Introduction to Operating Systems

Daniel Hagimont

https://www.google.fr/search?q=daniel+hagimont+home+page

Outline

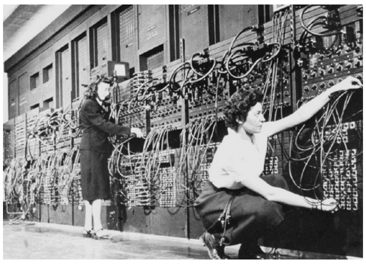
- Introduction
- Files
- Processes
- Inter-process communication
- Threads
- Scheduling
- Synchronization
- Memory

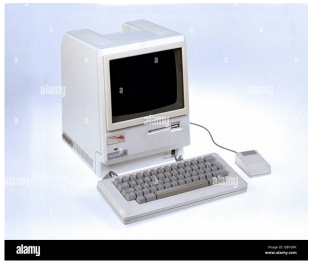
Organization

- Moodle
  - https://sd-160040.dedibox.fr/hagimoodle
  - usth2024/Usth2024!
  - Access to docs
- Evaluation
  - Quiz after each lab
  - A final quiz exam

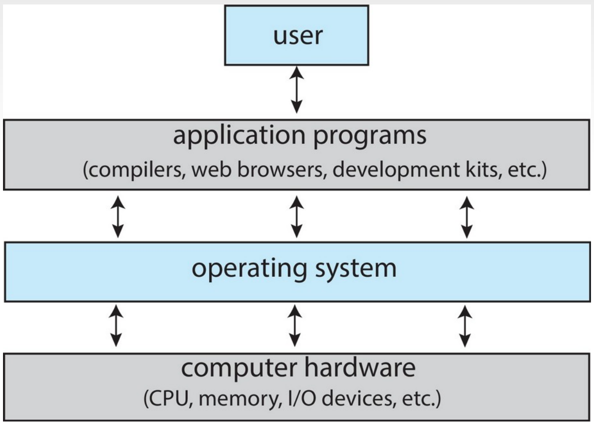
What is an Operating System?

- A program that acts as an intermediary between a user of a computer and the computer hardware
- Operating system goals:
  - Execute user programs and make solving user problems easier (programming)
  - Make the computer system convenient to use
  - Use the computer hardware in an efficient manner





Abstract view of computer system

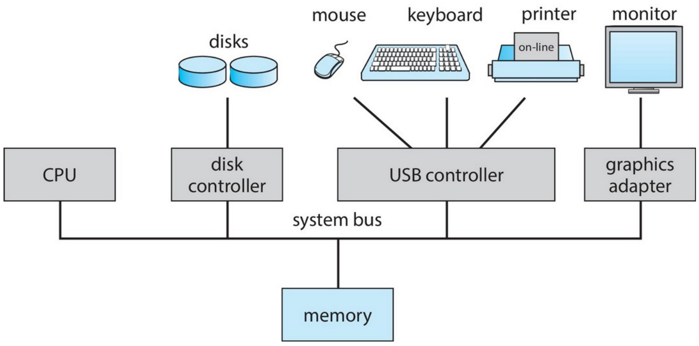


Computer system structure

- Computer system can be divided into four components:
  - Hardware – provides basic computing resources
    - CPU, memory, I/O devices
  - Operating system
    - Controls and coordinates use of hardware among various applications and users
    - Makes application programming easier
  - Application programs
    - Word processors, compilers, web browsers, database systems, video games
  - Users
    - People, machines, other computers

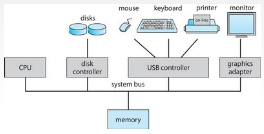
Hardware organization

- I/O devices and CPUs can execute in parallel
- Each device controller is in charge of a particular device type
- CPUs and device controllers connect through common bus providing access to shared memory



Computer system operation

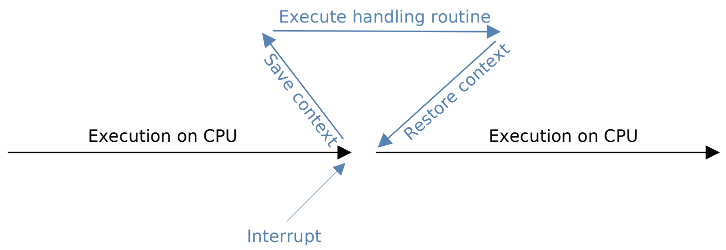
- Each device controller has
  - Local buffers (for sending/receiving data)
    - I/O is from the device to local buffer of controller
  - Control registers (for triggering actions)
  - They are generally mapped in memory



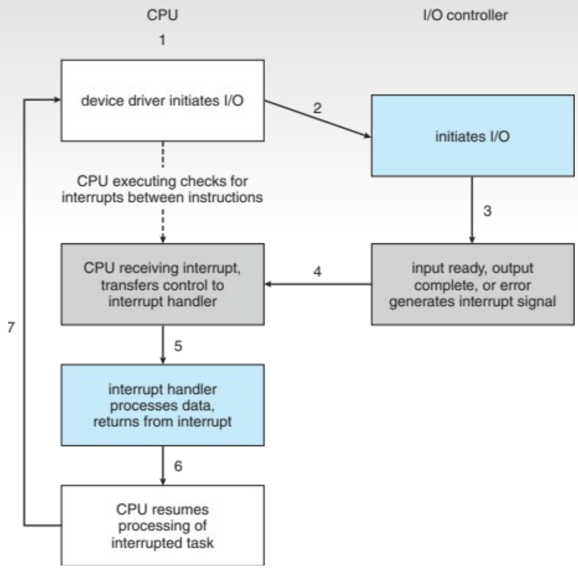
- Performing an I/O (e.g. write to disk)
  - Copy data from memory to local buffer (mapped in memory)
  - Write in the control register (mapped in memory) to trigger the I/O
- Device controller informs CPU that it has finished its operation by causing an interrupt
- An operating system device driver to manage it
  - Provides an API
  - Performs I/Os
  - Handles interrupts

Functioning of interrupts

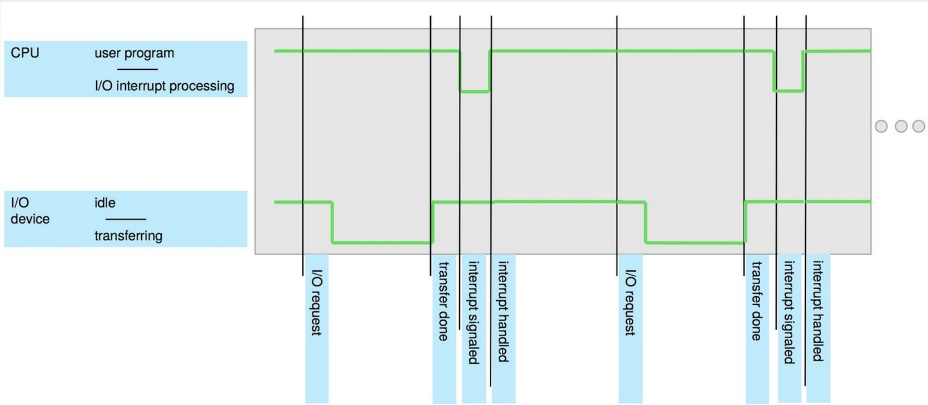
- An interrupt stops the execution on one CPU
  - The execution context of the CPU (mainly registers, including the address of the next instruction) is saved
- Invokes the handling routine associated with the interrupt
  - Different interrupts (identified by a number, so an interrupt vector provides all routines)
- When the handling routine terminates, the execution context is restored



Performing I/O operations



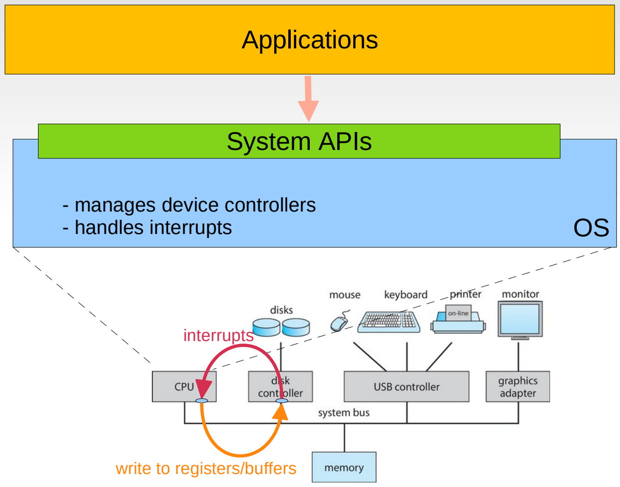
I/O operations timeline



Different types of I/O interfaces

- Non-blocking
  - After I/O starts, control returns to user program without waiting for I/O completion
- Blocking
  - After I/O starts, control returns to user program only upon I/O completion
- Process blocking is implemented by the OS

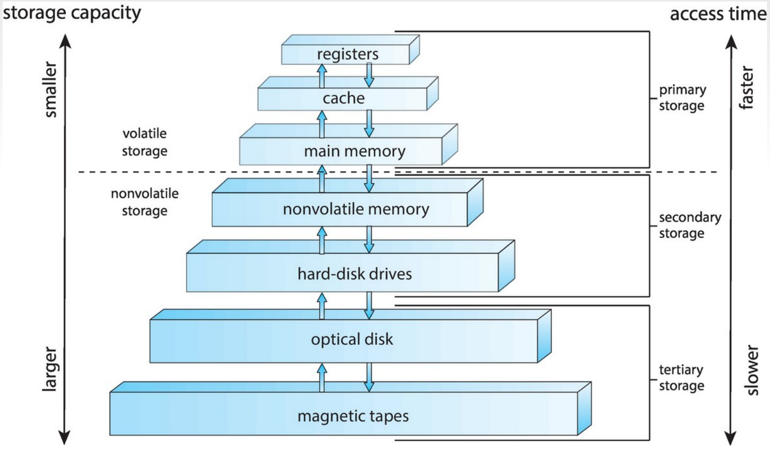
Overview


Applications
System APIs
- manages device controllers
- handles interrupts
OS
interrupts
write to registers/buffers
disks
mouse
keyboard
printer
monitor
CPU
disk controller
USB controller
graphics adapter
system bus
memory

Memory

- Main memory – storage media that the CPU can access directly
  - Typically volatile
  - Typically random-access memory in the form of Dynamic Random-access Memory (DRAM)
- Secondary storage – extension of main memory that provides large non-volatile storage capacity
  - Managed in external devices
  - Typically HDD, SSD ...

Memory hierarchy



Memory hierarchy

Level	1	2	3	4	5
Name	registers	cache	main memory	solid-state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS SRAM	flash memory	magnetic disk
Access time (ns)	0.25-0.5	0.5-25	80-250	25,000-50,000	5,000,000
Bandwidth (MB/sec)	20,000-100,000	5,000-10,000	1,000-5,000	500	20-150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape

Memory management

- Registers are managed by the programmer or compilers
- Processor caches are managed by the hardware
- Main memory and secondary storage are managed by the operating system

Caching principle

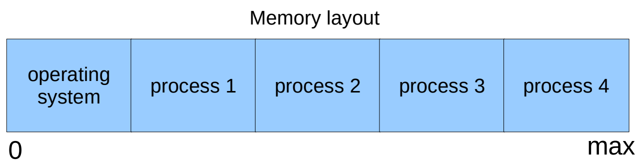
- Important principle, performed at many levels in a computer (in hardware, operating system, software)
- Information in use copied from slower to faster storage temporarily
- Faster storage (cache) checked first to determine if information is there
  - If it is, information used directly from the cache (fast)
  - If not, data copied to cache and used there
- Cache smaller than storage being cached
  - Cache management important design problem
  - Cache size and replacement policy

Process management

- A process is a program in execution. It is a unit of work within the system. Program is a passive entity; process is an active entity.
- Process needs resources to accomplish its task
  - CPU, memory, ...
- Process termination requires reclaim of any reusable resources
- Each process has one program counter specifying location of next instruction to execute
  - Process executes instructions sequentially, one at a time, may branch to any location, until completion
- Typically system has many processes, some user, some operating system running concurrently on one or more CPUs
  - The process scheduler is the OS component which decides which process executes on which CPU (at any time)
  - e.g., on my laptop, 350 running processes / 12 cores (but most processes are waiting)

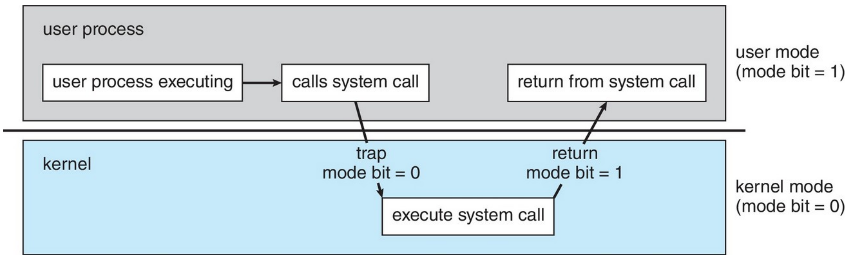
Process management

- The operating system is responsible for the following activities in connection with process management
  - Creating and deleting both user and system processes
  - Suspending and resuming processes
  - Providing mechanisms for process synchronization
  - Providing mechanisms for process communication
  - Managing the allocation of memory to processes



OS protection

- Dual-mode (user mode and kernel mode) operation allows OS to protect itself and other system components
    - Mode bit provided by hardware to distinguish when running in an application or in the system
    - System call changes mode to kernel, return from call resets it to user
- System memory only accessible in kernel mode
- Some instructions designated as privileged, only executable in kernel mode



Protection and security

- Protection – any mechanism for controlling access of processes or users to resources defined by the OS
- Security – defense of the system against internal and external attacks
  - Huge range, including denial-of-service, worms, viruses, identity theft, theft of service
- Access control
  - User identities (userid) include name and associated number, one per user
    - userid then associated with all files, processes of that user to determine access control
  - Group identifier (groupid) allows set of users to be defined and to define access control for all users in the group
  - Privilege escalation allows user to augment access rights to perform a specific operation

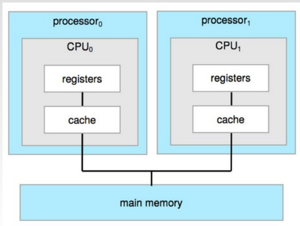
File-system management

- OS implements basic functions to access storage devices (typically disks)
- At an higher level: File-System management
  - Files have a name
  - Files usually organized into directories
  - Access control on most systems to determine who can access what
  - OS activities include
    - Creating and deleting files and directories
    - APIs to manipulate files and directories (in programs)
    - Applications to manipulate files and directories (for users)

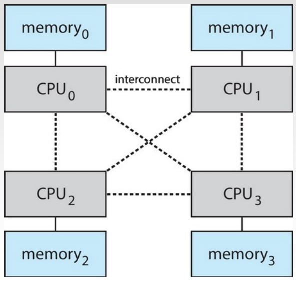
Multiprocessor architectures

- Most systems use a single general-purpose processor
  - This processor can be multi-core
  - Most systems have special-purpose processors as well
- Multiprocessors systems growing in use and importance
  - Also known as parallel systems, tightly-coupled systems
  - Advantages include:
    - Scalability
    - Increased reliability – fault tolerance
  - Two types:
    - Asymmetric Multiprocessing – each processor is assigned a specific task
    - Symmetric Multiprocessing – each processor performs all tasks

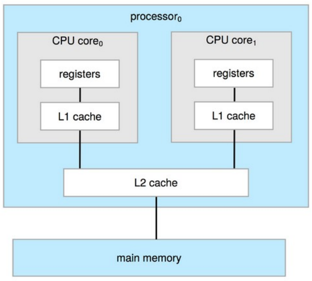
Multiprocessor architectures



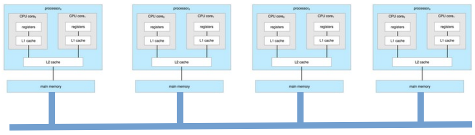
Symetric Multi-Processor



Non Uniform Memory
Access multi-processor



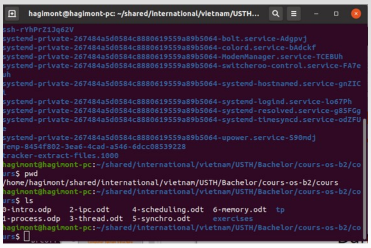
Multi-core processor



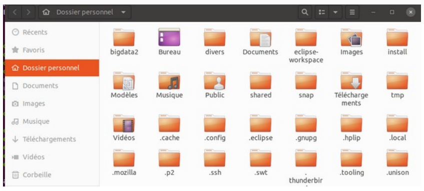
Clustered multi-processor

System interface

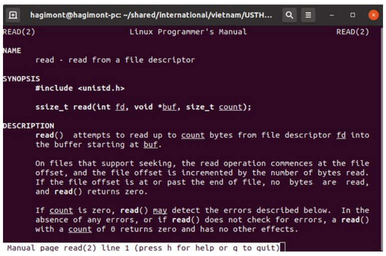
- Command Line Interpreter (shell)



