Processes

Daniel Hagimont (INPT)

hagimont@enseeiht.fr

http://hagimont.perso.enseeiht.fr

Process

- What difference between a process and a program
  - A program is passive
    - Stored on disk as an executable file
      - e.g. /bin/ls

```
hagimont@hagimont-pc:~$ ls -la /bin/ls
-rwxr-xr-x 1 root root 142144 sept.  5  2019 /bin/ls
```

  - A process is active
    - Execute on a processor

```
hagimont@hagimont-pc:~$ which ls
/usr/bin/ls
hagimont@hagimont-pc:~$ ls
bigdata2  Documents          install  Public  Téléchargements
Bureau    eclipse-workspace  Modèles  shared  tmp
divers    Images             Musique  snap    Vidéos
hagimont@hagimont-pc:~$
```

Process

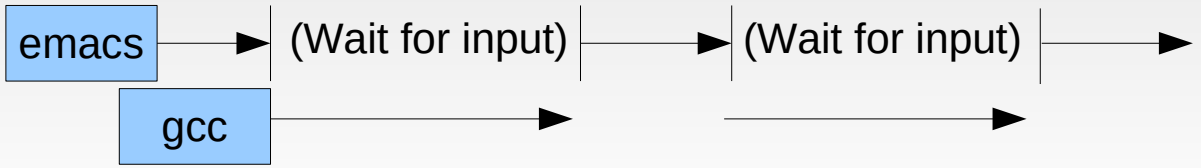
- A process is an instance of a running program
  - Eg : gcc, sh, firefox ...
  - Created by the system or by an application
  - Created by a parent process
  - Uniquely idendified (PID)
- Correspond to two units :
  - Execution unit
    - Sequential control flow (execute a flow of instructions)
  - Addressing unit
    - Each process has its own address space
    - Isolation

Process

- Processes can run on one or multiple processors
  - Several processes on one CPU: concurrency
  - Several processes on several CPU: parallelism

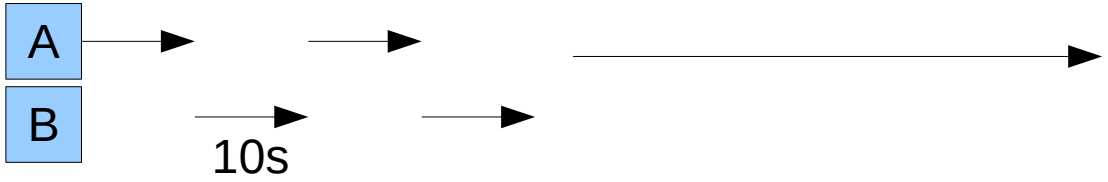
Concurent processes

- Multiple processes can increase CPU utilization
  - Overlap one process's computation while another waits



- Multiple processes can reduce latency
  - Running A then B requires 100 secs for B to complete

  - Running A and B concurently (with preemption) improves the average response time

