pstree command
```

PROCESS MEMORY LAYOUT

- Text segment
  - Machine-language instructions of program run by the process
  - Read-only area that process don't accidentally modify it own intructions via bad pointer access
  - When multiple process run the same program, text segment shareable so that a single copy of program code can be mapped into virtual address of all of processes
- Initialized data segment
  - Global and static variables that explicitly initialized
- Uninitialized data segment
  - Global and static variables that explicitly initialized

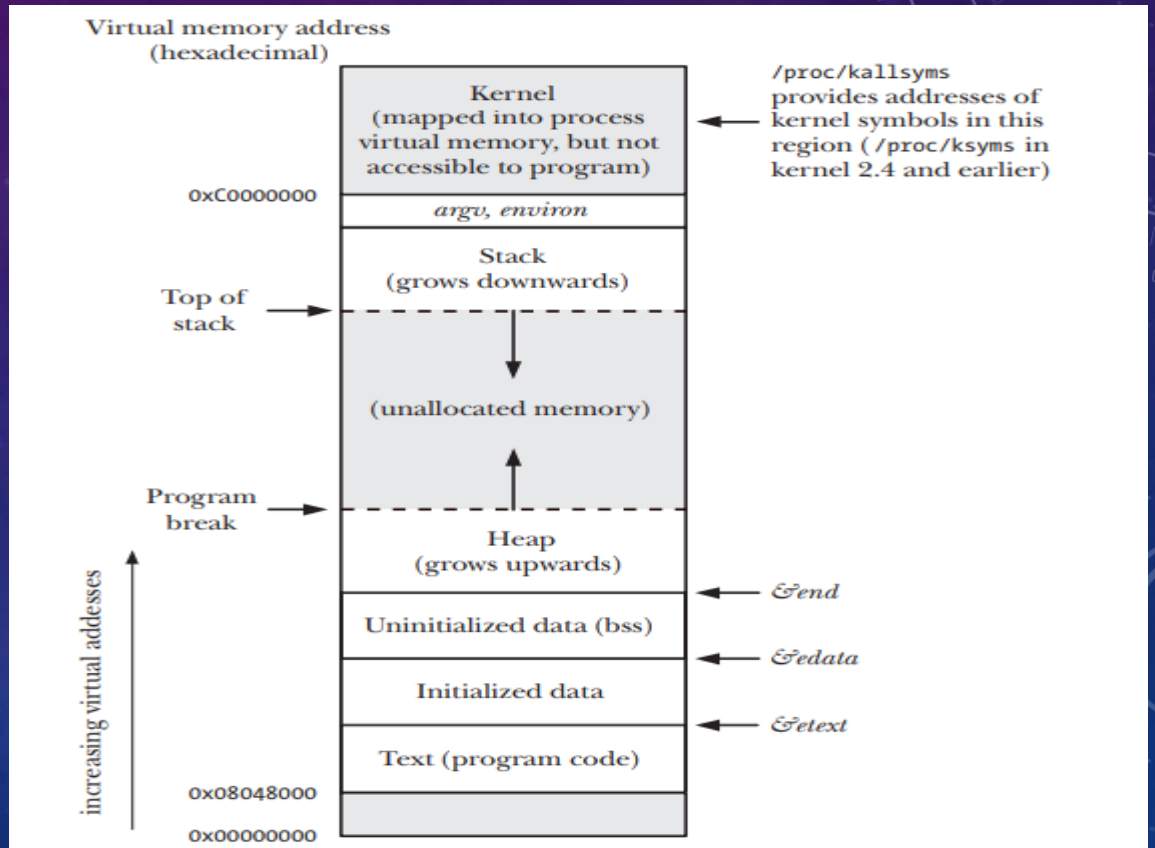


Figure 6-1: Typical memory layout of a process on Linux/x86-32

PROCESS MEMORY LAYOUT

- Stack
  - Dynamically growing and shrinking segment containing stack frame
  - One stack frame is allocated for each currently called function
  - One frame store local variables (automatic variables), argument and return value
- Heap
  - Dynamically allocation by run time
  - Size command

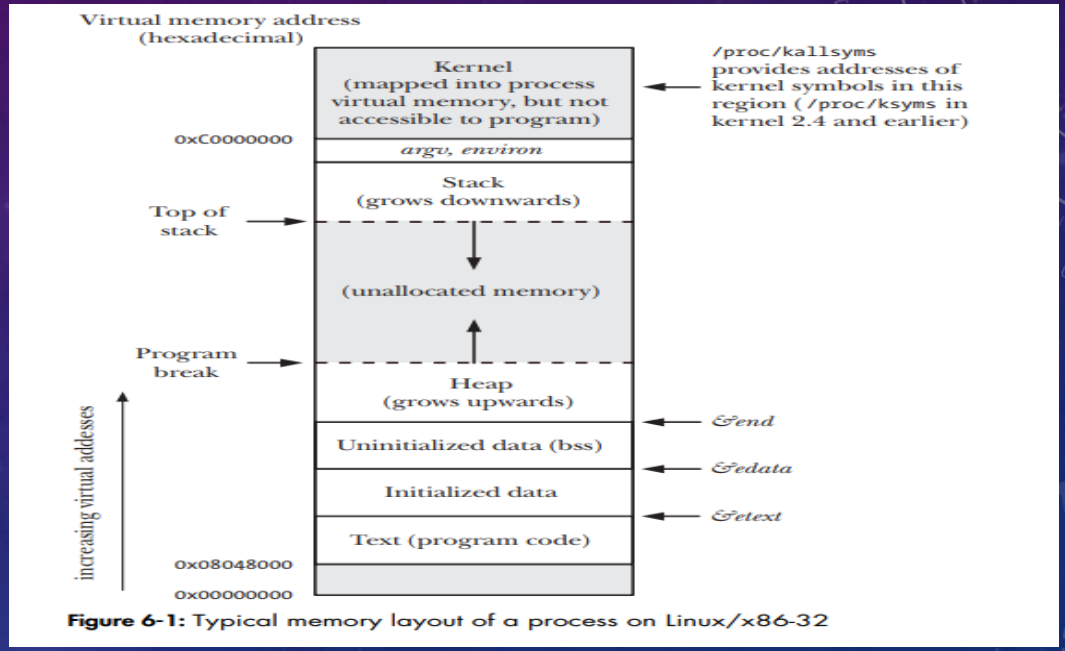


Figure 6-1: Typical memory layout of a process on Linux/x86-32