- Graphical User Interface
  (window manager)



- System APIs



Basic shell commands

Commands	Description
man [command]	Display user **man**ual for the specified command.
cd /directorypath	**C**hange **d**irectory.
ls [opts]	**L**i**s**t directory contents.
cat [files]	Display file's contents after cont**cat**enation.
mkdir [opts] dir	**M**a**k**e a new **dir**ectory.
cp [opts] src dest	**C**o**p**y files and directories.
mv [opts] src dest	Rename or **m**o**v**e file(s) or directories.
rm [opts] dir	**R**e**m**ove files and/or directories.
chmod [opts] mode file	**Ch**ange a file's **mod**es (permissions).
chown [opts] file	**Ch**ange **own**er of a file.
df [opts]	Display **d**isk's **f**ree and used space.
du [opts]	Show **d**isk **u**sage that each file takes up.
find [pathname] [expr]	**Find** for files matching a provided pattern.
grep [opts] pattern [file]	Search files or output for a particular pattern.
nano [file]	**N**ano's **ano**ther editor

Basic shell commands

Commands	Description
`kill [opts] pid`	**Kill** a process.
`less [opts] [file]`	View the contents of a file one page at a time.
`ln [opts] src [dest]`	Create a shortcut. (**lin**ks)
`passwd`	Change your **password**
`ps [opts]`	List **p**rocess **s**tatus.
`pwd`	**P**rint **w**orking **d**irectory
`ssh [opts] user@host [cmd]`	Remotely log in to another machine with **s**ecured **sh**ell
`su [opts] [user]`	**S**witch to another **u**ser account.
`head [opts] [file]`	Display the first n **head**ing lines of a file.
`tail [opts] [file]`	Display the last n **tail**ing lines of a file.
`tar [opts] file`	Store/Extract (and compress/decompress) **t**ape **ar**chives
`top`	Displays resources being used on your system.
`touch file`	Create an empty file with the specified name.
`who [opts]`	Display **who** is logged on.
`wget url`	Non-interactive network downloader

Exercise (cmds)

- find
  - Find all the core file in a directory (recursively)
  - Remove them
- chmod
  - Make a binary executable or not
  - Make a directory accessible (with cd) or not
- grep
  - Find the occurences of a word in a file
- ps/kill
  - Create a process (gedit), stop it, resume it, and kill it
- ln
  - Create a link to a file

Exercise (cmds)

- apt-get install ….
  - Install a ssh server
- ssh
  - Connect to the ssh server you installed
- wget
  - Dowload a software from a web site
- tar
  - Uncompress an archive
  - Create an archive
- pipe
  - Count the number of firefox processes running

- Mesure the time it takes to go in a directory with a shell and with a window manager

File System

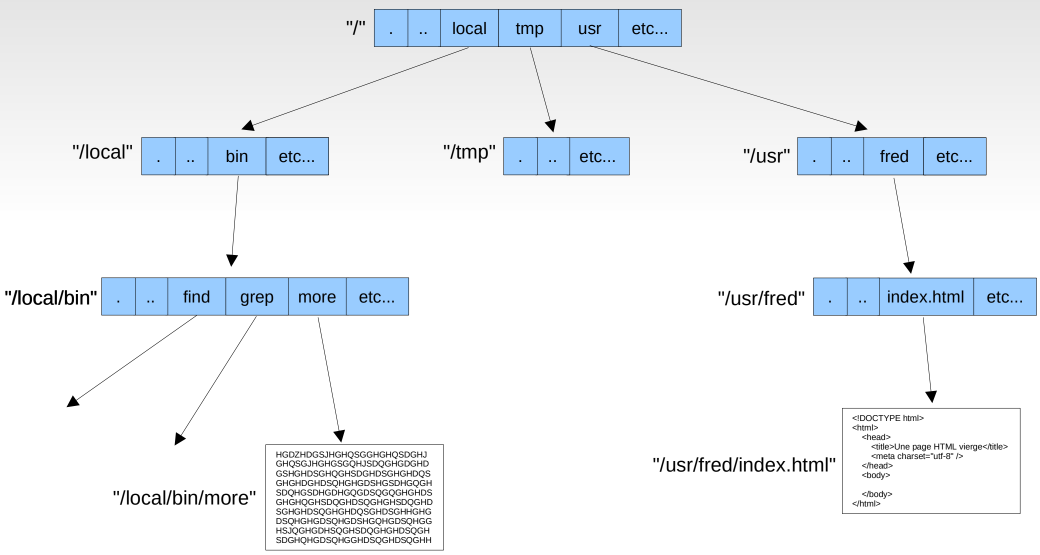
Daniel Hagimont

https://www.google.fr/search?q=daniel+hagimont+home+page

File concept

- Contiguous address space in storage
- Type of files
  - Data: Numeric, Character, Binary
  - Program
- Contents defined by file’s creator
  - Many types: text, source (for different applications), executable
  - Content identified by the file name extension (e.g. .doc)
- File can be handled
  - By the end user using a shell or graphical interface
  - By a programmer through an API

Tree structured directory



Naming

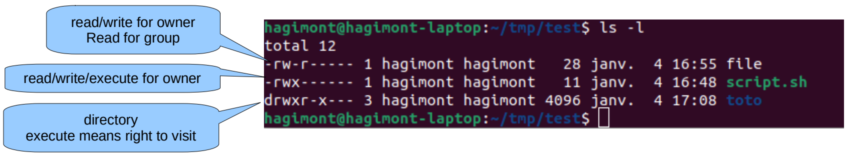
- Notion of current directory
  - Each process has a "current directory" variable (can be modified)
- Root is the top of the tree ("/")
- A name can be absolute
  - The path from "/"
  - e.g., for the "more" file, /local/bin/more
- A name can be relative
  - to the current directory
  - e.g., if current directory is "/local", for the "more" file, bin/more
- "." is the current directory, ".." is the parent directory

Access control

- Each process execute on the account of a user
  - This user is identified by a userid
  - This user may belong to several groups of user

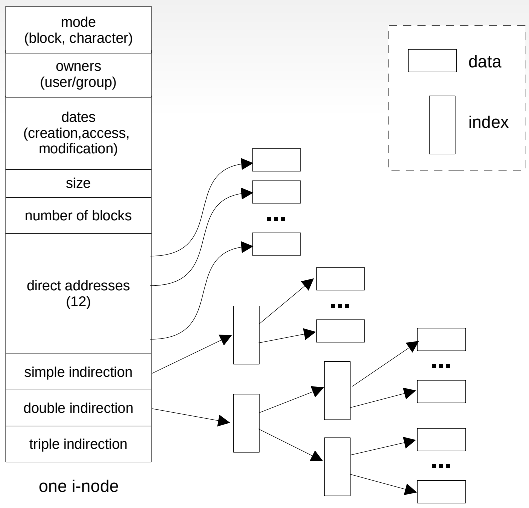
```
hagimont@hagimont-laptop:~$ id
uid=1000(hagimont) gid=1000(hagimont) groupes=1000(hagimont),4(adm),24(cdrom),27
(sudo),30(dip),46(plugdev),109(kvm),122(lpadmin),135(lxd),136(sambashare),139(li
bvirt),999(docker)
```

- Each file belongs to a user (owner) and a group
- Access modes on files : read, write, execute
- Definition of access rights for 3 categories of users
  - Owner of the file : rwx
  - Users in the group of the file : rwx
  - Other users : rwx



File management

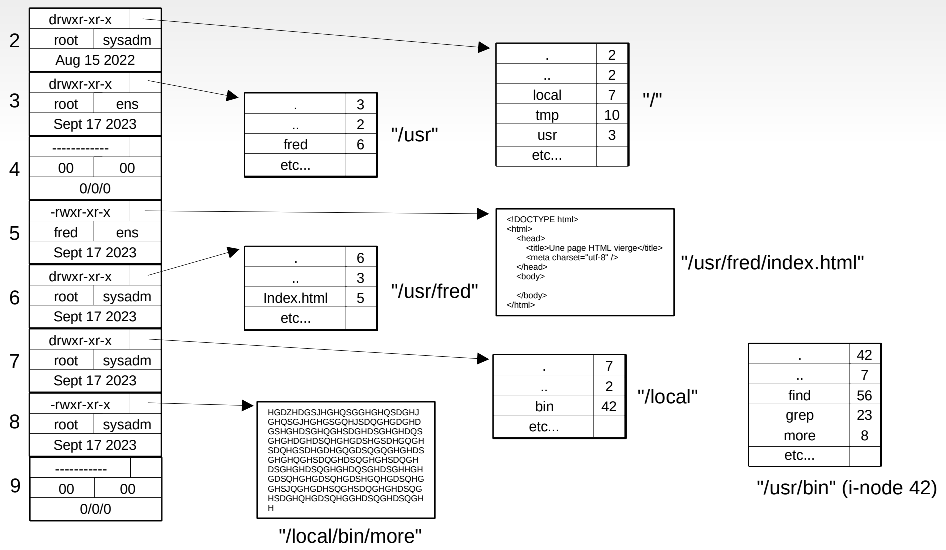
- On disk and in memory
  - Each file is represented by a i-node



one i-node

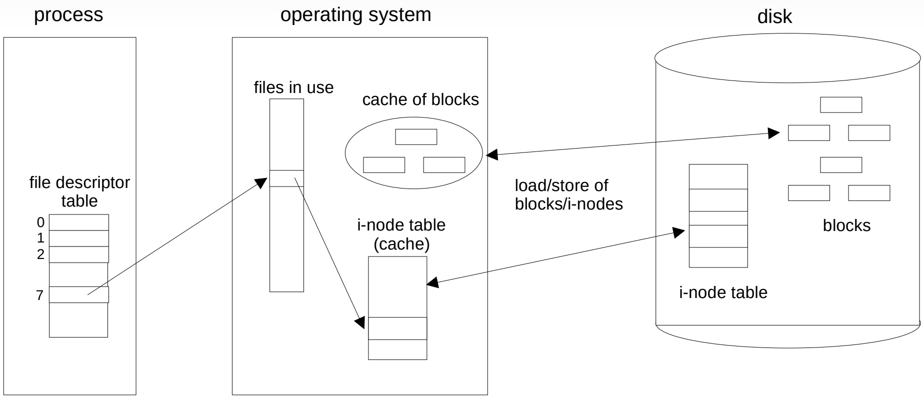
Implementation of naming

- Independant from i-node
- A directory is a file including <filename,i-node> pairs



Memory management

- Each process has a table of opened (in use) files
  - An opened file is identified by its index in this table
  - Indexes 0/1/2 are reserved for input (keyboard), screen (terminal) and error (another terminal)
- OS provides caching of blocks and i-nodes



Manipulating files from a shell

Commands	Description
`man [command]`	Display user **man**ual for the specified command.
`cd /directorypath`	**C**hange **d**irectory.
`ls [opts]`	**L**i**s**t directory contents.
`cat [files]`	Display file's contents after con**cat**enation.
`mkdir [opts] dir`	**M**a**k**e a new **dir**ectory.
`cp [opts] src dest`	**C**o**p**y files and directories.
`mv [opts] src dest`	Rename or **m**o**v**e file(s) or directories.
`rm [opts] dir`	**R**e**m**ove files and/or directories.
`chmod [opts] mode file`	**Ch**ange a file's **mod**es (permissions).
`chown [opts] file`	**Ch**ange **own**er of a file.
`df [opts]`	Display **d**isk's **f**ree and used space.
`du [opts]`	Show **d**isk **u**sage that each file takes up.
`find [pathname] [expr]`	**Find** for files matching a provided pattern.
`grep [opts] pattern [file]`	Search files or output for a particular pattern.
`nano [file]`	**N**ano's **ano**ther editor

Manipulating files from a program

- Directories
  - int mkdir(const char *pathname, mode_t mode);
  - int rmdir(const char *pathname);
  - int chdir(const char *path);
- Files
  - int open(const char *pathname, int flags, mode_t mode);
  - off_t lseek(int fd, off_t offset, int whence);
  - ssize_t read(int fd, void *buf, size_t count);
  - ssize_t write(int fd, const void *buf, size_t count);

Opening a file

```
int open(const char *pathname, int flags, mode_t mode);
```

file descriptor

file name

access rights if creation
e.g. : rwx rw- r-- = 111 110 100 = 764

O_RDONLY (read only)
O_WRONLY (write only)
O_RDWR (read and write)
O_APPEND (write only at end of file)
O_CREAT (create file with access rights defined in mode)
O_EXCL (return error if file exists)
O_TRUNC (empty the file if exists)

- Example:
  - fd = open ("/home/ubuntu/file", O_RDWR|O_CREAT, 700);
  - Open file /home/ubuntu/file in read/write mode. If file doesn't exist, create it with access rights rwx------

Moving the cursor

off_t lseek(int fd, off_t offset, int whence);

new position

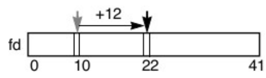
file descriptor

requested move

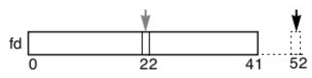
SEEK_SET (from the beginning of the file)
SEEK_CUR (from the current position)
SEEK_END (from the end of the file)

- Examples:

lseek(fd, 12, SEEK_CUR);
the current position is right shifted of 12 bytes from the current position



lseek(fd, 52, SEEK_SET);
the current position is set to 52



Reading a file

```
ssize_t read(int fd, void *buf, size_t count);
```

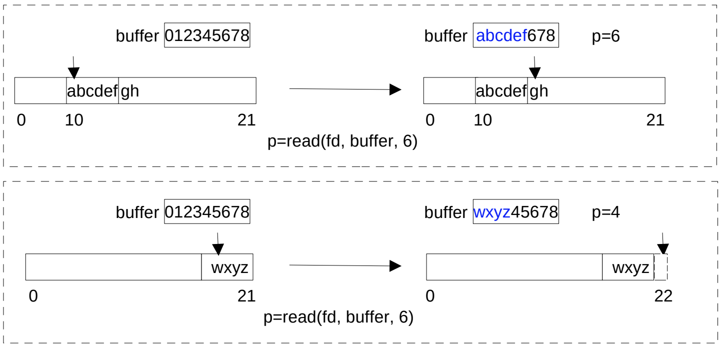
number of bytes effectively read

file descriptor

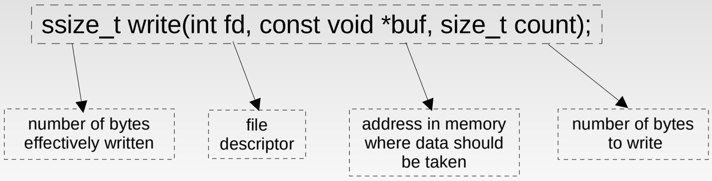
address in memory where data should be placed

number of bytes to read

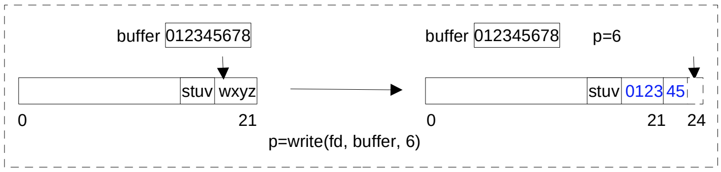
- Examples:



Writing a file



- Example:



Resources you can read

- Operating System Concepts, 10th Edition, Abraham Silberschatz, Peter B. Galvin, Greg Gagne
    - http://os-book.com/
    - Chapters 13 - 14
- Modern Operating Systems, Andrew Tanenbaum
    - http://www.cs.vu.nl/~ast/books/mos2/
    - Chapter 6

Processes

Daniel Hagimont

https://www.google.fr/search?q=daniel+hagimont+home+page

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 (memory)
    - Isolation

Process

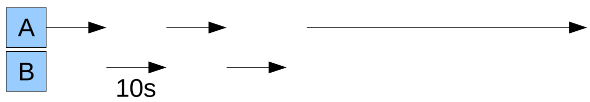
- Processes can run on one or multiple processors
  - Several processes on one CPU: concurrency
  - Several processes on several CPU: parallelism
  - e.g., on my laptop, 350 running processes / 12 cores (but most processes are waiting)

Concurrent 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

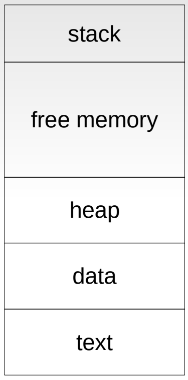
- Hold a process execution context
- PCB (Process Control Block):
  - Data required by the OS to manage process
- Process tables:
  - PCB [MAX-PROCESSES]

Process state (ready, ...)
Process ID
User ID
Registers
Address space
Open files
...