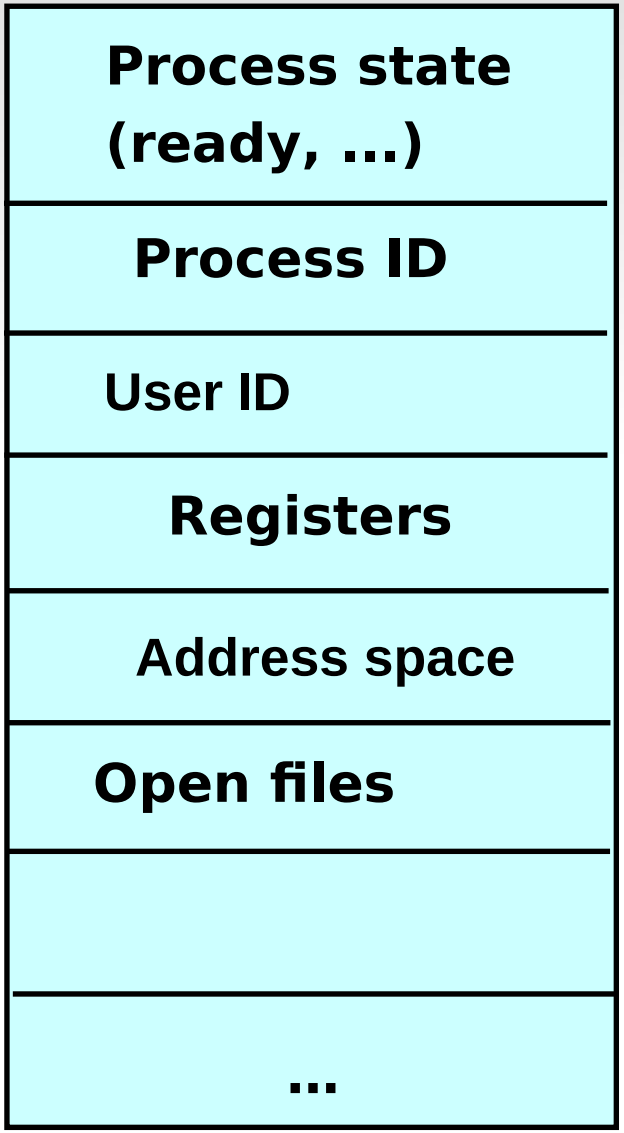


Execution context

- A process is characterized by its context
- Process' current state
  - Memory image
    - Code of the running program
    - Static and dynamic data
  - Register's state
    - Program counter (PC), Stack pointer (SP) ...
  - List of open files
  - Environment Variables
  - ...
- To be saved when the process is switched off
- To be restored when the process is switched on

Process Control Structure

- Hold a process execution context
- PCB (Process Control Block):
  - Data required by the OS to manage process
- Process tables:
  - PCB [MAX-PROCESSES]



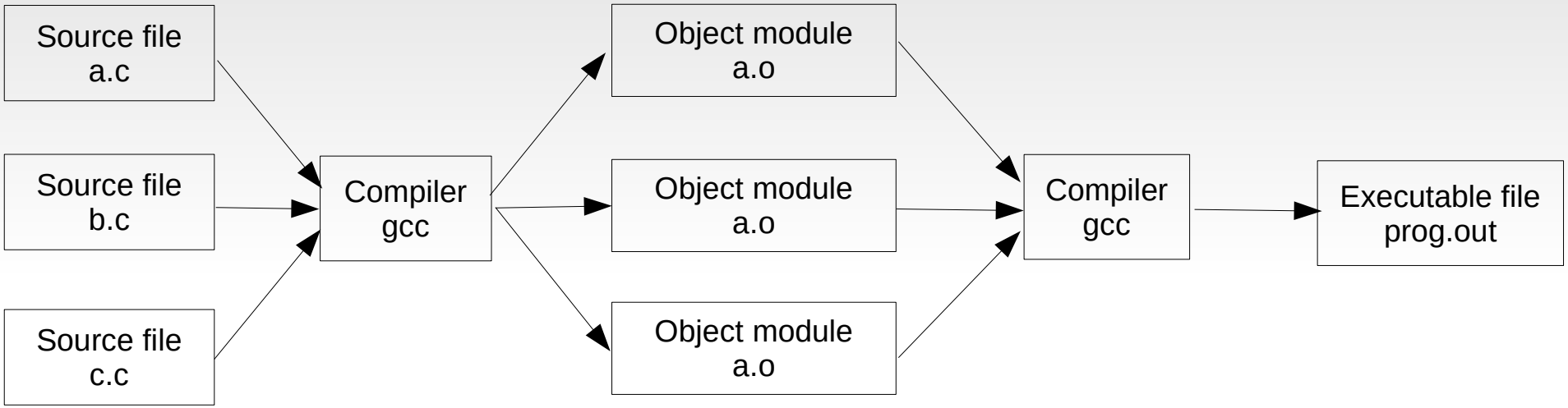
Running mode

- User mode
  - Access restricted to process own adress space
  - Limited instruction set
- Supervisor mode
  - Full memory access
  - Full access to the instruction set
- Interrupt, trap
  - Asynchronous event
  - Illegal instruction
  - System call request

Process memory layout

- Process execution state
  - Processor state
  - File descriptors
  - Memory allocation