```
thientran@ubuntu:~/Desktop/training/process$ size bin_stack
   text    data     bss     dec     hex filename
   1198     292 1089568 1091058  10a5f2 bin_stack
```

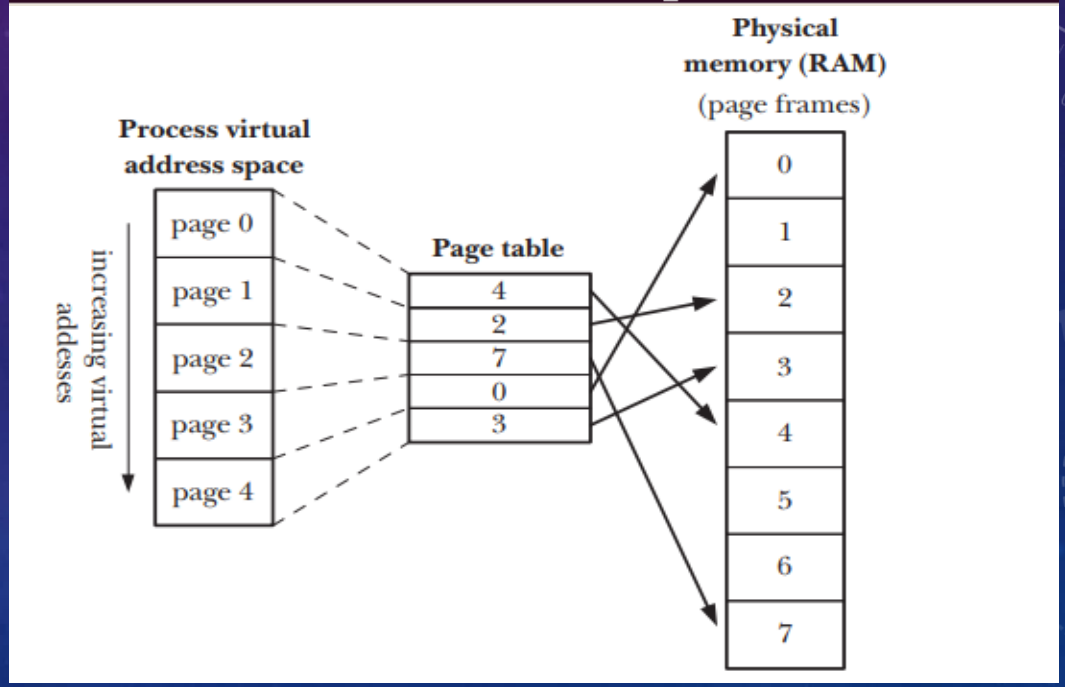
VIRTUAL MEMORY MANAGEMENT

- To make efficient use of both the CPU and RAM by exploiting a typical property of most programs : Locality of refrence
- Two kinds of locality
  - Spatial locality
    - The tendency of program to reference memory addresses that recently access
  - Temporal locality
    - The tendency of program to reference memory addresses that access recent past (due to loop)

VIRTUAL MEMORY MANAGEMENT

- Virtual memory split memory used by each program into small fixed size unit called pages
- Only some of the pages of a program need to be resident in physical memory page frames, these pages is called by resident-set
- A reserved-area of disk space used to supplement the computer RAM is called swap-area
- When a process references a page that is not currently resident in physical memory , a page fault occurs, kernel suspends execution of process while the page is loaded from disk into memory

```
thientran@ubuntu:~/Desktop/training/process$ getconf PAGESIZE
4096
thientran@ubuntu:~/Desktop/training/process$ getconf PAGE_SIZE
4096
thientran@ubuntu:~/Desktop/training/process$
```



VIRTUAL MEMORY MANAGEMENT

- Page table
  - Describe location of each page in process virtual address space
  - Each entry in page table indicate it is currently reside on RAM or Disk
  - Not all address ranges in process virtual address virtual space require page-table entries, it is not necessary to maintain corresponding page-table entries
  - If process access in a address that is not corresponding page table entry. It receive a SIGSEGV

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

int main(int argc, char *argv[])
{
    char buf[5];

    buf[0] = 0x45;
    buf[2] = 0x45;
    buf[5000] = 0x49;

    printf("%x ", buf[5000]);
    exit(EXIT_SUCCESS);
}
```