Running mode

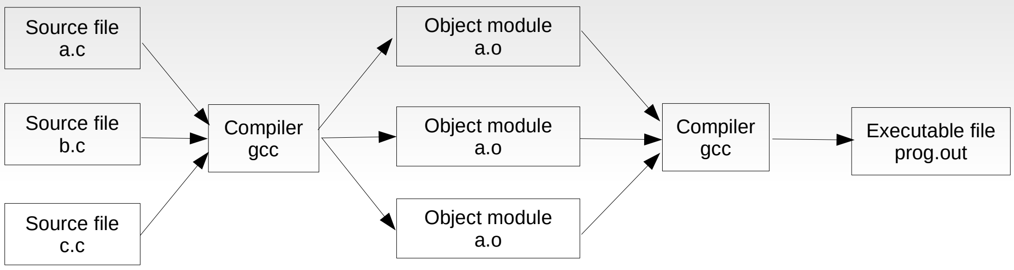
- User mode
  - Access restricted to process own adress space
  - Limited instruction set
- Supervisor mode
  - Full memory access
  - Full access to the instruction set
- Interrupt, trap
  - Asynchronous event
  - Illegal instruction
  - System call request

Process memory layout



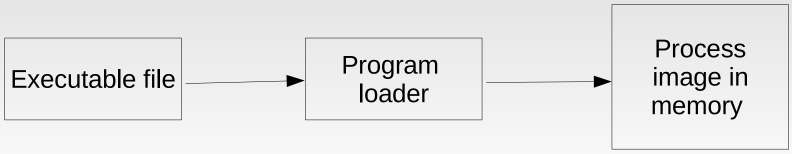
- Process execution state
  - Processor state
  - File descriptors
  - Memory allocation

Compiling



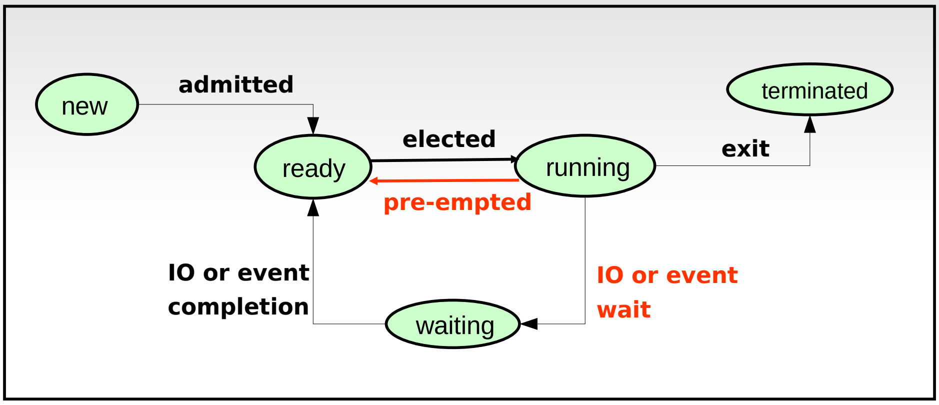
- Source files are compiled to object modules
- Object modules are linked into a single executable file
  - Example: gcc <source> [-o output]

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 (process)

- 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

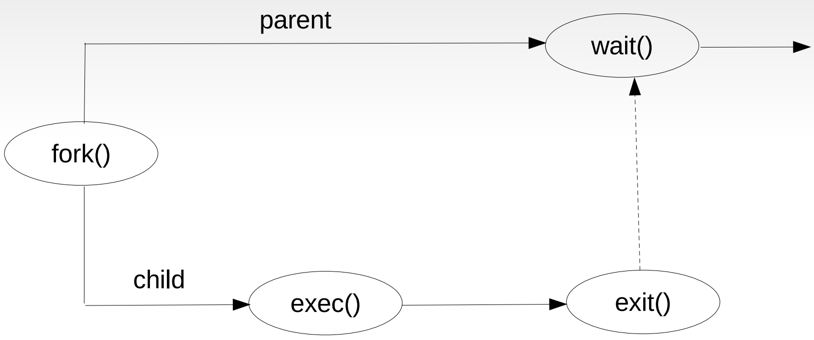
Process SVC overview

- 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
  - Open files are thus opened by both father and child. They should both close the files.
  - Some shared memory segment are still shared (e.g. shared libraries)
  - All other memory is 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 (process)

- How many processes are created ?

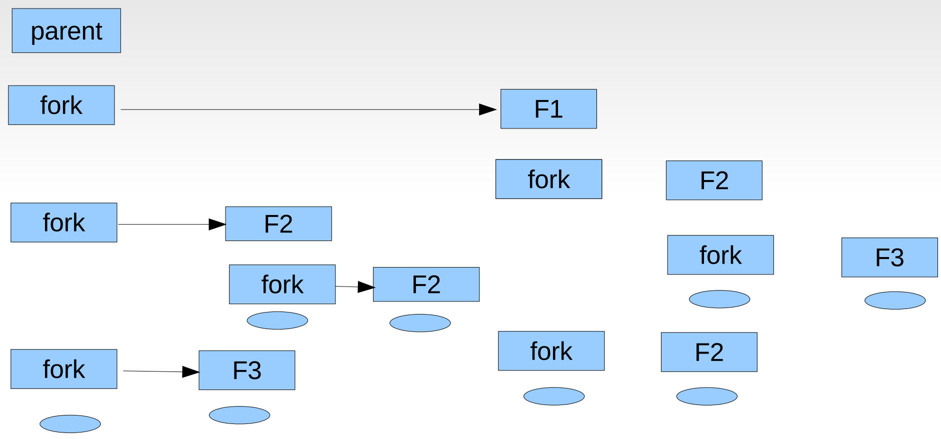
```
fork();
fork();
fork();
```

```
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);

}
```

3 forks


parent
fork
F1
fork
F2
fork
F2
fork
F2
fork
F3
fork
F2
fork
F3
fork
F2

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 (process)

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

or

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

Father/child synchronization

- The father process waits for the completion of one of its children
  - 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 (process)

```
#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)

Inter Process Communication

Daniel Hagimont

https://www.google.fr/search?q=daniel+hagimont+home+page

Motivations

- A process is blind, deaf, mute
- Create channels to disk, between processes, shared memory
- I/O redirections to disk
- Pipes
- Message queues
- Shared memory

Remember file descriptors

- A file is addressed through a descriptor
  - 0, 1 et 2 correspond to standard input, standard output, and standard error
  - The file descriptor number is returned by the open system call
- Basic operation
  - int open(const char *pathname, int flags);
    - O_RDONLY, O_WRONLY, O_RDWR …
  - int creat(const char *pathname, mode_t mode);
  - int close(int fd)
  - ssize_t read(int fd, void *buf, size_t count);
  - ssize_t write(int fd, void *buf, size_t count);

I/O redirection

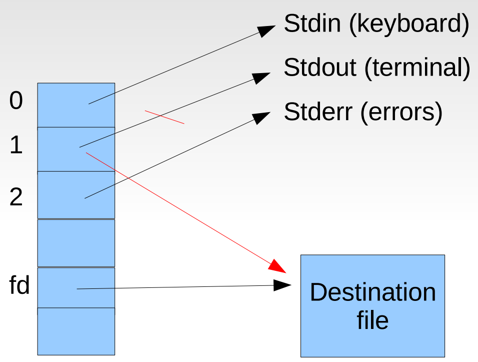
- From a shell, we can redirect the standard input/output
  - ls > foo.txt
    - Redirects the standard output (stdout) of "ls" towards file "foo.txt"
    - Here, notice that file descriptor 1 is replaced by the file descriptor of "foo.txt"
  - grep toto < foo.txt
    - Redirects the standard input (stdin) of "grep toto" to be taken from file "foo.txt"
    - Here, notice that file descriptor 0 is replaced by the file descriptor of "foo.txt"
- From a program, an API allow managing redirections

I/O redirection

- Descriptor duplication
  - dup2(int oldfd, int newfd);
  - Used to redirect standard I/O

```
#include <stdio.h>
#include <unistd.h>
int f;
/* redirect std input */

...
dup2(f,0) ;              // dupliquate f on descriptor 0
close(f);                // free f
...
```

I/O redirection



In shell

ls > foo.txt

In C

fd = open ("foo", …)

dup2(fd,1)

exec ("ls" ...)

Redirection of stdout to a file

Exercice (ipc)

- Copy with cat

```
int main (int argc, char *argv[]) {
    char *src, *dest;
    int fd;
    if (argc != 3) {
        printf("usage: copy <src> <dest>\n");
        return 0;
    }
    src = argv[1];
    dest = argv[2];
    fd = open(dest, O_CREAT | O_WRONLY | O_TRUNC,
                              S_IRUSR | S_IWUSR);
    if (fd == -1) {perror("error open");exit(0);}
    if (dup2(fd,1) == -1) {perror("error dup2");exit(0);}
    if (execlp("cat","cat", src, NULL) == -1) {perror("error execl");exit(0);}
}
```

Cooperation between processes

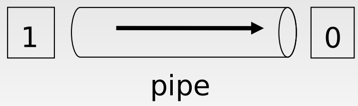
- Independent process cannot affect or be affected by the execution of another process
- Cooperating process can affect or be affected by the execution of another process. Advantages:
    - Information sharing
    - Computation speed-up
    - Modularity
    - Convenience
- Techniques
    - Pipes
    - Message queues
    - Shared memory

Pipe

- From a shell, we can connect the stdout of one process with the stdin of another process
  - ls : generate a list of files (one line for each file) on stdout
  - grep toto : read lines on stdin and print on stdout lines which include the word "toto"
  - ls | grep toto
    - The vertical bar is called a pipe
    - A pipe receives data from the stdout of the first process and send it to the stdin of the second process

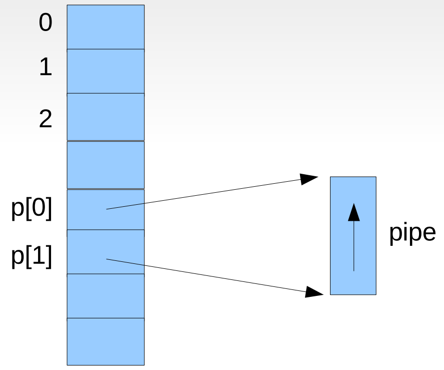
Pipe

- Communication mechanism between processes
  - Fifo structure
  - Limited capacity
  - Producer/consumer synchronization



- int pipe (int fds[2]);
  - Returns two file descriptors in fds[0] and fds[1]
  - Writes to fds[1] will be read on fds[0]
  - Returns 0 on success, -1 on error
- Operations on pipes
  - read/write/close – as with files
  - When fds[1] closed, read(fds[0]) returns 0 bytes (EOF)
  - When fds[0] closed, write(fds[1]): kill process with SIGPIPE

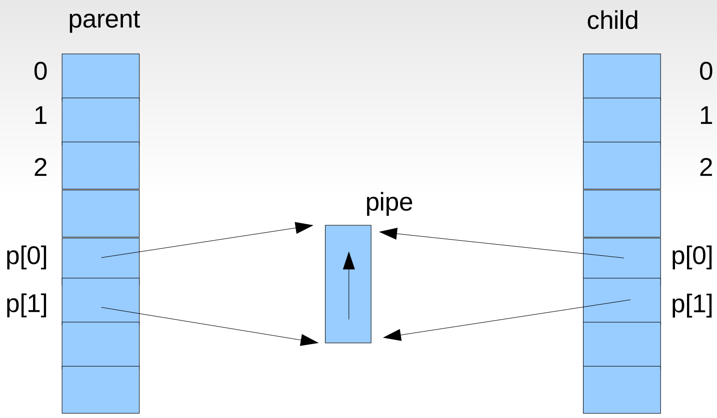
Pipe



In C

```
int p[2];

pipe(p);

write(p[1] ….);

read(p[0], ….);
```

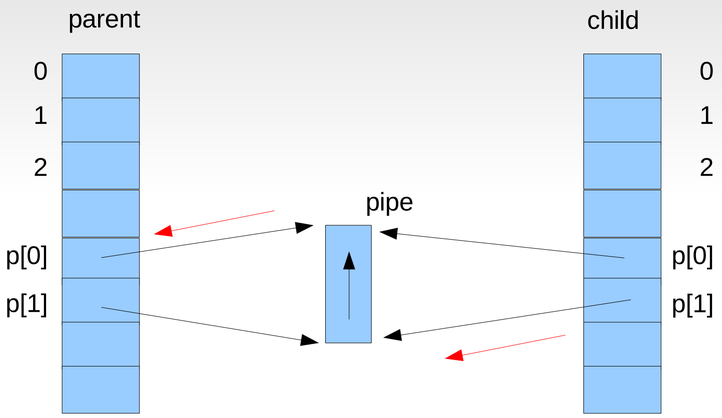
Pipe



In C

```
int p[2];

pipe(p);

If (fork() == 0) {
      // I am the child

} else {
      // I am the parent

}
```

Pipe

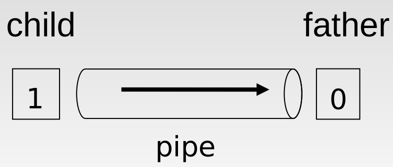


In C

```
Int p[2];

pipe(p);

if (fork() == 0) {
        // I am the child
        close(p[0]);
        write(p[1] ….);

} else {
        // I am the parent
        close(p[1]);
        read(p[0], ….);

}
```

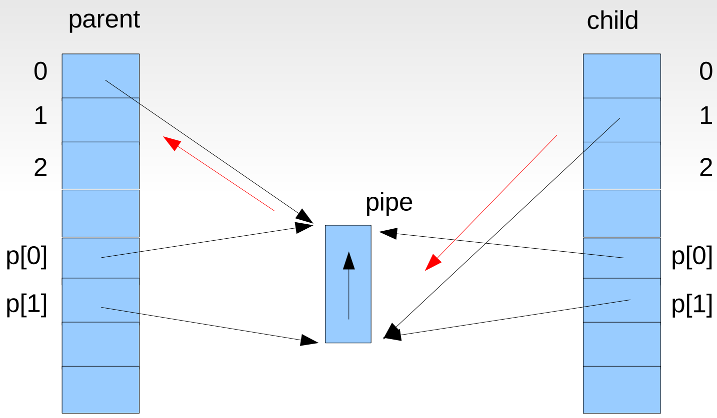
Exercise (ipc)

```
int main (int argc, char *argv[]) {
   int p[2];
   pipe (p);
   if (fork () == 0) {
        // child
        dup2 (p[1], 1);
        close (p[0]); close (p[1]);
        execlp("ps", "ps", "-ef", NULL);
   } else {
        //  father
        dup2 (p[0], 0);
        close (p[0]); close (p[1]);
        execlp("grep", "grep", "firefox", NULL);
   }
}
```



<u>In shell</u>
ps -ef | grep firefox

Pipe



dup2(p[0], 0);
close(p[0]; close(p[1]);
exec("grep" ….);

dup2(p[1], 1);
close(p[0]; close(p[1]);
exec('ps" ….);

In C

```
int p[2];

pipe(p);

if (fork() == 0) {
        // I am the child
        close(p[0];
        write(p[1] ….);

} else {
        // I am the parent
        close(p[1];
        read(p[0], ….);

}
```

Signals

- Signal : asynchronous notification
- From a shell, we can use signals by two means
  - Some signals are generated from the keyboard
    - SIGINT (ctrl-C), SIGSTOP (ctrl-S), SIGCONT (ctrl-Q), SIGTSTP (ctrl-Z)
    - Other signal : SIGTERM, SIGKILL ...
  - Kill <signal> <pid>
    - Allows sending a signal to a process (pid)
  - Example : start a process, suspend it (ctrl-Z), resume it (kill SIGCONT <pid>)
- From a program, an API allow managing signals

Signals

- A process may send a SIGSTOP, SIGTERM, SIGKILL signal to suspend, terminate or kill a process using the kill function:
  - int kill (int pid, int sig);
  - A lot of signals … see man pages
  - Some signals cannot be blocked (SIGSTOP and SIGKILL)
- Upon reception of a signal, a given handler is called. This handler can be obtained and modified using the signal function:
  - typedef void (*sighandler_t)(int); // handler
  - sighandler_t signal(int signum, sighandler_t handler); // set a handler

Signal example

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

void handler(int signal_num) {
   printf("Signal %d => ", signal_num);
   switch (signal_num) {
   case SIGTSTP:
      printf("pause\n");
      break;
   case SIGINT:
   case SIGTERM:
      printf("End of the program\n");
      exit(0);
      break;
   }
}
```

```
int main(void) {
  signal(SIGTSTP, handler);
  /* if control-Z */
  signal(SIGINT, handler);
  /* if control-C */
  signal(SIGTERM, handler);
  /* if kill process */
  while (1) {
    sleep(1);
    printf(".\n");
   }
printf("end");
exit(0);
}
```

- Signal handling is vulnerable to race conditions: another signal (even of the same type) can be delivered to the process during execution of the signal handling routine.
- The sigprocmask() call can be used to block and unblock delivery of signals.