Compiling

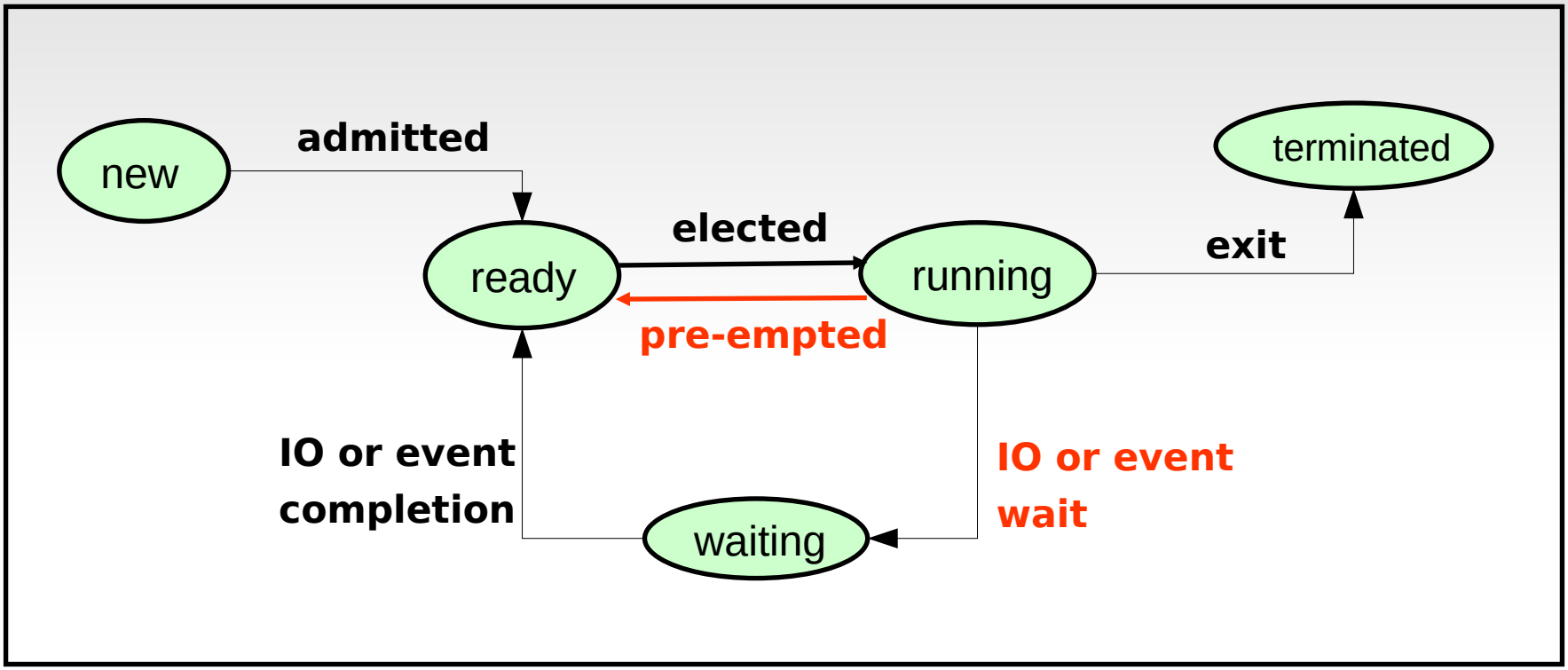


- Source files are compiled to object modules
- Object modules are linked into a single executable file
  - Example: gcc <source> [-o output]

Execute a process

- Create a new process (paused)
- Load executable file into process memory
- Load dynamic libraries
- Relocated APIs
- Set the program counter and stack pointer
- Resume the process

Process Lifecycle



- Which process should the kernel run ?
  - If 0 runnable, run a watchdog, if 1 runnable, run it
  - If n runnable, make scheduling decision

Exercise

- List all the running processes (with ps – see man)
- Start a new process (e.g. gnome-calculator)
- Find the id of this new process
- Show its status (see content of /proc/<id>/status)
- Pause it (kill with signal STOP)
- Resume it (kill with signal CONT)
- Terminate it (kill with signal KILL)
- Look at the tree of processes (pstree -a)

Process SVC overview

- int fork ();
  - Creates a new process that is an exact copy of the current one
  - Returns the process ID of the new process in the “parent”
  - Returns 0 in “child”
- int waitpid (int pid, ...);
  - pid – the process to wait for, or -1 for any
  - Returns pid of resuming process or -1 on error
- Hierarchy of processes
  - run the pstree -p command