VIRTUAL MEMORY MANAGEMENT

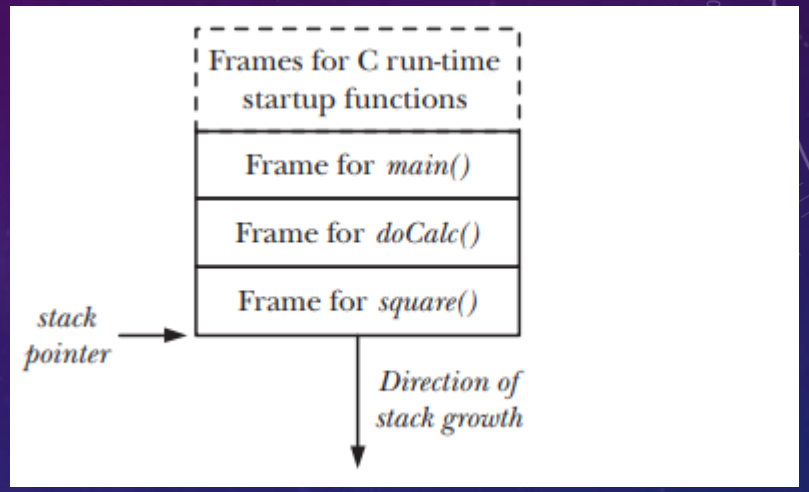
- Process range of valid virtual addresses
  - Stack grows downward beyond limits
  - Memory allocated or deallocated on the heap
    - malloc or free
  - Shared memory attach or detached
    - shmat or shmdt
  - Memory mapping mapped or unmapped
    - mmap or munmap

VIRTUAL MEMORY MANAGEMENT

- Advantages
  - Process isolated from other
    - One process cannot read or write memory of other process or kernel
    - Accomplished by having the page-table entries for each process point to distinct sets of physical pages in RAM
  - When appropriate, two or more process can share memory
    - Multi-process execute the same program can share single copy of program code as multiple process run the same binary or load the same share library
    - IPC to communication between process
    - Accomplished by having page-tables entries in difference process refer to same page of RAM
  - Easy to implement memory protection
    - by allow some processes can readable, writeable or executable on some page-table entries
  - Programmer don't need to concern about physical layout of program
  - All part of program reside in memory then program run faster and memory capacity exceed physical memory

STACK AND STACK FRAME

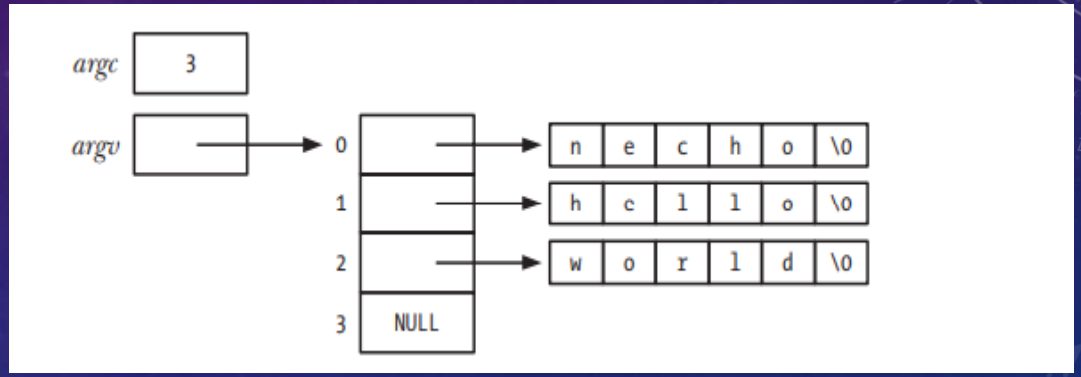
- Stack pointer
  - Special purpose register track the current top of stack
  - Each time a function is called , an additional frame is allocated on stack, this frame is remove when function returns
- User stack
  - Include function argument and local variables
    - Automatic variable vs static variable
  - Call linkage information
    - Each function use certain CPU register (PC, EAX), each a function call another , a copy of these registers saved in the called function stack frame so that function return
    - A function call itself recursively, it has multiple stack frame




```
(gdb) bt
#0  0x00110422 in __kernel_vsyscall ()
#1  0x00894651 in raise () from /lib/tls/i686/cmov/libc.so.6
#2  0x00897a82 in abort () from /lib/tls/i686/cmov/libc.so.6
#3  0x08048469 in square (x=9973) at stack.c:11
#4  0x0804847a in docalc (val=9973) at stack.c:22
#5  0x080484dc in main (argc=1, argv=0xbfae0084) at stack.c:44
(gdb)
```

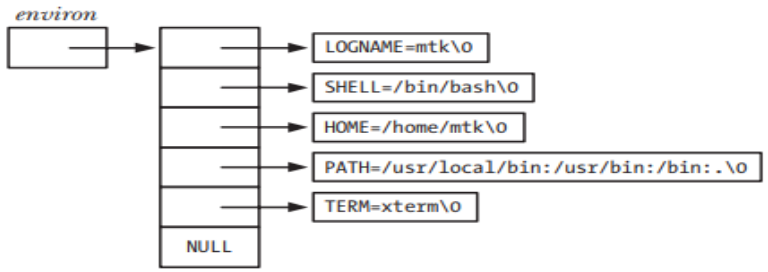
COMMAND LINE ARGUMENT

- main(int argc, char *argv[])
- argc
  - How many command line argument there are
- argv
  - argv[0] is name of program
  - convert argument to number by strtol or atoi
  - use getopt library
- See command line argument of a process
  - /proc/PID/cmdline



ENVIRONMENT VARIABLES

- Environment list
  - Array of strings associated for process
  - When a process created, it inherit a copy of parent environment
  - A common use of environment variables is in the shell. By placing values in shell environment, the shell ensure it will pass to process that execute in the shell
- export command
  - export
  - export PATH=$PATH:/path_to_dir
  - bashrc
- PATH and LD_LIBRARY_PATH environment
- environ global variables
- APIs