Exercise (ipc)

- Without signals
  - Try control-C, control-Z
- With signals (previous slide)
  - Try control-C, control-Z

```
int main(void) {
  while (1) {
    sleep(1);
    printf(".\n");
  }
}
```

Message queue

- Creation/finding of a message queue
  - int msgget(key_t key, int msgflg);
- Control of the message queue
  - int msgctl(int msqid, int cmd, struct msqid_ds *buf);
- Emission of a message
  - int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
- Reception of a message
  - int msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);

Exercise (ipc)

creator

```
int main() {
   int msgid;
   key_t key = 1234;

   /* Create the queue */
   if ((msgid = msgget(key,
        IPC_CREAT | 0666)) < 0) {
        perror("msgget failedt");
        exit(1);
   }
}
```

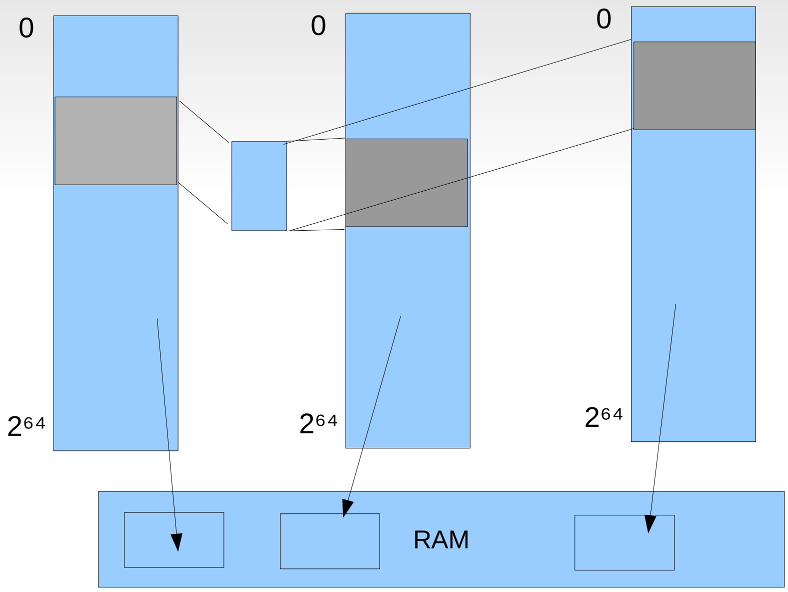
sender

```
struct message {
        long mtype;
        char mtext[20];
};

int main() {
   int msgid;
   key_t key = 1234;
   struct message msg;

   /* get the queue */
   if ((msgid = msgget(key, 0666)) < 0) {
         perror("msgget failedt");
         exit(1);
   }

   /* send a message */
   msg.mtype=1;
   strcpy(msg.mtext, "hello vietnam");
   if ((msgsnd(msgid,
               (void *)&msg,
               sizeof(struct message),
               0)) == -1) {
            perror("msgsnd failed");
            exit(1);
   }
}
```

receiver

```
struct message {
        long mtype;
        char mtext[20];
};

int main() {
   int msgid;
   key_t key = 1234;
   struct message msg;

   /* get the queue */
   if ((msgid = msgget(key, 0666)) < 0) {
          perror("msgget failedt");
          exit(1);
   }

   /* receive a message */
   if ((msgrcv(msgid, (void *)&msg,
              sizeof(struct message),
              0,0)) == -1) {
         perror("msgsnd failed");
         exit(1);
   }
   printf("received : %s\n", msg.mtext);
}
```

Shared memory



Shared memory segment

- A process can create/use a shared memory segment using:
  - int shmget(key_t key, size_t size, int shmflg);
  - The returned value identifies the segment and is called the shmid
  - The key is used so that process indeed get the same segment.
- The owner of a shared memory segment can control access rights with shmctl()
- Once created, a shared segment should be attached to a process address space using
  - void *shmat(int shmid, const void *shmaddr, int shmflg);
- It can be detached using int shmdt(const void *shmaddr);
- Can also be done with the mmap function
- Example

Exercise (ipc)

creator

```
int main() {
   int shmid;
   key_t key = 1234;
   /* Create the segment */
   if ((shmid = shmget(key, 10,
            IPC_CREAT | 0666)) < 0) {
            perror("shmget failed");
            exit(1);
   }
}
```

writer

```
int main() {
   int shmid, i, t;
   char *shm;
   key_t key = 1234;

   /* Get the segment */
   if ((shmid = shmget(key, 10,
                        0666))< 0) {
          perror("shmget failed");
          exit(1);
   }

   /* Attach the segment */
   if ((shm = shmat(shmid, NULL,
                   0)) == (void *) -1) {
           perror("shmat failed");
           exit(1);
   }

   t = 0;
   while (1) {
      sleep(1);
      for (i=0;i<5;i++) shm[i] = 'a'+t;
      shm[i] = 0;
      printf("wrote : %s\n",shm);
      t++;
   }
}
```

reader

```
int main() {
   int shmid, i, t;
   char *shm;
   key_t key = 1234;

   /* Get the segment */
   if ((shmid = shmget(key, 10,
                        0666))< 0) {
          perror("shmget failed");
          exit(1);
   }

   /* Attach the segment */
   if ((shm = shmat(shmid, NULL,
                   0)) == (void *) -1) {
          perror("shmat failed");
          exit(1);
   }

   while (1) {
      sleep(1);
      printf("read : %s\n",shm);
   }
}
```

Mmap

- Another interface for sharing memory

```
void * mmap (void * addr, size_t len, int prot, int flags, int fd, off_t offset);
```

returned address
of shared segment

requested address
of shared segment

requested size
of shared segment

PROT_READ	read access
PROT_WRITE	write access
PROT_EXEC	execution access
PROT_NONE	no access (SIGSEGV)

MAP_SHARED or MAP_PRIVATE
MAP_ANON   no file
MAP_FIXED   mandatory requested address

file descriptor

offset in the file

Mmap examples

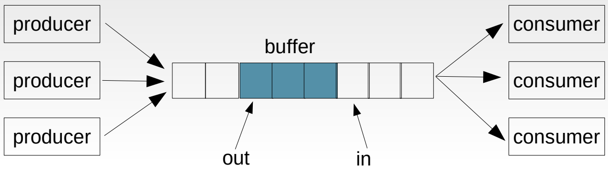
```
long pagesize = sysconf(_SC_PAGESIZE);
int cf = open("content",O_RDWR);
char* base = mmap(0,pagesize, PROT_WRITE|PROT_READ,MAP_SHARED,cf,0);
```

```
/* adresses [base,base+pagesize[ accessible in read/write mode
      - can be shared between indepedent processes
*/
```

```
char* b = mmap(0,pagesize,PROT_WRITE|PROT_READ,
                                   MAP_SHARED|MAP_ANON,-1,0);
```

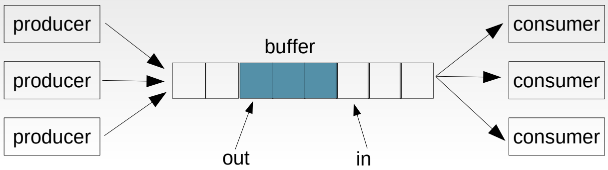
```
/* adresses [base,base+pagesize[ accessible in read/write mode
      - has to be shared with fork
*/
```

Use-case: producer-consumer



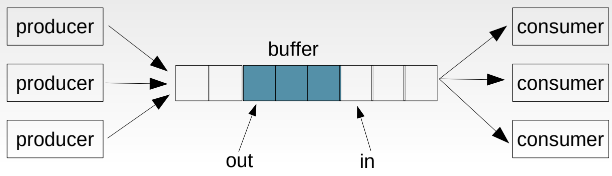

```
#define BUFFER_SIZE 10

typedef struct {
        char product;
        int price;
} item;

item buffer [BUFFER_SIZE];
int in = 0; // where to produce
int  out = 0; // where to consume
int nb = 0; // number of items
```

Use-case:
producer-consumer




```
void produce(item *i) {
        while (nb == BUFFER_SIZE) {
                // do nothing – no free place in buffer
        }
        memcopy(&buffer[in], i, sizeof(item));
        in = (in+1) % BUFFER_SIZE;
        nb++;
}
```

Use-case: producer-consumer




```
item *consume() {
        item *i = malloc(sizeof(item));
        while (nb == 0) {
                // do nothing – nothing to consume
        }
        memcopy(i, &buffer[out], sizeof(item));
        out = (out+1) % BUFFER_SIZE;
        nb--;
        return i;
}
```

Socket

- A socket is defined as an endpoint for communication
- Used for remote communication
- Basic message passing API
- Identified by an IP address and port
- The socket 161.25.19.8:1625 refers to port 1625 on host 161.25.19.8
- Communication between a pair of sockets and bidirectionnal
- Another Teaching Unit (networking)

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 (2.3)

Threads

Daniel Hagimont

https://www.google.fr/search?q=daniel+hagimont+home+page

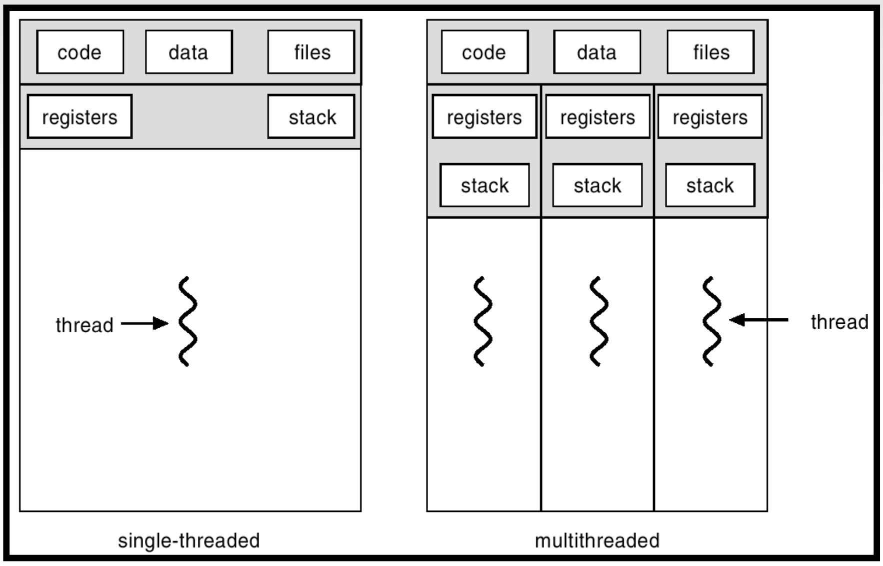
Process

- Unix process: heavy
  - Context: large data structure (includes an address space)
  - Protected address space
    - Address space not accessible from another process
    - Sharing / communication
      - At creation time (fork)
      - Via shared memory segments
      - Via messages (queues, sockets)
  - Communication is costly

Threads

- Light weight process
  - Light weight context
    - A shared context: address space, open files …
    - A private context: stack, registers …
- Faster communication within the same address space
  - Message exchange, shared memory, synchronization
- Useful for concurrent/parallel applications
  - Easier
  - More efficient
  - Multi-core processors

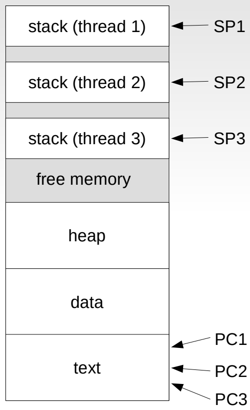
Single-threaded vs multi-threaded processes



A.Sylberschatz

Multi-threaded process

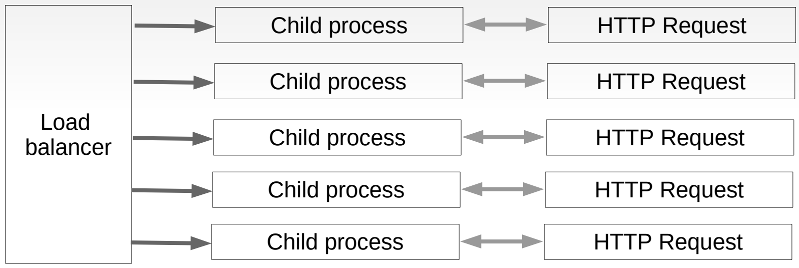
- Each thread has:
  - Private stack
  - Private stack pointer
  - Private program counter private register values
- Share:
  - Common text section (code)
  - Common data section (global data)
  - Common heap (dynamic allocations)
  - File descriptors (opened files)
  - Signals



Multi-threaded process

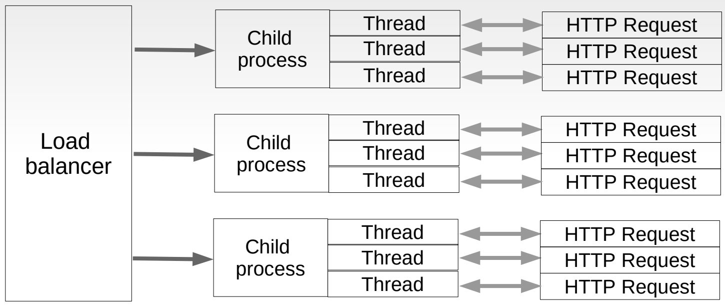
- Threads: same goal as processes
  - Do several thing at the same time
  - Increase CPU utilization
  - Increase responsiveness
- What's the difference
  - Multi-process with fork(): resource cloning
  - Multi-thread process: resource sharing

Some multi-process architectures



Apache HTTPD Prefork Model

Some multi-process architectures



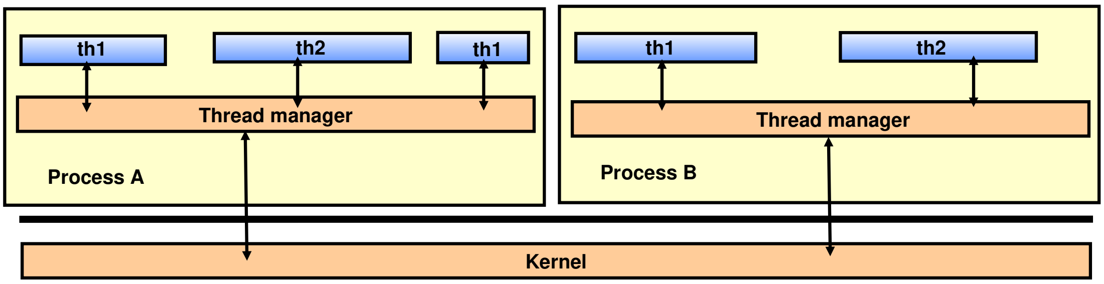
Apache HTTPD Worker Model

Exercise (thread)

- Show the number of threads for process firefox or google-chrome
  - ps with NLWP (number of lightweight processes) option
    - e.g. ps -o nlwp <processId>
  - Count number of subdirectories in /proc/<processId>/task

User-level Threads

- Implemented in a user level library
- Unmodified Kernel
- Threads and thread scheduler run in the same user process

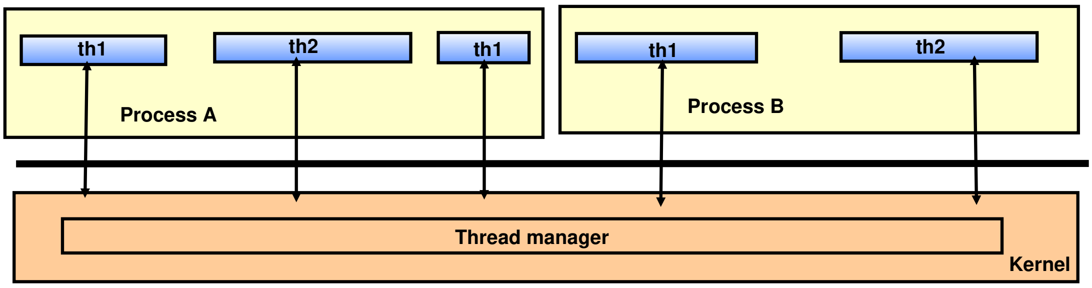


Advantages and disadvantages of User-level threads

- Parallelism (-)
  - No real parallelism between the threads within a process
- Efficiency (+)
  - Quick context switch
- Blocking system call (-)
  - The process is blocked in the kernel
  - All thread are blocked until the system call (I/O) is terminated

Kernel level threads

- Thread managed by the kernel
- Thread creation as a system call
- When a thread is blocked, the processor is allocated to another thread by the kernel



Advantages and disadvantages of Kernel-level threads

- Blocking system call (+)
  - When a thread is blocked due to an SVC call, threads in the same process are not
- Real Parallelism (+)
  - N threads in the same process can run on K processors (multi-core)
- Efficiency (-)
  - More expensive context switch / user level threads
  - Every management operation goes throught the kernel
  - Require more memory

POSIX Threads : pthreads API

- int pthread_create (pthread_t *thread, const pthread_attr_t *attr, void * (*start_routine)(void *), void *arg);
  - Creates a thread
- pthread_t pthread_self (void);
  - Returns id of the current thread
- int pthread_equal (pthread_t thr1, pthread_t thr2);
  - Compare 2 thread ids
- void pthread_exit (void *status);
  - Terminates the current thread
- int pthread_join (pthread_t thr, void **status);
  - Waits for completion of a thread
- int pthread_yield(void);
  - Relinquish the processor
- Plus lots of support for synchronization [next lecture]

Exercise (thread) (1/2)

```
#include <pthread.h>

void * ALL_IS_OK = (void *)123456789L;

char *mess[2] = { "thread1", "thread2" };

void * writer(void * arg)
{
  int i, j;

  for(i=0;i<10;i++) {
    printf("Hi %s! (I'm %lx)\n", (char *) arg, pthread_self());
    j = 800000; while(j!=0) j--;
  }

  return ALL_IS_OK;
}
```

Exercise (thread) (2/2)

```
int main(void)
{ void * status;
  pthread_t writer1_pid, writer2_pid;

   pthread_create(&writer1_pid, NULL, writer, (void *)mess[1]);
   pthread_create(&writer2_pid, NULL, writer, (void *)mess[0]);

   pthread_join(writer1_pid, &status);
   if(status == ALL_IS_OK)
      printf("Thread %lx completed ok.\n", writer1_pid);

   pthread_join(writer2_pid, &status);
   if(status == ALL_IS_OK)
      printf("Thread %lx completed ok.\n", writer2_pid);

   return 0;
}
```

Fork(), exec()

- What happens if one thread of a program calls fork()?
  - Does the new process duplicate all threads ? Or is the newprocess single-threaded ?
  - Some UNIX systems have chosen to have two versions of fork()
- What happens if one thread of a program calls exec()?
  - Generally, the new program replace the entire process, including all threads.