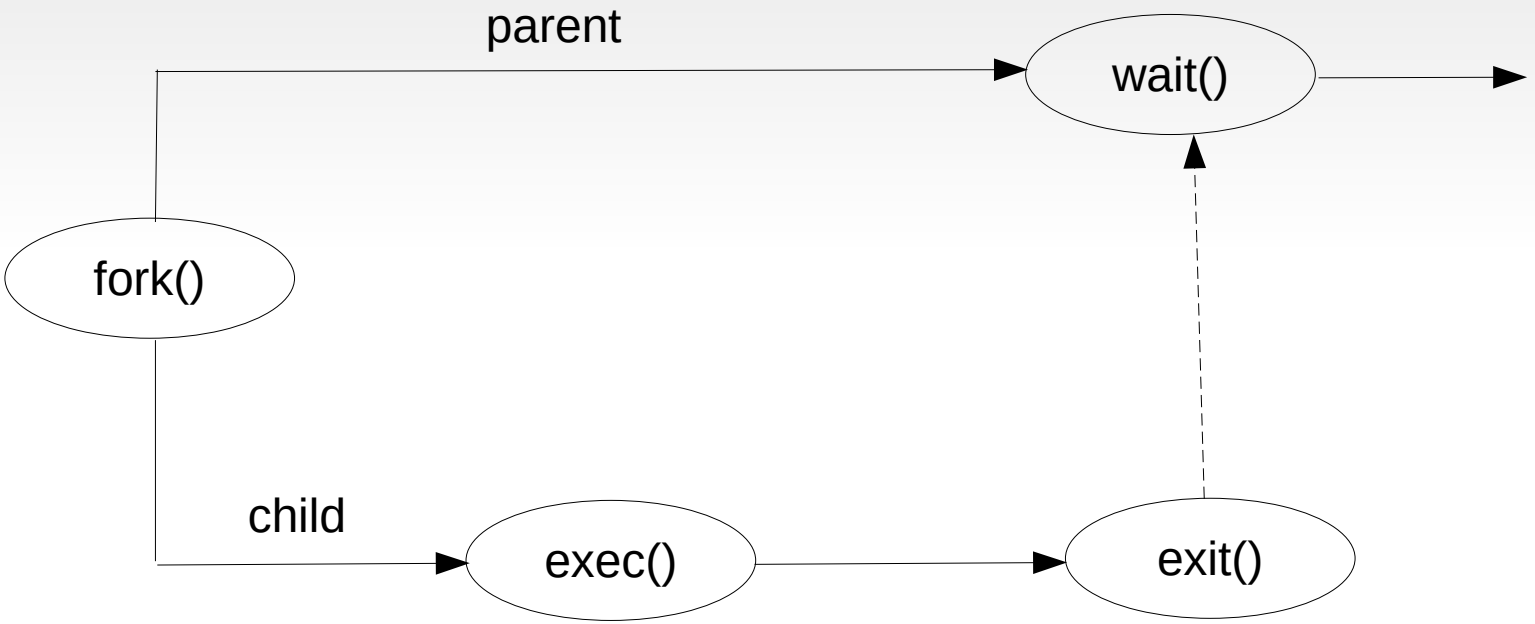
Process SVC overview

- void exit (int status);
  - Current process stops
  - status: returned to waitpid (shifted)
  - By convention, status of 0 is success
- int kill (int pid, int sig);
  - Sends signal sig to process pid
  - SIGTERM most common value, kills process by default (but application can catch it for “cleanup”)
  - SIGKILL stronger, kills process always
- When a parent process terminates before its child, 2 options:
  - Cascading termination (VMS)
  - Re-parent the orphan (UNIX)

Process SVC overview

- int execve (const char *prog, const char **argv, char **envp;)
  - prog – full pathname of program to run
  - argv – argument vector that gets passed to main
  - envp – environment variables, e.g., PATH, HOME
- Many other versions
  - int execl(const char *path, const char *arg, ... /* (char *) NULL */);
  - int execlp(const char *file, const char *arg, ... /* (char *) NULL */);
  - int execle(const char *path, const char *arg, ... /*, (char *) NULL, char * const envp[] */);
  - int execv(const char *path, char *const argv[]);
  - int execvp(const char *file, char *const argv[]);
  - int execvpe(const char *file, char *const argv[], char *const envp[]);