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

extern char **environ;
int main(int argc, char *argv[])
{
    char **ep;
    for (ep = environ; *ep != NULL; ep++)
        puts(*ep);

    exit(EXIT_SUCCESS);
}
```

```
char *getenv(const char *name);
```

```
int setenv(const char *name, const char *value, int overwrite);
```

```
int unsetenv(const char *name);
```

```
int clearenv(void)
```

PROCESS CREATION- OVERVIEW OF BASIC SYSTEM CALL

- fork()
  - Allow parent process create child process
  - Child obtains copies of the parent's stack, data, heap and text segments
- exit(status)
  - Terminate a process, making all resources like memory, open file descriptor used by process now available for kernel reallocation
  - status determines the termination status for process
- wait(status)
  - If process is not terminated , wait will suspends execution of the process until one of children has terminated
  - If process terminated, wait will return immediately with child termination status
- execve(pathname, argv, envp)
  - Loads program into process memory with argument argv and environment list envp
  - The existing program text is discarded, the stack, data, heap segments are freshly created

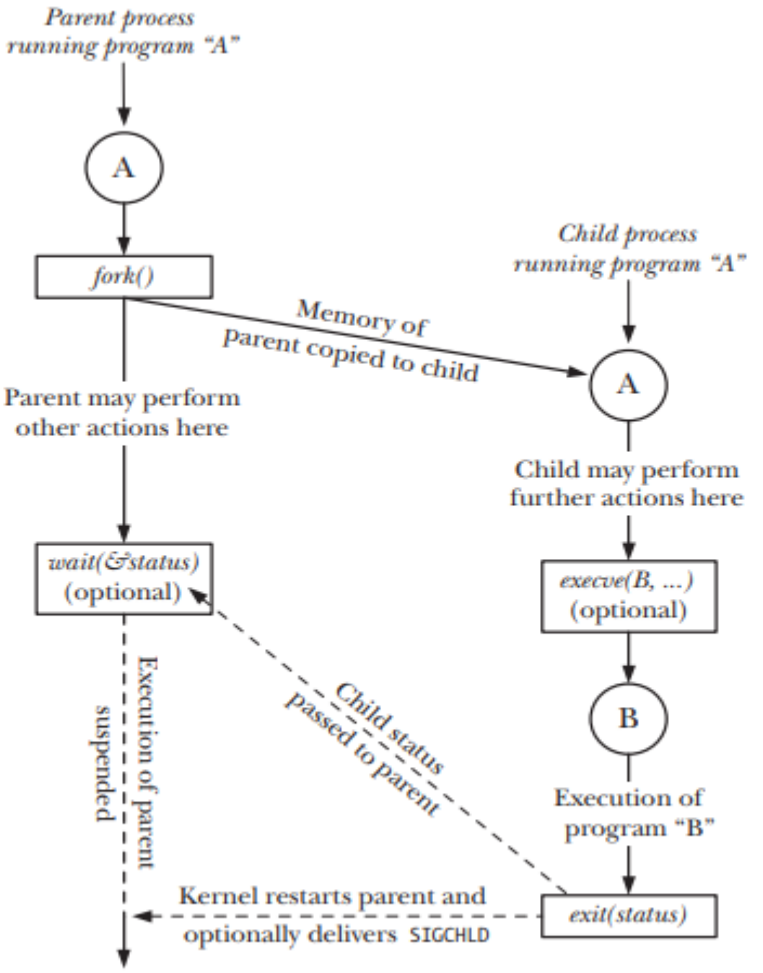


Figure 24-1: Overview of the use of *fork()*, *exit()*, *wait()*, and *execve()*

CREATING NEW PROCESS WITH FORK

- Creating new process is useful way of dividing tasks as network server receive request from multi-clients
- fork creates a new process, the child, that is an almost exact duplicate of calling process, the parent. When fork completed, two process exist and execution of two process is continuing
- Two processes are executing the same program text but they have separate copies of stack, data and heap segment
- The child's stack data and heap segment are initially exact duplicates of corresponding part of parent memory.
- After fork, each process can modify the variables in stack, data, heap segment without affecting the other process
- fork will return child process id to parent process, 0 to child process and -1 if something error

```
pid_t fork(void);
```

In parent: returns process ID of child on success, or −1 on error;
in successfully created child: always returns 0

```
pid_t childPid;             /* Used in parent after successful fork()
                               to record PID of child */
switch (childPid = fork()) {
case -1:                    /* fork() failed */
    /* Handle error */

case 0:                     /* Child of successful fork() comes here */
    /* Perform actions specific to child */