Resources you can read

- Pthreads
  - https://computing.llnl.gov/tutorials/pthreads/
- Operating System Concepts, 10th Edition, Abraham Silberschatz, Peter B. Galvin, Greg Gagne
  - http://os-book.com/
  - Chapters 4
- Modern Operating Systems, Andrew Tanenbaum
  - http://www.cs.vu.nl/~ast/books/mos2/
  - Chapter 2 (2.2)

Scheduling

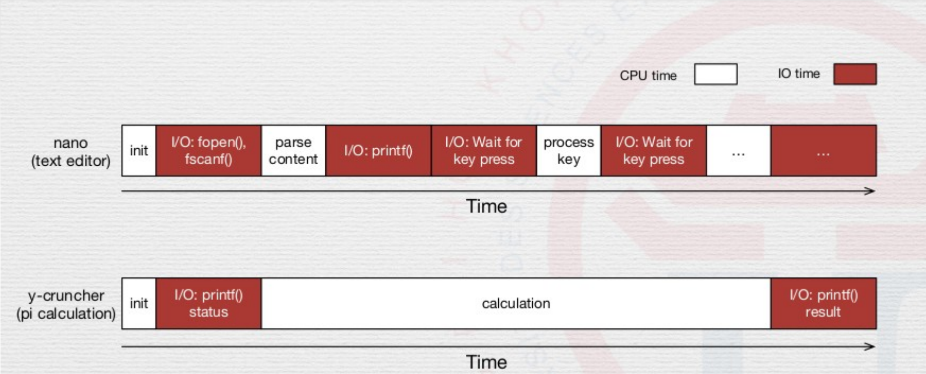
Daniel Hagimont

https://www.google.fr/search?q=daniel+hagimont+home+page

Scheduler

- The kernel component that
  - Select a process to be executed on a CPU
- Maximize CPU usage
  - For a set of process
  - With one or more CPU
- Different characteristics of processes
  - CPU bound: spend more time on computation
  - I/O bound: spend more time on I/O devices (reading/writing on disk …)
- Process execution consists of
  - CPU execution
  - I/O wait

Task scheduling

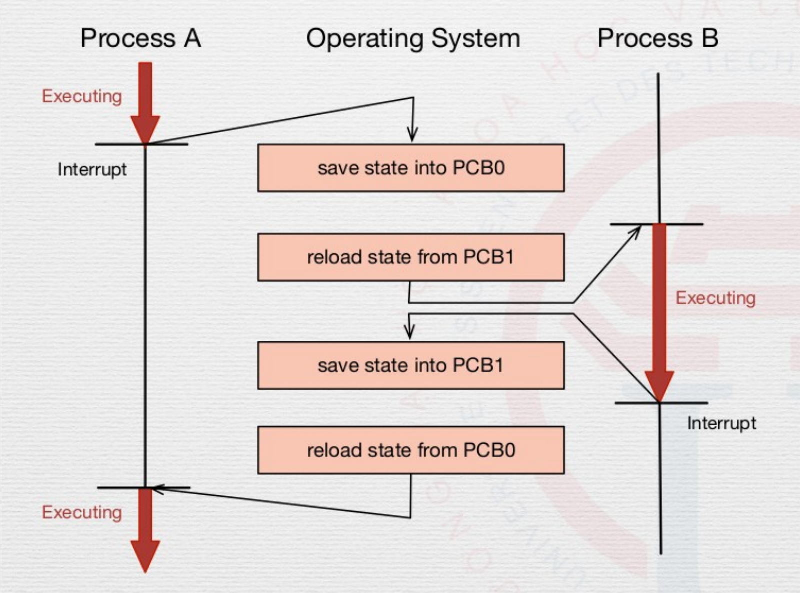


Different tasks with different priorities

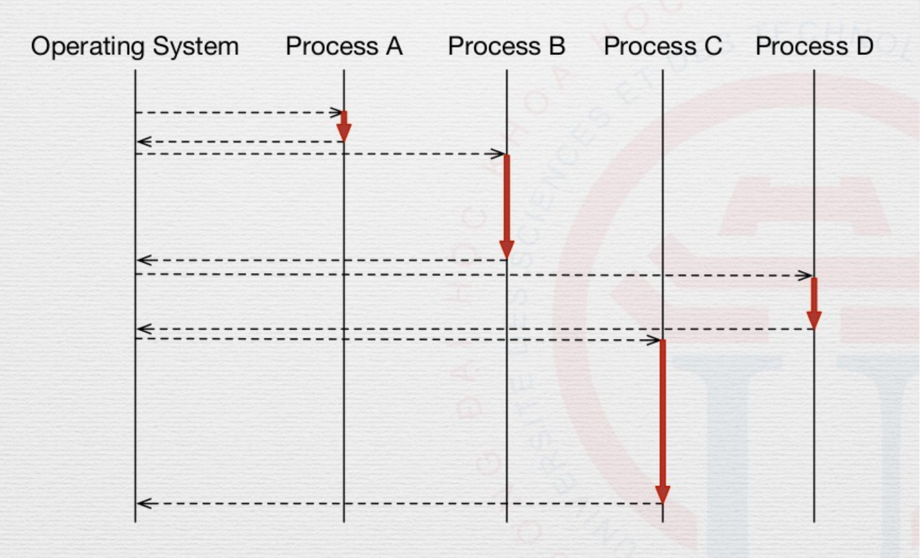
Characteristics of schedulers

- Ability to pause running processes
  - Preemption: OS forcely pauses running processes
  - Non-preemption: only at the end of tasks or process willing to pause itself
- Duration between each switch
  - Short term scheduler: milliseconds (fast, responsive)
  - Long term scheduler: seconds/minutes (batch jobs)
- Switch between processes
  - Save data of old process
  - Load previously saved data of new process
  - Context switch is overhead

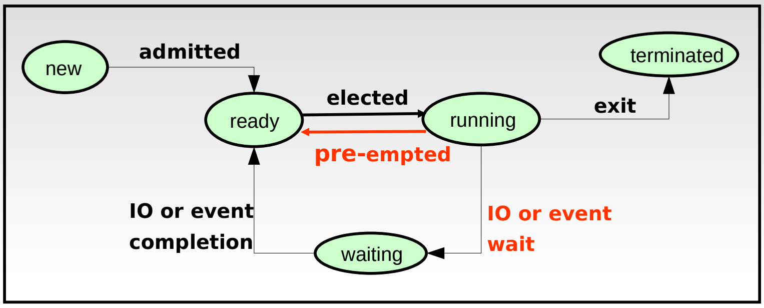
Context switches



Context switches



Context switches



- When to perform a context switch
  - Process switches from running to waiting (IO) – non preemptive
  - Process terminates (exit) – non preemptive
  - Process switches from running to ready – preemptive
  - Process switches from witing to ready - preemptive

Process management by the OS

- Process queues
  - Ready queue (ready processes)
  - Device queue (Process waiting for IO)
  - Blocked Queue (Process waiting for an event)
  - ...
- OS migrates processes across queues

CPU Allocation to processes

- The scheduler is the OS's part that manages CPU allocation
- Criteria / Scheduling Algorithm
  - Fair (no starvation)
  - Minimize the waiting time for processes
  - Maximize the efficiency (number of jobs per unit of time)

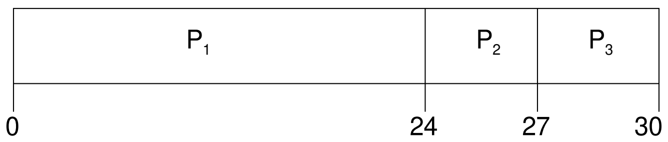
Simple scheduling algorithms

- Non-pre-emptive scheduler
  - FCFS (First Come First Served)
    - Fair, maximize efficiency
- Pre-emptive scheduler
  - SJF (Shortest Job First)
    - Priority to shortest task
    - Require to know the execution time (model estimated from previous execution)
    - Unfair but optimal in term of response time
  - Round Robin (fixed quantum)
    - Each processus is affected a CPU quantum (10-100 ms) before pre-emption
    - Efficient (unless the quantum is too small), fair / response time (unless the quantum too long)

First-Come, First-Served (FCFS) non pre-emptive (1/2)

Process's	execution time
P1	24
P2	3
P3	3

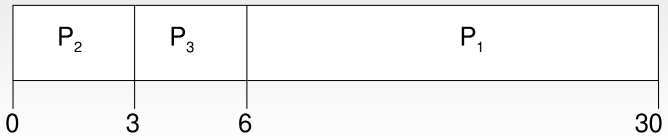
- Let's these processes come in this order : P1,P2,P3



- Response time of P1 = 24; P2 = 27; P3 = 30
- Mean time : (24 + 27 + 30)/3 = 27

First-Come, First-Served (FCFS) (2/2)

- Let's these processes come in this order : P2 , P3 , P1 .



- Response time : P1 = 30; P2 = 3; P3 = 6
- Mean time : (30 + 3 + 6)/3 = 13
- Better than the previous case
- Schedule short processes before

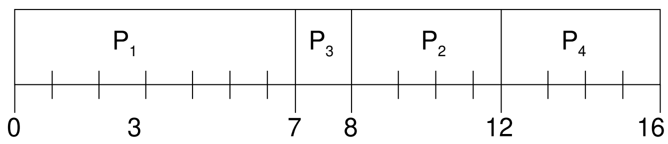
Shortest-Job-First (SJF)

- Associate with each process its execution time
- Two possibilities :
  - Non pre-emptive – When a CPU is allocated to a process, it cannot be pre-empted
  - Pre-emptive – if a new process comes with a shorter execution time than the running one, this last process is pre-empted (Shortest-Remaining-Time-First - SRTF)
- SJF is optimal / mean response time

Non Pre-emptive SJF

Process	Come in	Exec. Time
P1	0.0	7
P2	2.0	4
P3	4.0	1
P4	5.0	4

- SJF (non pre-emptive)

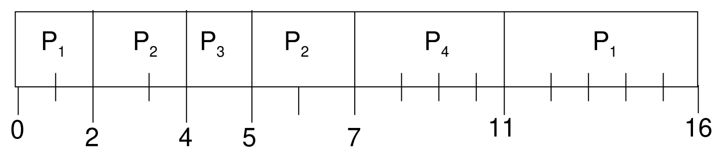


- Mean response time = (7 + 10 + 4 + 11)/4 = 8

Pre-emptive SJF (SRTF)

Process	Come in	Exec. Time
P1	0.0	7
P2	2.0	4
P3	4.0	1
P4	5.0	4

- SJF (pre-emptive)

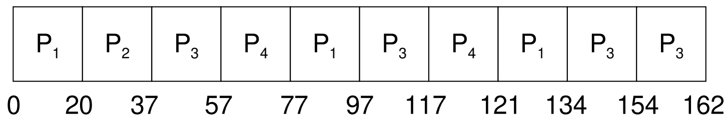


- Mean response time = (16 + 5 + 1 + 6)/4 = 7

Round Robin (Quantum = 20ms)

Process	Exec Time
P1	53
P2	17
P3	68
P4	24

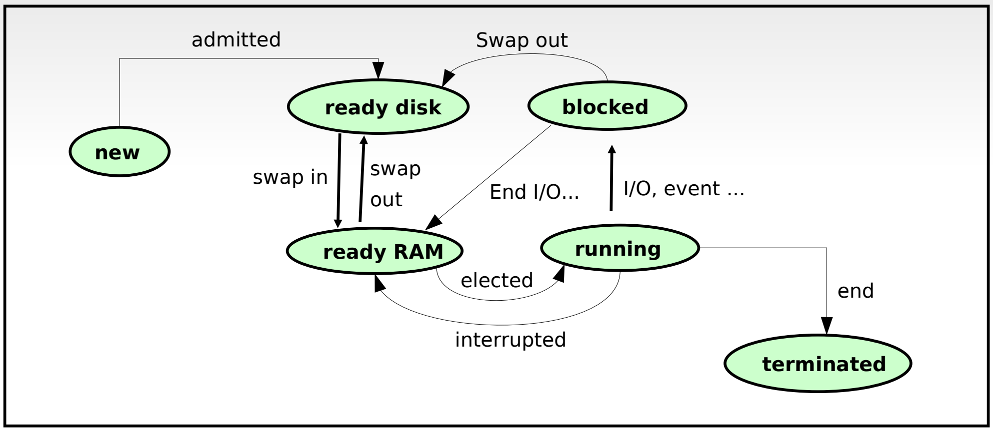
- Efficiency and mean response worse than SJF
- But don't need to estimate execution time



Multiple level scheduling algorithm

- The set of ready processes too big to fit in memory
- Part of these processes are swapped out to disk. This increases their activation time
- The elected process is always choosen from those that are in memory
- In parallel, another scheduling algorithm is used to manage the migration of ready process between disk and memory

Two level scheduling



A scheduler implementation (sched)

```
void thread0() {
	int i,k;
	for (i=0;i<10;i++) {
		printf("thread 0\n");
		sleep(1);
	}
}
void thread1() {
	int i,k;
	for (i=0;i<10;i++) {
		printf("thread 1\n");
		sleep(1);
	}
}
void thread2() {
	int i,k;
	for (i=0;i<10;i++) {
		printf("thread 2\n");
		sleep(1);
	}
}
```

Procedures executed by the 3 threads

A scheduler implementation (sched)

```
#include <stdio.h>
#include <setjmp.h>
#include <signal.h>
#include <stdlib.h>
#include <ucontext.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

#define MAX_THREAD 3
#define STACK_SIZE 16000
#define TIME_SLICE 4

void thread0();
void thread1();
void thread2();
void schedule(int sig);

ucontext_t uctx_main;

void (*thread_routine[MAX_THREAD])() = {thread0, thread1, thread2};
ucontext_t thread_save[MAX_THREAD];
char thread_stack[MAX_THREAD][STACK_SIZE];
int thread_state[MAX_THREAD];
int current;
```

- save area (context)
- stacks
- state (active, dead ...)