Process creation



Fork and Exec

- The fork system call creates a copy of the PCB
  - Opened files and memory mapped files are thus similar
  - Open files are thus opened by both father and child. They should both close the files.
  - The pages of many read only memory segments are shared (text, r/o data)
  - Many others are lazily copied (copy on write)
- The exec system call replaces the address space, the registers, the program counter by those of the program to exec.
  - But opened files are inherited

Why fork

- Most calls to fork followed by execvp
- Real win is simplicity of interface
  - Tons of things you might want to do to child
  - Fork requires no arguments at all
  - Without fork, require tons of different options
  - Example: Windows CreateProcess system call

```
Bool CreateProcess(
    LPCTSTR lpApplicationName, //pointer to a name to executable module
    LPTSTR lpCommandLine, // pointer to a command line string
    LPSECURITYATTRIBUTES lpProcessAttributes, //process security attr
    LPSECURITYATTRIBUTES lpThreadAttributes, // thread security attr
    BOOL bInheritHandles, //creation flag
    DWORD dwCreationFlags, // creation flags
    LPVOID lpEnvironnement, // pointer to  new environment block
    LPCTSTR lpCurrentDirectory, // pointer to crrent directory name
    LPSTARTUPINFO lpStartupInfo, //pointer to STARTUPINFO
    LPPROCESSINFORMATION lpProcessInformaton // pointer to PROCESSINFORMATION );
```

Fork example

- Process creation
  - Done by cloning an existing process
    - Duplicate the process
  - Fork() system call
    - Return 0 to the child process
    - Return the child's pid to the father
    - Return -1 if error

```
#include <unistd.h>

pid_t fork(void);
```

```
r = fork();
if (r==-1) ...    /* error */
else if (r==0) ... /* child's code */
else ...           /* father's code */
```

Exercise

- How many processes are created ?

```
fork();
fork();
fork();
```

```
for (i=0; i<3;i++){
   fork();
}
```

- What are the possible different traces ?

```
int i = 0;
switch (j=fork()) {
        case -1 : perror("fork"); break;
        case  0 : i++; printf("child :%d",i); break;
        default : printf("father :%d",i);

}
```

Exec example

- Reminder: main function definition
  - int main(int argc, char *argv[]);
- Execvp call
  - Replaces the process's memory image
  - int execvp(const char *file, const char *argv[]);
    - file : file name to load
    - argv : process parameters
  - execvp calls main(argc, argv) in the process to launch

Exercice

```
char  * argv[3];
argv[0] = “ls “;
argv[1] = “-al “;
argv[2] = NULL;
execvp(“ls“, argv);
```

or

```
execlp(“ls“, “ls”, “-al”, NULL);
```

Father/child synchronization

- The father process waits for the completion of one of its child
  - pid_t wait(int *status):
    - The father waits for the completion of one of its child
      - pid_t : dead child's pid or -1 if no child
      - status : information on the child's death
  - pid_t waitpid(pid_t pid, int *status, int option);
    - Wait for a specific child's death
    - Option : non blocking … see man

Wait example

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(){
  int spid, status;
  switch(spid = fork()){
        case -1 : perror("..."); exit(-1);
        case 0  :  // child's code
                break;
        default : // the father wait for this child's terminaison
                if (waitpid(spid,&status,0)==-1) {perror("...");exit(-1);}
                  ...
  }
}
```

Exercise (minishell)

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>

pid_t pid;
char *av[2];
char cmd[20];

void doexec() {
if (execvp(av[0],av)==-1)
            perror ("execvp failed");
    exit(0);
}
```

```
int main() {
for (;;) {
    printf(">");
    scanf("%s",cmd);
    av[0] = cmd;
    av[1] = NULL;
    switch (pid = fork()) {
        case -1: perror("fork"); break;
        case 0:
            doexec();
        default:
            if (waitpid(pid, NULL, 0) == -1)
                    perror ("waitpid failed");
  }
}
}
```

Resources you can read

- Operating System Concepts, 10th Edition, Abraham Silberschatz, Peter B. Galvin, Greg Gagne
  - http://os-book.com/
  - Chapters 3
- Modern Operating Systems, Andrew Tanenbaum
  - http://www.cs.vu.nl/~ast/books/mos2/
  - Chapter 2 (2.1)