default:                    /* Parent comes here after successful fork() */
    /* Perform actions specific to parent */
}
```

FILE SHARING BETWEEN PARENT AND CHILD

- After fork, the child receives duplicate of all of the parent file descriptor, it mean parent and child refer to the same open file descriptor
- All attributes of open file is shared between the parent and child
- For example, the child modify file offset, this change is visible through the corresponding descriptor in parent
- If the parent and the child both writing the file, sharing offset ensure that the two processes don't override each other's output
- However it is not prevent the output of two process es from being randomly intermingled
- The code should close unused descriptor

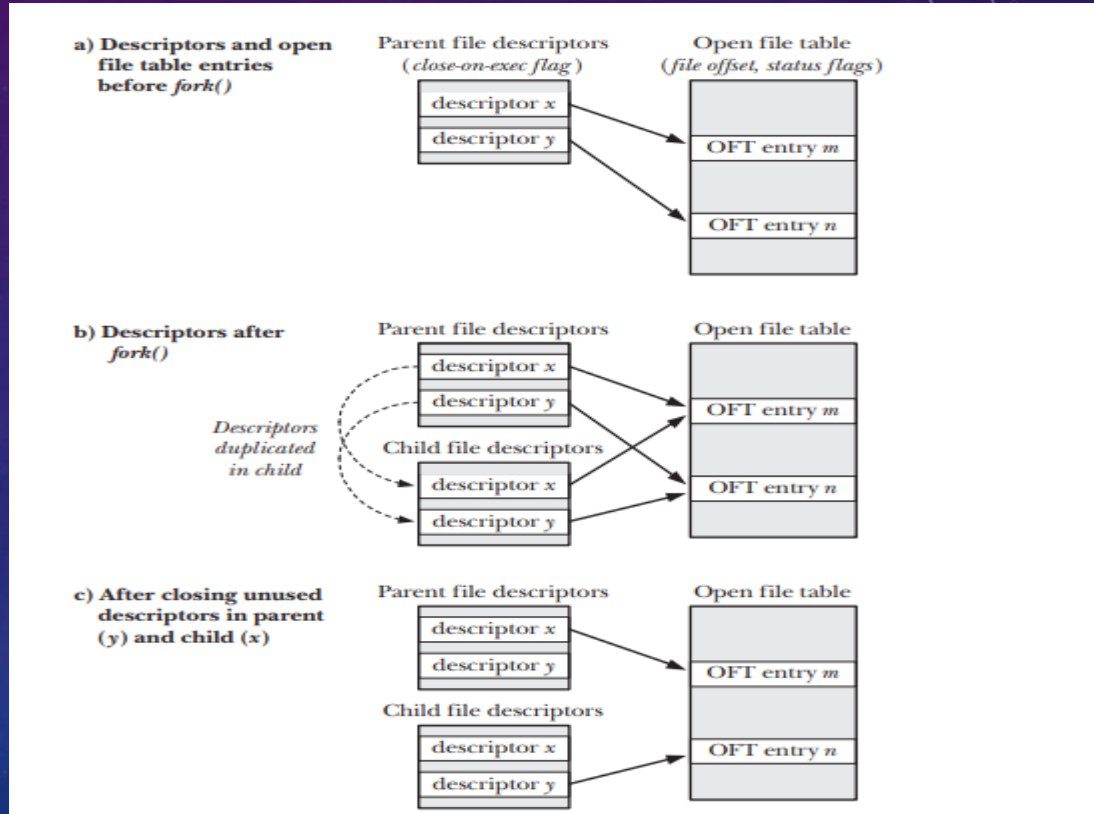
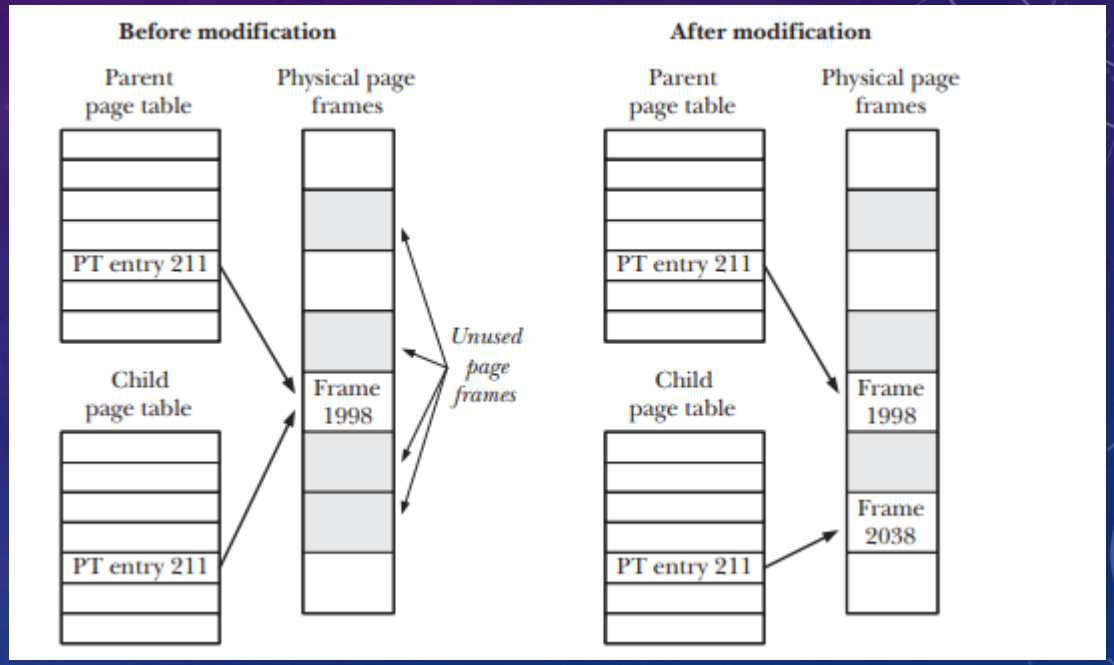


Figure 24-2: Duplication of file descriptors during *fork()*, and closing of unused descriptors

MEMORY SEMATIC OF FORK

- The wasteful copy of parent text, data, heap and stack
  - Especially after fork is exec
  - Exec will discard current text, data, heap, stack and replace by its own program
- Kernel use some techniques to avoid such wasteful copying
  - With text segment, this area is read-only, so that process cannot modify its own text segment, then
  - All entries in per-process per-tables in child refer to same virtual memory page frame already used by parent
  - For pages in data, heap and stack, kernel employs copy-on-write technique

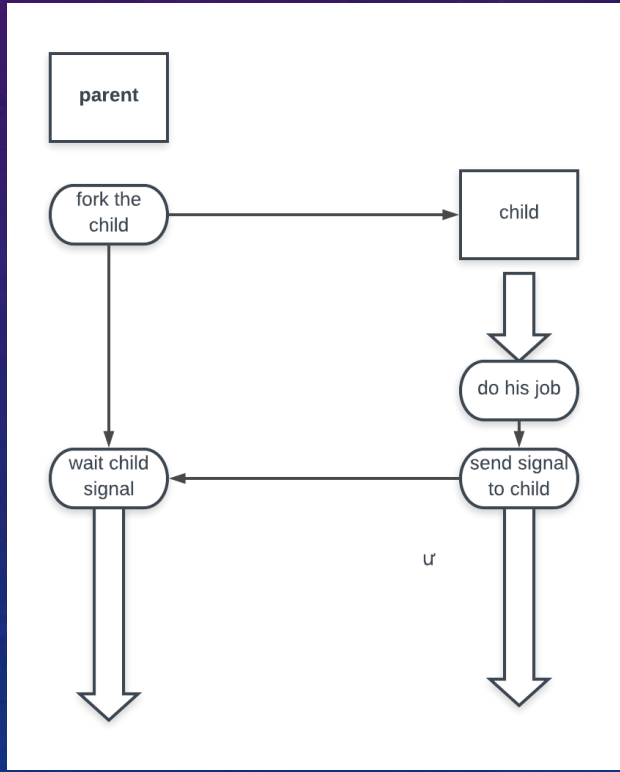


RACE CONDITIONS AFTER FORK

- Parent scheduled first in almost times but not guarantee
- /proc/sys/kernel/sched_child_runs_first work?
- Several way to guarantee the order of process of parent and child
  - Semaphore
  - File lock
  - Messaging through IPC
  - Signal

AVOID RACE CONDITION BY SYNCHRONIZING WITH SIGNAL

- Parent wait his child finish the job and process



PROCESS TERMINATION – TERMINATING A PROCESS WITH EXIT AND _EXIT

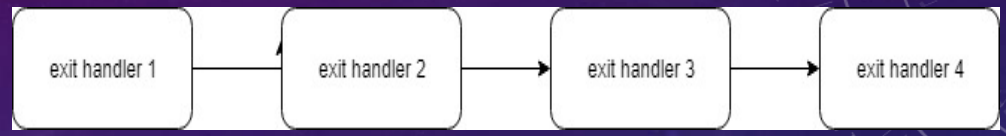
- Exit a process
  - Two way for terminating a process
    - Normal way through exit or program finish
    - Abnormal way through receiving a signal
  - Actions are performed by calling exit
    - Exit handler
    - stdio buffer stream flush
    - _exit system call involved