A scheduler implementation (sched)

```
int main() {
        int i;
        for (i=0;i<MAX_THREAD;i++) {
                if (getcontext(&thread_save[i]) == -1)
                        { perror("getcontext"); exit(0); }
                thread_save[i].uc_stack.ss_sp = thread_stack[i];
                thread_save[i].uc_stack.ss_size = sizeof(thread_stack[i]);
                thread_save[i].uc_link = &uctx_main;
                makecontext(&thread_save[i], thread_routine[i], 0);
                thread_state[i] = 1;
                printf("main: thread %d created\n",i);
        }

        signal(SIGALRM, schedule);
        alarm(TIME_SLICE);

        printf("main: swapcontext thread 0\n");
        current = 0;
        if (swapcontext(&uctx_main, &thread_save[0]) == -1)
                { perror("swapcontext"); exit(0); }

        while (1) {
                printf("thread %d completed\n", current);
                thread_state[current] = 0;
                schedule(0);
        }
}
```

Initialize each thread

Program next alarm

Switch from current thread to thread0

A scheduler implementation (sched)

```
void schedule(int sig) {
        int k, old;
        alarm(TIME_SLICE);
        old = current;
        for (k=0;k<MAX_THREAD;k++) {
                current = (current + 1) % MAX_THREAD;
                if (thread_state[current] == 1) break;
        }
        if (k==MAX_THREAD) {
                printf("last thread completed: exiting\n");
                exit(0);
        }
        printf("schedule: save(%d) restore (%d)\n",old, current);
        if (swapcontext(&thread_save[old], &thread_save[current]) == -1)
                { perror("swapcontext"); exit(0); }
}
```

Select the next active thread

Context switch

Exercise (sched)

```
hagimont@hagimont-pc:~/shared/cours/enseeiht/cours/Systemes/scheduler$ gcc sched
-ctx.c -o sched
hagimont@hagimont-pc:~/shared/cours/enseeiht/cours/Systemes/scheduler$ ./sched
main: thread 0 created
main: thread 1 created
main: thread 2 created
main: swapcontext thread 0
thread 0
thread 0
thread 0
thread 0
schedule: save(0) restore (1)
thread 1
thread 1
thread 1
thread 1
schedule: save(1) restore (2)
thread 2
thread 2
thread 2
thread 2
schedule: save(2) restore (0)
thread 0
thread 0
thread 0
thread 0
schedule: save(0) restore (1)
thread 1
thread 1
thread 1
^C
hagimont@hagimont-pc:~/shared/cours/enseeiht/cours/Systemes/scheduler$
```

Resources you can read

- Operating System Concepts, 10th Edition, Abraham Silberschatz, Peter B. Galvin, Greg Gagne
  - http://os-book.com/
  - Chapters 5
- Modern Operating Systems, Andrew Tanenbaum
  - http://www.cs.vu.nl/~ast/books/mos2/
  - Chapter 2 (2.5)

Synchronization

Daniel Hagimont

https://www.google.fr/search?q=daniel+hagimont+home+page

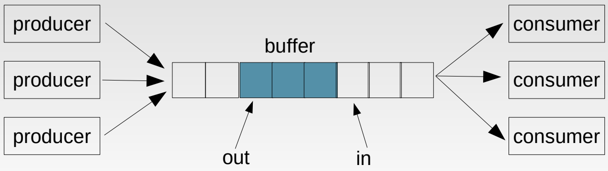
Problem statement

(1) y := read_account(1);
(2) x := read_account(2);
(3) x := x + y;
(4) write_account(2, x);
(5) write_account(1, 0);

(a) v := read_account(2);
(b) v = v - 100;
(c) write_account(2, v);

- Account 2 is shared between both executions
- Variables x, y, v are local
- Executions are performed in parallel and instructions can be intertwined
- (1) (2) (3) (4) (5) (a) (b) (c) is consistent (200/200)(0,300)
- (1) (a) (b) (c) (2) (3) (4) (5) is consistent (200/200)(0,300)
- (1) (2) (a) (3) (b) (4) (c) (5) is not consistent (200/200)(0,100)

Problem statement




```
#define BUFFER_SIZE 10

typedef struct {
	char product;
	int amount;
} item;

item buffer [BUFFER_SIZE];
int in = 0; // where to produce
int  out = 0; // where to consume
int nb = 0; // number of items
```

```
void produce(item *i) {
	while (nb == BUFFER_SIZE) {
		// do nothing – no free place in buffer
	}
	memcopy(&buffer[in], i, sizeof(item));
	in = (in+1) % BUFFER_SIZE;
	nb++;
}
```

```
item *consume() {
	item *i = malloc(sizeof(item));
	while (nb == 0) {
		// do nothing – nothing to consume
	}
	memcopy(i, &buffer[out], sizeof(item));
	out = (out+1) % BUFFER_SIZE;
	nb--;
	return i;
}
```

Problem statement

- N processes all competing to use some shared data
  - A critical section is a code fragment, in which the shared data is accessed
- Problem:
  - Ensure shared data consistency
- Ensure mutual exclusion
  - When one process is executing in one critical section, no other process is allowed to execute in this critical section

Desired properties

- Mutual Exclusion
  - Only one thread can be in a given critical section at a time
- Progress
  - If no process currently in a given critical section, one of the processes trying to enter will eventually get in
- Fairness
- No starvation

Critical section

- n processes: P0, P1, .., Pn
- P0, P1, .., Pn use a set of shared variables a, b, c, …

- Structure of a process Pi :

```
…
<enter section>       // enter mutex
<access a,b,c,..>         // critical section
<exit section>        // leave mutex
…
```

Software implementation (1)

Shared data :

```
boolean busy = false;
```

Pi :

```
while (busy) ;      (1)   // busy waiting
busy = true;
<critical section>
busy = false;
```

- No mutual exclusion if context switch at (1)
  - Test and set are not atomic

It seems to work

```
while (busy) ;
busy = true;
<critical section>
busy = false;
```

P1

Busy ? No

Busy = true

In critical section

Busy = false

P2

Busy ? Yes … looping
Busy ? Yes … looping
Busy ? No .. stop looping
Busy = true

In critical section

Busy = false

It doesn't work

```
while (busy) ;
busy = true;
<critical section>
busy = false;
```

P1	P2
Busy ? No	Busy ? No
Busy = true	Busy = true
In critical section	In critical section
Busy = false	Busy = false

Software implementation (2)

Shared data :

```
int turn = 0;  // turn = i  : Pi's turn to enter
```

Pi (0 or 1):

```
while (turn != i);          // busy waiting
<critical section>
turn =1-i;
```

- Mutual exclusion
- Can be generalized to N processes
- Progress issue
- Many other software solutions, but not satisfactory

Synthesis

- Software solutions
  - Complex
  - Not very efficient
- Hardware solutions
  - Masking interrupts
  - Test&Set

Masking interrupts

- Entry section : mask the IT
- Exit section : unmask the IT

- Cannot control the time spent in critical section
- Acceptable if the critical section exec time is short
- Cannot be used with multiprocessors

Test&Set instruction

- Most CPUs support atomic read-[modify-]write
- Example: int test_and_set (int *lockp);
  - Atomically sets *lockp = 1 and returns old value

```
int Test&Set (int *b) {
    // set b to 1, and return initial value of b
    int res = *b;
    *b = 1;
    return res;
}
```

Test&Set critical section

Shared data :

```
int busy = 0; // false
```

Pi:

```
 while (Test&Set (&busy));
<critical section>
busy = 0;
```

- Busy waiting
- Starvation issue (not FIFO)

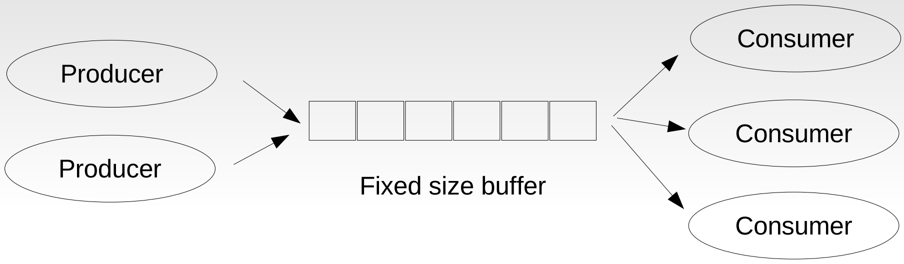
Sleep and wake up solutions

- Previous solution disadvantage :
  - CPU wasting (polling)
- Sleep and wake up solutions :
  - Block a process when it cannot enter a critical section
  - Wake up when the critical section is free
- Different abstractions
  - Lock
  - Semaphore
  - Monitor

Locks

- Simple synchronization primitives
  - Lock/unlock function
  - Only one process can go through a lock at the same time
  - Based on sleep/wakeup
- Different interfaces, implementations, properties (fifo ...)
  - e.g. Thread packages :
    - void mutex_init (mutex_t *m, ...);
    - void mutex_lock (mutex_t *m);
    - int mutex_trylock (mutex_t *m);
    - void mutex_unlock (mutex_t *m);

Producer Consumer example



- Fixed size buffer
- Variable number of producers and consumers

Producer Consumer example

Shared data:

```
int bufferSz = N;
int in = 0, out = 0, nb = 0;
Msg buffer[] = new Msg[bufferSz];
```

```
produce (Msg msg) {
    buffer[in] = msg;
    in = in + 1 % bufferSz;
    nb++;
}
```

```
Msg Consume {
    Msg msg = buffer[out];
    out = out + 1 % bufferSz;
    nb--;
    return msg;
}
```

Producer Consumer with locks

locks are not sufficient

Shared data:

```
int bufferSz = N;
int in = 0, out = 0, nb = 0;
Msg buffer[] = new Msg[bufferSz];
Lock mutex = new Lock();
```

```
produce (Msg msg) {
        mutex.lock();
        while (nb == bufferSz) {
            mutex.unlock();
            yield(); // back to ready queue
            mutex.lock();
        }
        buffer[in] = msg;
        in = in + 1 % bufferSz;
        nb++;
        mutex.unlock();
}
```

busy waiting

```
Msg Consume {
        mutex.lock();
        while (nb == 0) {
            mutex.unlock();
            yield(); // ready queue
            mutex.lock();
        }
        Msg msg = buffer[out];
        out = out + 1 % bufferSz;
        nb--;
        mutex.unlock();
    return msg;
}
```

Higher synchronization abstractions

- Principles
  - Use application's semantic to suspend/wake up a process that wait for a condition to happen
- Examples
  - Semaphore
  - Monitor

Semaphores (Dijkstra, 1965)

- Semaphore S :
  - counter S.c; //Model a ressource number or a condition
  - waiting queue S.f; //Waiting processes
- Think of a semaphore as a purse with a certain number of tokens
  - Suspend when no more token
  - Wake up when token released
- A Semaphore is initialized with an integer N
- Accessed through P() and V() operations

Semaphores (Dijkstra, 1965)

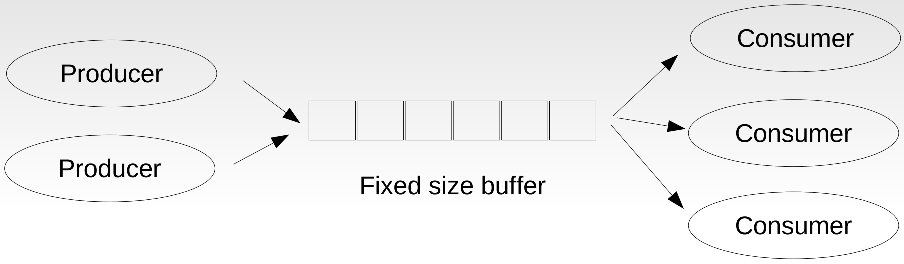
```
wait() or P ():
      S.c--
      if S.c < 0 do {
          // no more free resources
          put(myself, S.f);
          suspend();  // suspension
}

signal() or V ():
      S.c++;
      if (S.c <= 0) do {
        // at least 1 waiting process
          P = get(S.f);
          wakeup(p);
}
```

Critical section

Semaphores

- Counter S.c == S.c initial + NV - NP
  - NV is the number of V operations executed on the semaphore
  - NP is the number of P operations executed on the semaphore
- Counter S.c < 0
  - Correspond to the number of blocked processes
- Counter S.c > 0
  - Correspond to the number of available resources
  - Correspond also to the number of processes that can call a P operation without beeing blocked
- Counter S.c == 0 :
  - No more resources available and no blocked process
  - The next process that call P() will be blocked
- Can use semaphores to implement mutual exclusion (init =1)

Producer Consumer



- Condition to produce/consume
  - Produce: the buffer is not full
  - Consume: the buffer is not empty

Producer Consumer

- Can write producer/consumer with three semaphores
- Semaphore mutex initialized to 1
    - Used as mutex, protects buffer, in, out. . .
- Semaphore products initialized to 0 ($\approx$ number of items)
    - To block consumer when buffer is empty
- Semaphore places initialized to N ($\approx$ number of free locations)
    - To block producer when buffer is full

Producer Consumer

Shared data:

```
int bufferSz = N;
int in = 0, out = 0;
int nb = 0;
Msg buffer[] = new Msg[bufferSz];
Semaphore places = new Semaphore(bufferSz);
Semaphore products = new Semaphore(0);
Semaphore mutex = new Semaphore(1);
```

```
produce (Msg msg) {
        places.P();
        mutex.P();
        buffer[in] = msg;
        in = in + 1 % bufferSz;
        mutex.V();
        products.V();
    }
```

```
Msg Consume {
        products.P();
        mutex.P();
        Msg msg = buffer[out];
        out = out + 1 % bufferSz;
        mutex.V();
        places.V();
        return msg;
    }
```

Thread and Semaphore

- Thread packages typically provide semaphores
  - int sem_init(sem_t *sem, int pshared, unsigned int value);
  - int sem_post(sem_t *sem);
  - int sem_wait(sem_t *sem);
  - int sem_trywait(sem_t *sem);
  - int sem_timedwait(sem_t *sem, const struct timespec *abs_timeout);
  - int sem_getvalue(sem_t *sem, int *sval);

Semaphore conclusion

- They are quite error prone
  - If you call P instead of V, you'll have a deadlock
  - If you forget to protect parts of your code, you end up with a mutual exclusion violation
  - If you have "tokens" of different types, it may be hard to reason about
  - If by mistake you interchange the order of the P and V, you may violate mutual exclusion or end up with a deadlock.
- That is why people have proposed higher-level language constructs

Deadlock ??

- A correct solution is not always ensured by the semaphore :

```
P(mutex);
if …
     P(S);
    …
else
     …
     V(S);
V(mutex);
```

Possible deadlock

RULE: never block in a critical section without releasing the section

Monitor

- Programming language construct
- A Monitor contains
  - Data
  - Function (f1,..,fn)
  - Init function
  - Conditions
- Functions are executed in mutual exclusion
- A “condition variable” is a synchronization structure (a queue) associated to a “logical condition”
  - wait() suspends the caller
  - signal() wakes up a waiting process if any, else the signal is LOST
- In general, condition queues are FIFO

Monitor

monitor <monitor-name> {

<shared variables + conditions declarations>

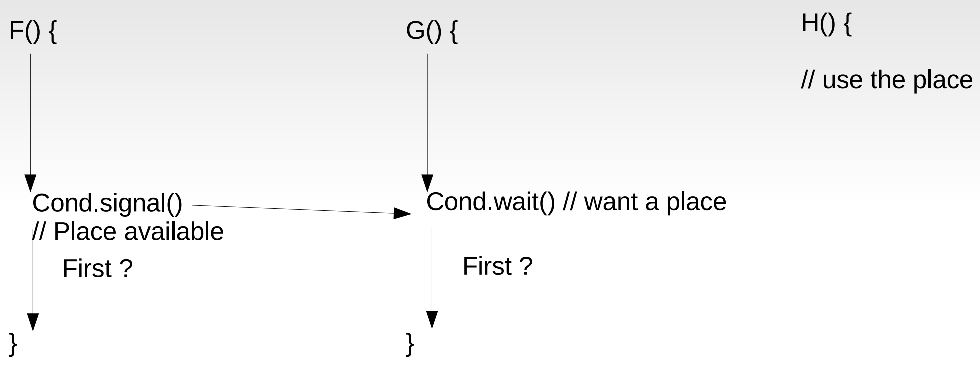
procedure init { initialization code }

procedure f1 (…) {
. . .
}
procedure f2 (…) {
. . .
}
procedure Pn (…) {
. . .
}
}

Monitor

- Only one process is running inside the monitor at a time
- On a signal
  - Either the signal sender keep the monitor (signal sender priority) = Signal and continue
  - Or the signal receiver acquires the monitor (signal receiver priority) = Signal and wait
- Monitor release
  - When the current procedure completes
  - When calling a wait operation

Monitor


F() {
Cond.signal()
// Place available
First ?
}
G() {
Cond.wait() // want a place
First ?
}
H() {
// use the place

Producer Consumer with monitors

```
Monitor ProdConsMonitor {
  int bufferSz, nb, in, out;
  Msg buffer[];
  Condition places, products;

procedure init() {
  bufferSz = N;
  nb = in = out = 0;
  buffer = new Msg[buffersz];
}
```

- Signal <u>receiver</u> priority

```
procedure produce(Msg msg) {
  if (nb==bufferSz)
     places.wait();
  buffer[in] = msg;
  in = in + 1 % bufferSz;
  nb++;
  products.signal();
}

procedure consume() : Msg {
  if (nb==0)
     products.wait();
  Msg msg = buffer[out];
  out = out + 1 % bufferSz;
  nb--;
  places.signal();
}
```

Producer Consumer with monitors

```
Monitor ProdConsMonitor {
  int bufferSz, nb, in, out;
  Msg buffer[];
  Condition places, products;

procedure init() {
  bufferSz = N;
  nb = in = out = 0;
  buffer = new Msg[buffersz];
}
```