```
void _exit(int status);
```

- Process termination detail
  - Open file descriptor, directory streams … are closed
  - File lock held by process are released
  - Attached share memory are detached
  - Semaphores are closed
  - Message queue are closed
  - Memory lock is removed
  - Memory mapping is unmapped

EXIT HANDLER – ATEXIT/ON_EXIT

```
int atexit(void (*func)(void));
```



- Cleanup action
  - Some libraries or application need to perform cleanup action automatically when process exit
  - Linux provide a programmer supplied function that is registered in lifetime of process and automatically called during normal process terminate by call exit
  - Exit handler is not called if a program call _exit() system call directly or if process terminated abnormal by a signal

func no have argument and return value

- When a program invoke exit, these func are called in reserve order of registration
- If one of exit handler call _exit or raise a signal, process will terminate and ignore the remaining exit handler
- A child process inherit parent exit handler registration after fork
- When a process perform exec , all exit handler registration is remove

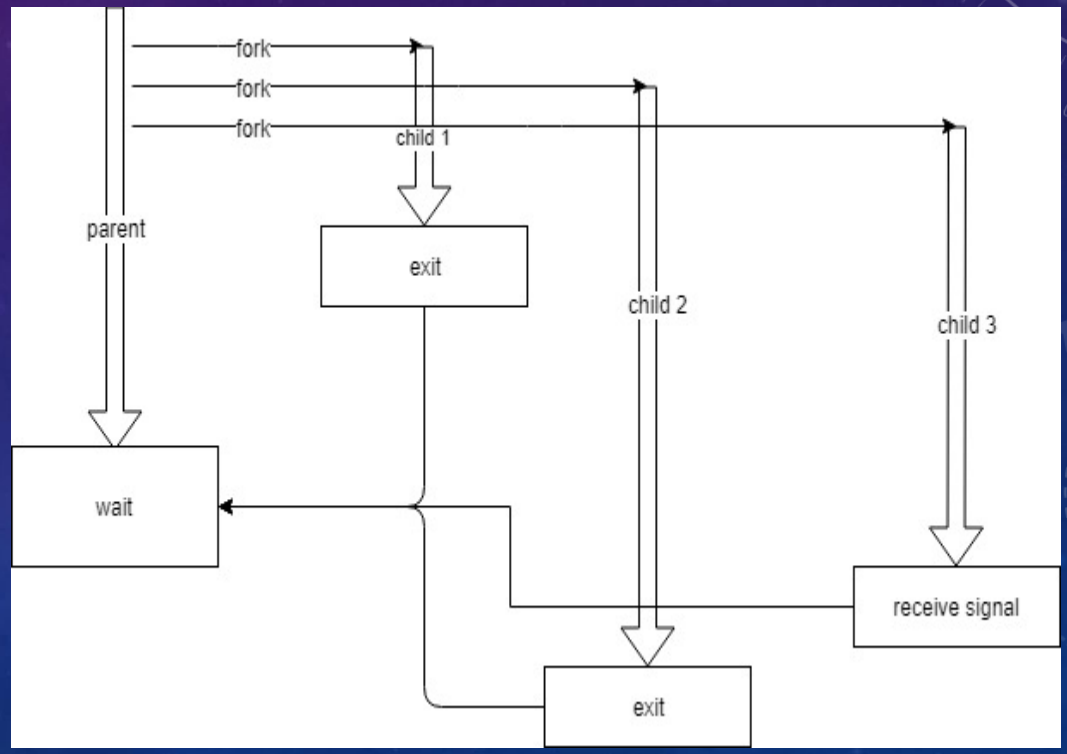
INTERACTION BETWEEN FORK, STDIO BUFFER AND EXIT

- Duplicate output when redirect output
  - Flush stdout before fork
  - Call setbuf to disable buffering on stdio stream
  - One process call exit, other call _exit system call

MONITORING CHILD PROCESS – WAIT SYSTEM CALL

- Waiting on child process
  - wait system call
    - Wait one of the children of calling process terminate and return termination status to status argument
    - If no child of calling process has yet terminated, the call blocks until one of children terminate
    - If one of children terminate already, wait return immediately
    - If status is not NULL, information about how the child terminated is returned in the status argument
    - The kernel adds the process CPU times and resource usage statistic to running totals for all children
    - Wait return the process id of children process that has terminated

```
pid_t wait(int *status);
```



WAITPID SYSTEM CALL

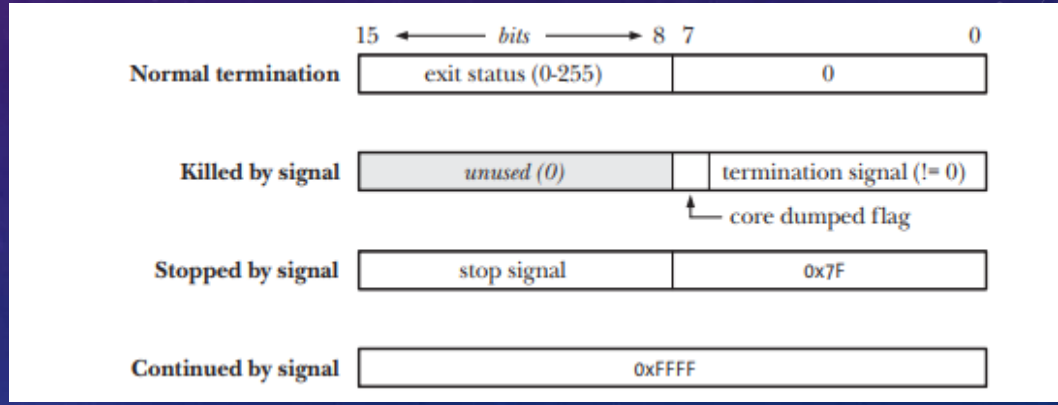
- Limitations of wait
  - Parent cannot wait a specific children
  - If no child has yet terminate, wait always block, sometime, program need parent do some other work
  - Parent don’t know when child receive a SIGSTOP