- Signal <u>sender</u> priority

```
procedure produce(Msg msg) {
  while (nb==bufferSz)
     places.wait();
  buffer[in] = msg;
  in = in + 1 % bufferSz;
  nb++;
  products.signal();
}

procedure consume() : Msg {
  while (nb==0)
     products.wait();
  Msg msg = buffer[out];
  out = out + 1 % bufferSz;
  nb--;
  places.signal();
}
```

pthread synchronization

- pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
- int pthread_mutex_init (pthread_mutex_t *mutex, const pthread_mutex_attr *attr);
- int pthread_mutex_destroy (pthread_mutex_t *m);
- int pthread_mutex_lock (pthread_mutex_t *m);
- int pthread_mutex_trylock (pthread_mutex_t *m);
- int pthread_mutex_unlock (pthread_mutex_t *m);

pthread synchronization

- pthread_cond_t vc = PTHREAD_COND_INITIALIZER;
- int pthread_cond_init (pthread_cond_t *vc, const pthread_cond_attr *attr);
- int pthread_cond_destroy (pthread_cond_t *vc);
- int pthread_cond_wait (pthread_cond_t *vc, pthread_mutex_t *m);
- int pthread_cond_timedwait (pthread_cond_t *vc, pthread_mutex_t *m, const struct timespec *abstime);
- int pthread_cond_signal (pthread_cond_t *vc);
- int pthread_cond_broadcast (pthread_cond_t *vc);

Java synchronization

- For each object
  - one lock
  - one condition
- Monitor principles
  - Synchronized methods = executed in mutual exclusion
  - wait and notify/notifyAll to manage the condition

```
class Example {

 int cpt;  // shared data

 public void synchronized get() {
       if (cpt <= 0) wait();
       cpt--;
 }
 public void synchronized put() {
       cpt++;
       notify();
 }
}
```

Exercise
reader/writer with semaphores

- A shared document
- Users can read/write the document

```
beginRead()                  beginWrite()
    < reading >                  < writing >
endRead()                    endWrite()
```

- Multiple readers / single writer

Exercise
reader/writer with semaphores

Shared data:

```
int nbReaders = 0;
Semaphore mutex = new Semaphore(1);
Semaphore exclusive = new Semaphore(1);
```

```
beginRead () {
        mutex.P();
        If (nbReaders == 0)
            blockwriters.P();
        nbReaders ++;
        mutex.V();
}
```

```
endRead () {
        mutex.P();
        nbReaders --;
        If (nbReaders == 0)
            blockwriters.V();
        mutex.V();
}
```

```
beginWrite () {
    blockwriters.P();
}
```

```
endWrite () {
        blockwriters.V();
}
```

- Potential starvation of writers

Exercise
reader/writer with monitors

```
monitor ReaderWriter () {
  int nbReaders;
  boolean writer;
  Condition canRead, canWrite;

procedure init() {
  nbReaders = 0;
  writer = false;
}

procedure beginRead()
  nbReader++;
  if (writer) canRead.wait();
  canRead.signal();
}
```

```
procedure endRead() {
  nbReader--;
  if (nbReaders == 0) canWrite.signal();
}

procedure beginWrite() {
  if ((nbReaders > 0) || (writer))
        canWrite.wait();
  writer = true;
}

procedure endWrite() {
  writer = false;
  if (nbReaders>0)
        canRead.signal();
  else canWrite.signal();
}
```

- Priority to signal receiver
- Priority to readers

Exercise
semaphore with monitor

```
monitor Semaphore () {
   int count;
   Condition positive;

procedure init(int v0) {
   count = v0;
}

procedure P() {
   count--;
   if (count < 0) positive.wait();
}

procedure V() {
   count++;
   positive.signal();
}
```

Resources you can read

- Operating System Concepts, 10th Edition, Abraham Silberschatz, Peter B. Galvin, Greg Gagne
  - http://os-book.com/
  - Chapters 6, 7
- Modern Operating Systems, Andrew Tanenbaum
  - http://www.cs.vu.nl/~ast/books/mos2/
  - Chapter 2 (2.3 & 2.4)

Memory management

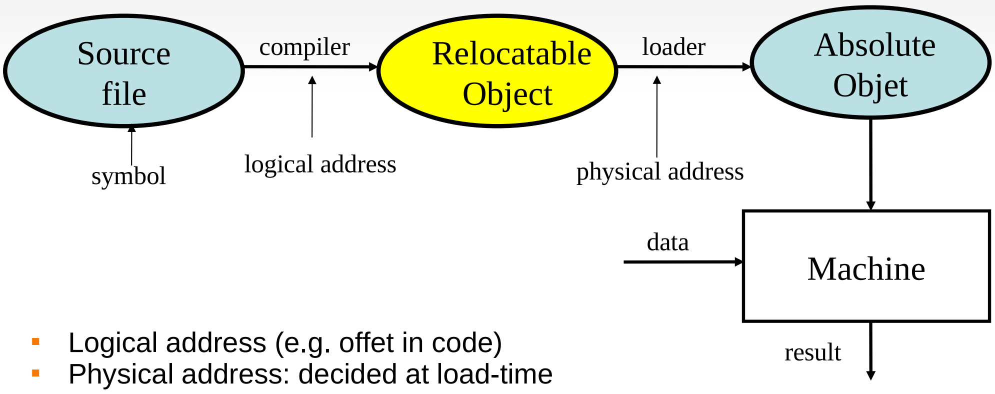
Daniel Hagimont

https://www.google.fr/search?q=daniel+hagimont+home+page

Introduction

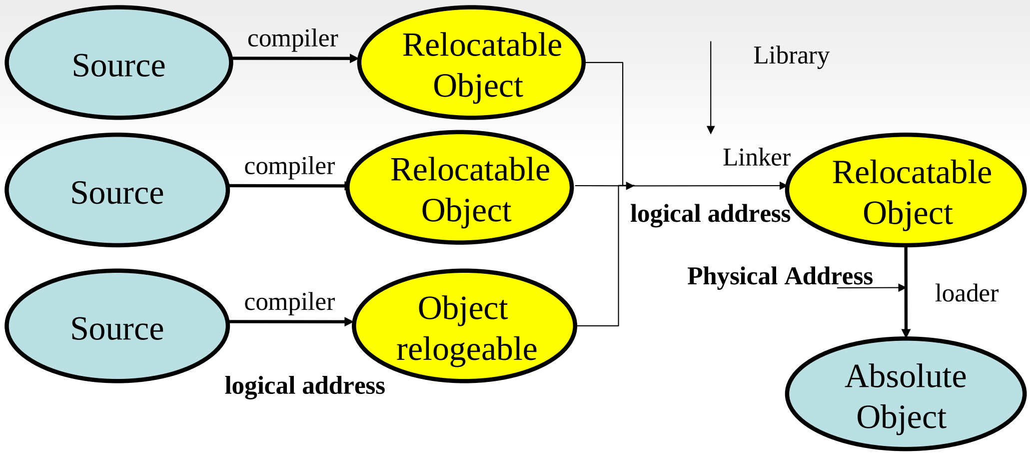
- Memory is a ressource required by all processes
  - Every program needs to be loaded in memory to be running
- Problems
  - Address translation
    - Symbol → Logical address → physical address
  - Memory allocation and exhaustion
  - Memory sharing
  - Memory protection

Life cycle of a single program



- Logical address (e.g. offet in code)
- Physical address: decided at load-time

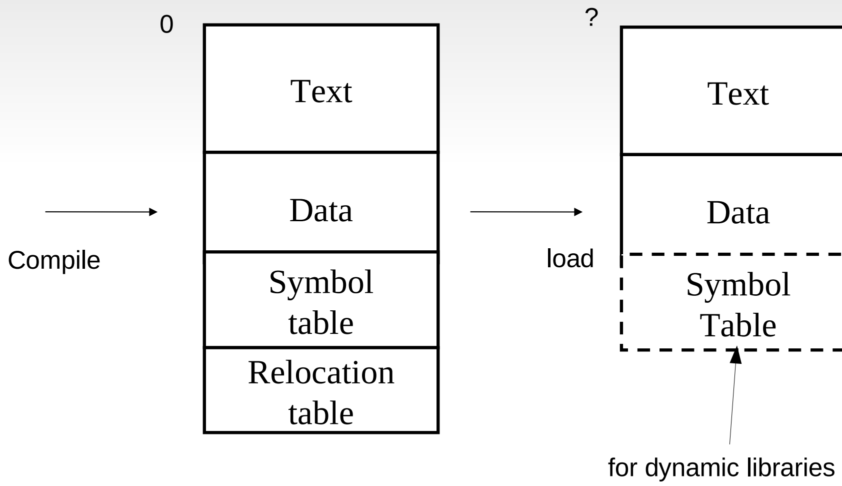
Lifecycle of a program assembled from multiple parts



Load-time translation

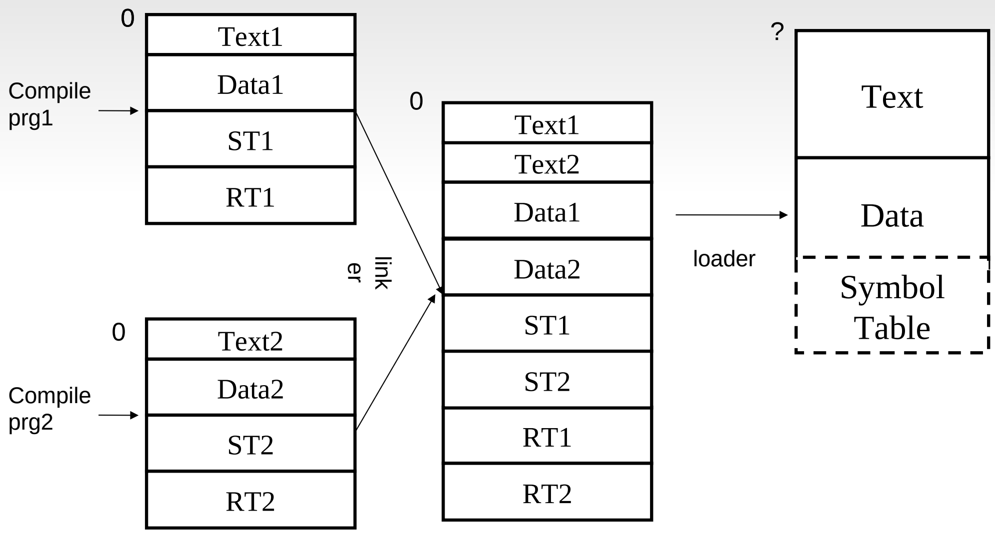
- Translation between logical and physical adresses
  - Determine where process will reside in memory
  - Translate all references within program
  - Established once for all
- Monoprogramming
  - One program in memory
  - Easy (could even be done before load-time)
- Multiprogramming
  - N programs in memory
  - Compiler and linker do not know the implantation of processes in memory
  - Need to track op-codes that must be updated at load-time

Simple program binary structure



6

Complex program binary structure



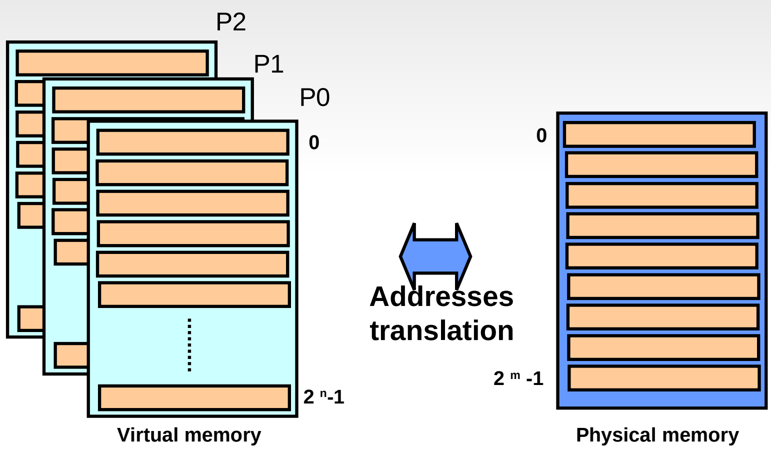
Load-time translation summary

- Remaining problems
  - How to enforce protection ?
  - How to move program once in memory ?
  - What if no contiguous free region fits program size ?
  - Can we separate linking from memory management problems ?

Virtual memory

- Separate linking problem from memory management
- Give each program its own virtual address space
  - Linker works on virtual addresses
  - Virtual address translation done at runtime
    - Relocate each load/store to its physical address
    - Require specific hardware (MMU)

Virtual memory



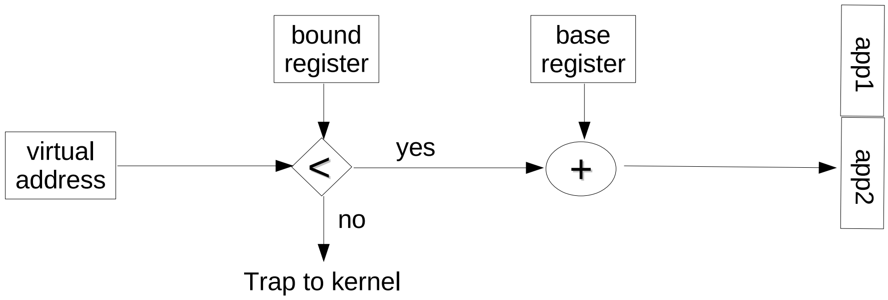
Ideally we want to enable n > m and non contiguous allocation

Virtual memory expected benefits

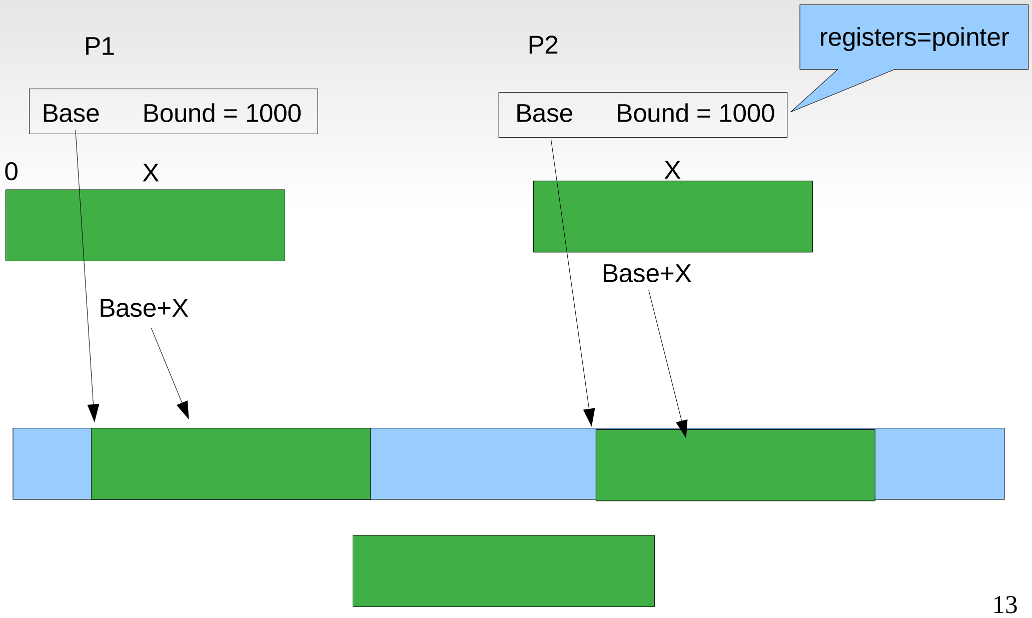
- Programs can be relocated while running
    - Ease swap in/swap out
- Enforce protection
    - Prevent one app from messing with another's memory
- Programs can see more memory than exists
    - Most of a process's memory will be idle
    - Write idle part to disk until needed (swap)

1st idea : Base + bound registers

- Contiguous allocation of variable size
- Two special privileged registers: base and bound
- On each load/store:
  - Check 0 <= virtual address < bound, else trap to kernel
  - Physical address = virtual address (plus) base



1st idea : Base + bound registers



13

Base + bounds register

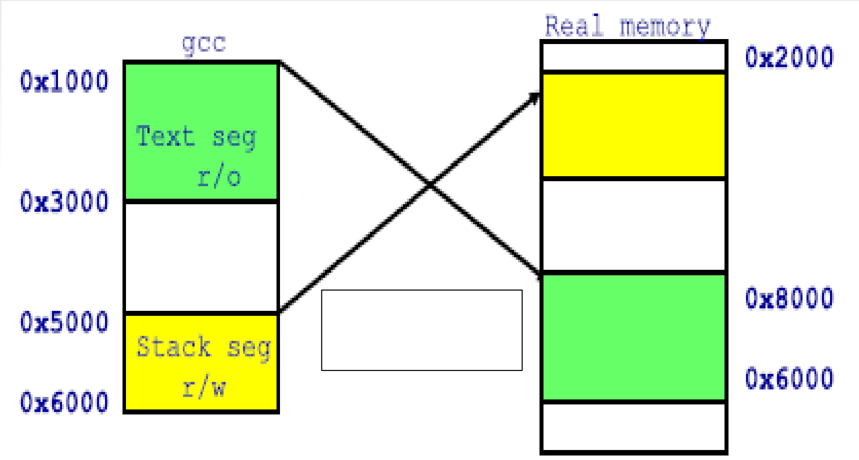
- Moving a process in memory
  - Change base register
- Context switch
  - OS must re-load base and bound register
- Advantages
  - Cheap in terms of hardware: only two registers
  - Cheap in terms of cycles: do add and compare in parallel
- Disadvantages
  - Still contiguous allocation
  - Growing a process is expensive or impossible
  - Hard to share code or data