```
pid_t waitpid(pid_t pid, int *status, int options);
```

- pid argument enables the selection of the child to be waited for
  - If pid > 0, wait process has process id is pid
  - If pid = 0, wait any child
  - Status argument as wait
  - Options
    - WUNTRACED return when child receive a SIGSTOP signal
    - WCONTINUED return when a stop process is resume
    - WNOHANG return immediately if children that has process id is pid no have state change (poll)

VALUE OF STATUS

- Status value returned by wait and waitpid allow distinguish following event of the child
  - The child terminating by calling exit
  - The child terminated by the delivery of an unhandled signal
  - The child stop by signal SIGSTOP
  - The child resume by signal SIGCONT
- Macro
  - WIFEXITED(status) – normal exit
  - WIFSIGNALED(status) –unhandled signal
  - WIFSTOPPED(status) – signal stop
  - WIFCONTINUED(status) –signal continue

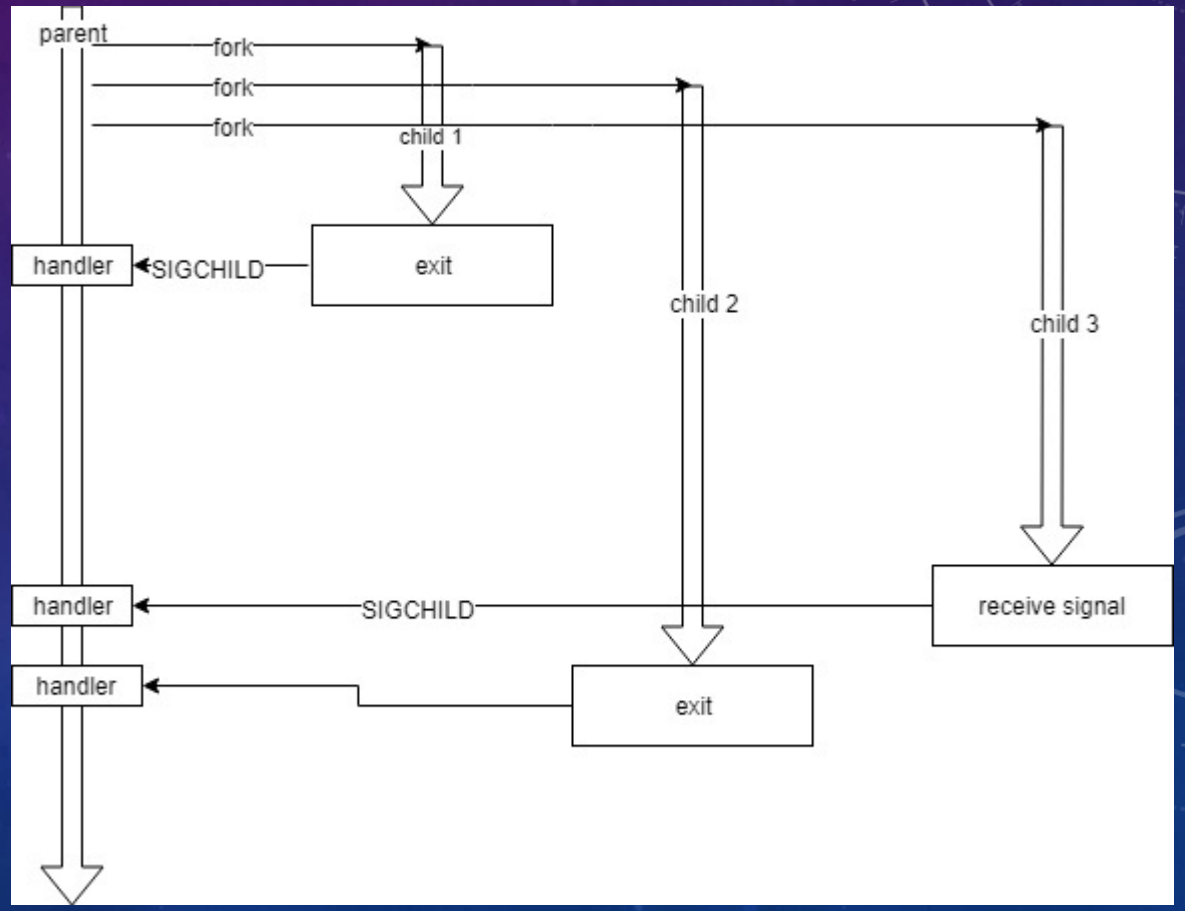


ORPHANS AND ZOMBIES

- Who become an parent of orphaned child?
  - init process –pid 1
  - getppid function
- What happen if a children terminate before its parent call wait
  - Parent be permitted call wait even when children terminated, kernel maintain terminated children at zombie state
  - Almost children resource already release back and reallocate for other process, only info in kernel process table entry that include child pid, termination status and resource usage status
  - When parent call wait, this entry removed from table
  - Otherwise, this entry exist forever until parent process terminated

SIGCHILD SIGNAL

- Child process terminated
  - An asynchronously event that parent is not predict
  - Parent can call wait and waitpid in blocking (without WNOHANG)
  - Parent can call waitpid in nonblocking (with WNOHANG)
  - Both blocking or non-blocking is inconvenient. Then child send SIGCHILD to parent each time it terminate



PROGRAM EXECUTION

- Executing a program with execve system call
  - Load a program into a process memory
  - Old program is discarded and process's stack, data, heap, text segment are replaced by new program
  - Most frequent use of execve is in the child produced by a fork

```
int execve(const char *pathname, char *const argv[], char *const envp[]);
```

- Pathname : name of binary
- argv : command line path to bin
- envp: environment list

FILE DESCRIPTOR AND EXEC

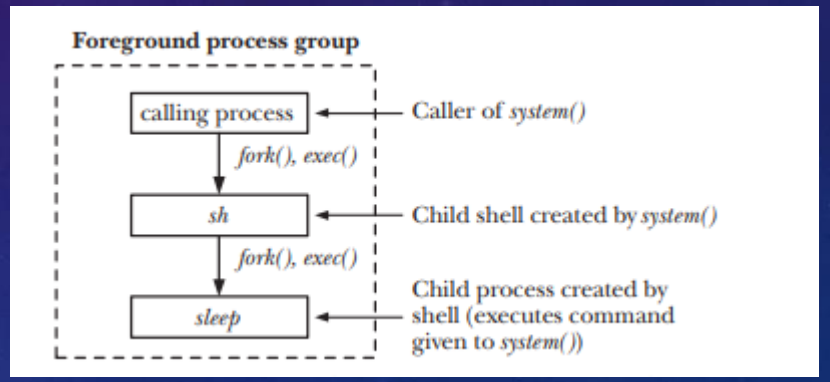
- Open file descriptor through exec
  - These files remain open across the exec and available for new program
  - Shell take advance of this feature to handle I/O redirection for program that it execute.
    - i.e. ls /tmp > dir.txt
    - Fork the child, pass the command
    - Child open dir.txt so that it has file descriptor 1
      - Close stdout, the open file descriptor of dir.txt
      - Dup2 fd of dir.txt and stdout
    - Exec the ls command, ls write to stdout which currently is dir.txt

```
thientran@ubuntu:~/Desktop/training/process$ ls /tmp/ > dir.txt
thientran@ubuntu:~/Desktop/training/process$ cat dir.txt
gedit.thientran.660062292
keyring-sgPMch
orbit-gdm
orbit-thientran
pulse-Bjz5ASSOYxiT
pulse-PKdhtXMmr18n
ssh-XBNQXF1738
virtual-thientran.wxEBI6
VMwareDnD
vmware-root
vmware-root-826451963
vmware-thientran
```

EXECUTING IN A SHELL COMMAND

```
int system(const char *command);
```

- system function
  - system(“ls /dev/ | grep rtc”) or system(“sleep”)
  - System function create a child process that involke a shell to execute command



PROCESS FUNCTION REVIEW

- GETPID/GETPPID
- FORK
- EXIT
  - ATEXIT/ON_EXIT
- WAIT/WAITPID
- EXECVE
- SYSTEM