Segmentation

- Non contiguous allocation
  - Split a program in different non contiguous segments of variable size
- Let processes have many base/bound registers
  - Address space built from many segments
  - Can share/protect memory at segment granularity
- Must specify segment as part of virtual address

15

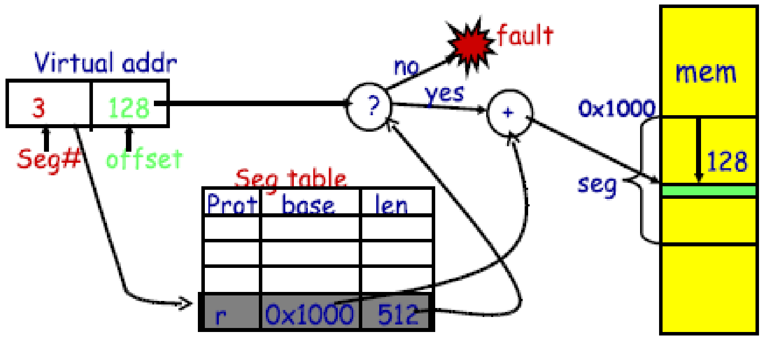
Segmentation



16

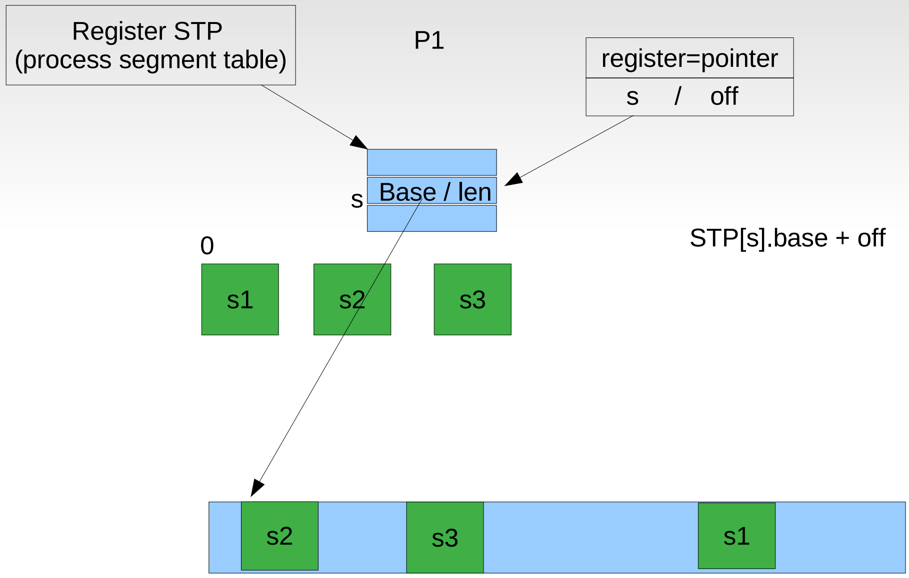
Segmentation mechanism

- Each process has a segment table
  - Each virtual address indicates a segment and offset:
    - Top bits of addr select segment, low bits select offset



17

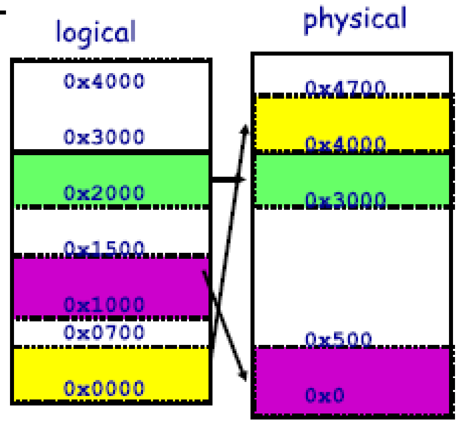
Segmentation



Segmentation example

- 4-bit segment number (1st digit), 12 bit offset (last 3)
  - Where is 0x0240? 0x1108? 0x265c? 0x3002? 0x1600?

Seg	base	bounds	rw
0	0x4000	0x6ff	10
1	0x0000	0x4ff	11
2	0x3000	0xfff	11
3			00

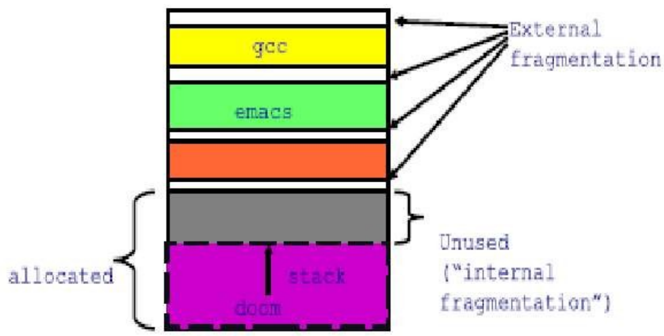


Segmentation tradeoffs

- Advantages
  - Multiple segments per process
  - Allows sharing
- Disadvantages
  - N byte segment needs N contiguous bytes of physical memory
  - Fragmentation (need moving segments)

Remember fragmentation problem

- Fragmentation => inability to use free memory
- Overtime:
  - Variable-size pieces = many small holes (external fragmentation)

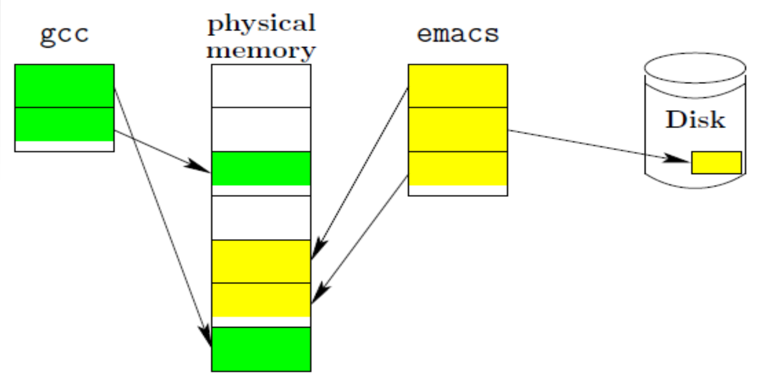


21

Paging

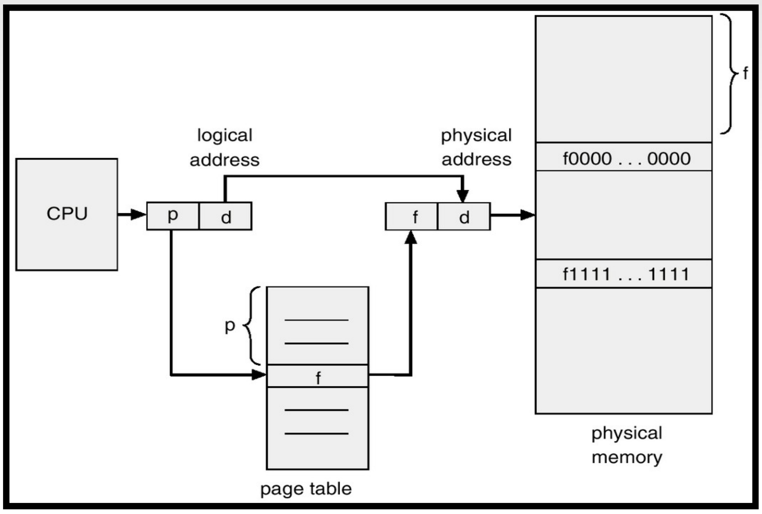
- Virtual memory is divided into small pages
  - Pages are fixed size
  - A page is contiguous
- Map virtual pages to physical block
  - Non contiguous allocation of blocks
  - Each process has a separate mapping but can share the same physical block
  - MMU
- OS gains control on certain operations
  - Non allocated pages trap to OS on access
  - Read only pages trap to OS on write
  - OS can change the mapping

Paging



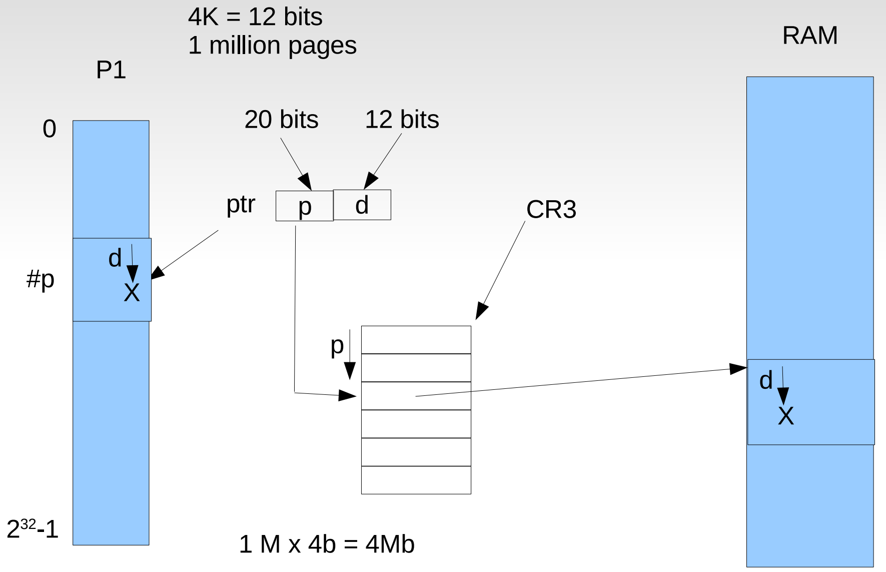
- Page table
  - Global or per process

Virtual address translation



24

Virtual address translation



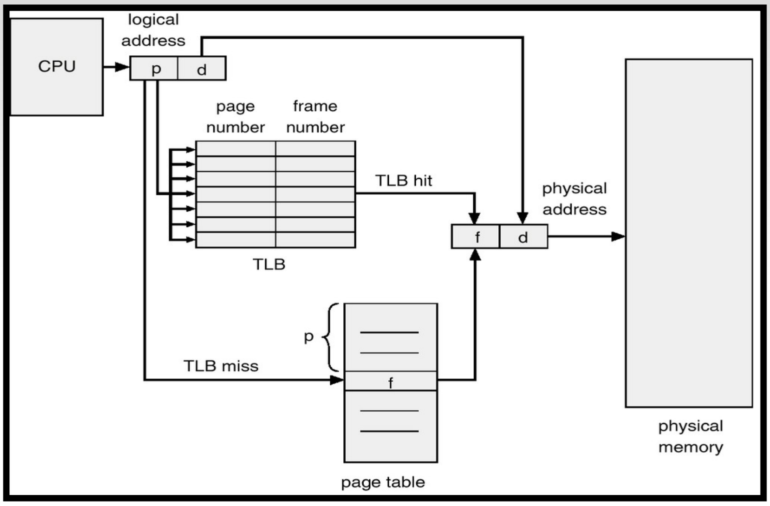
Access : CR3[p]+d

Problem : translation speed

- Require extra memory references on each load/store
  - Cache recently used translations
  - Locality principle
    - High probability that the next required address is close
- Translation Lookaside Buffer (TLB)
  - Fast (small) associative memory which can perform a parallel search
  - Typical TLB
    - Hit time : 1 clock cycle
    - Miss rate 1%
  - TLB management : hardware or software

26

TLB



- What to do when switch address space ?
  - Flush the TLB
  - Tag each entry with the process's id
- Update TLB on page fault (add/remove TLB entries)

Problem : page table size

32 : 10 (1k) 10 (1K) 10 (1k) 2 (4)
4Gb = 2^{32}

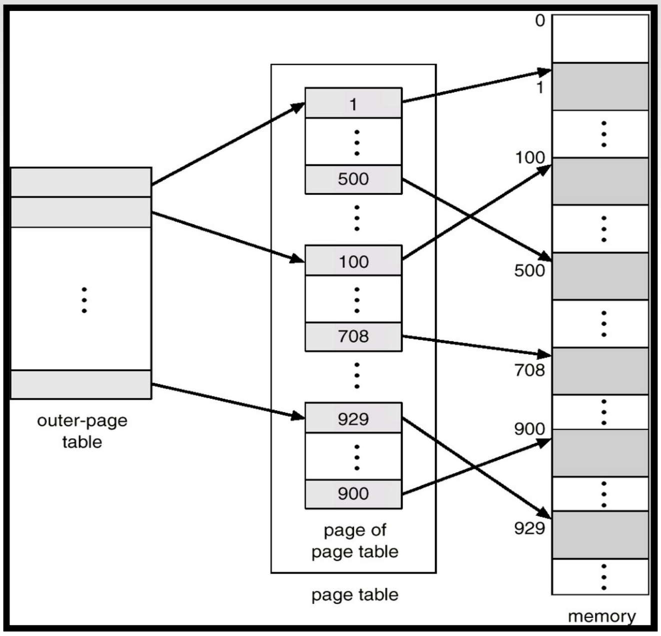
- Flat page tables are huge
- Example
  - 4GB of virtual memory (32 bits address)
  - 4KB pages
  - 20 bits page number, 12 bits offset
  - 4MB page table size :<
    - (1 million entries : 2^{10} x 2^{10})
    - PT size: 1K x 1K x 4 = 10^3 x 10^3 x 4 = 4 10^6
- 64 bit address space ?
  - Page table size ?
    - 52 bits for page number : 4.000 millions of millions
    - PT size:1K x 1K x 1K x 1K x 1K x 4 x 8 = 32 10^{15}

28

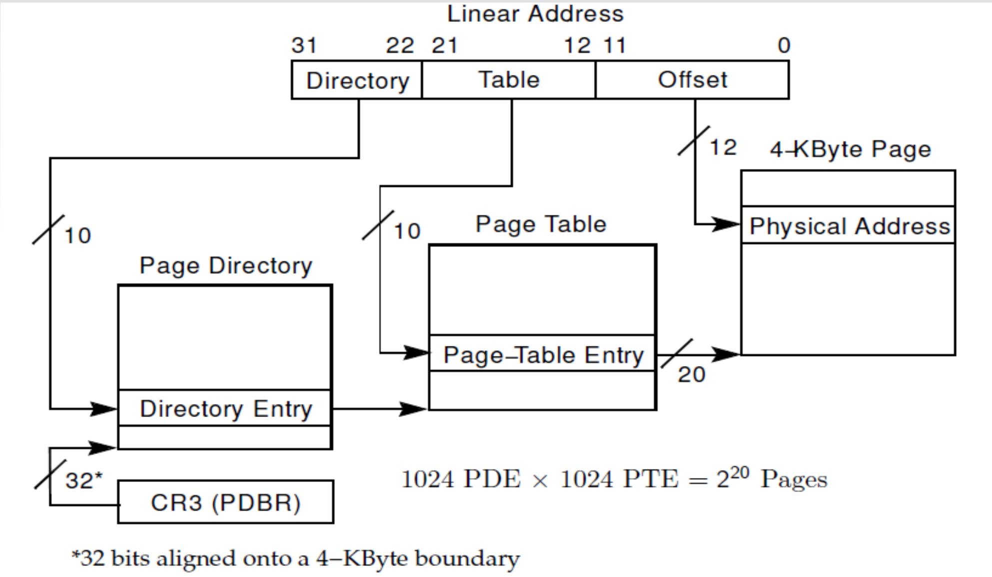
Multi-level page tables

- Reduce the size of page tables in memory
- Structured page tables in 2 or more levels
  - All the page tables are not present in memory all the time
  - Some page tables are stored on disk and fetched if necessary
- Based on a on-demand paging mechanism

Example: two level pages



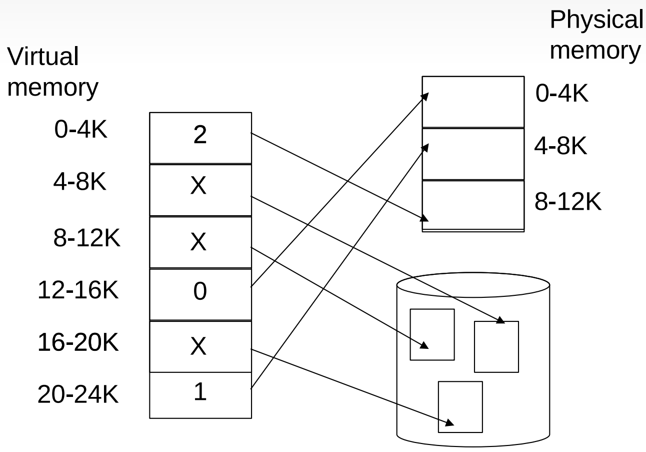
Example: two level pages



31

On demand paging

- Virtual memory > physical memory
  - Some pages are not present in memory (X)
  - Stored on disk

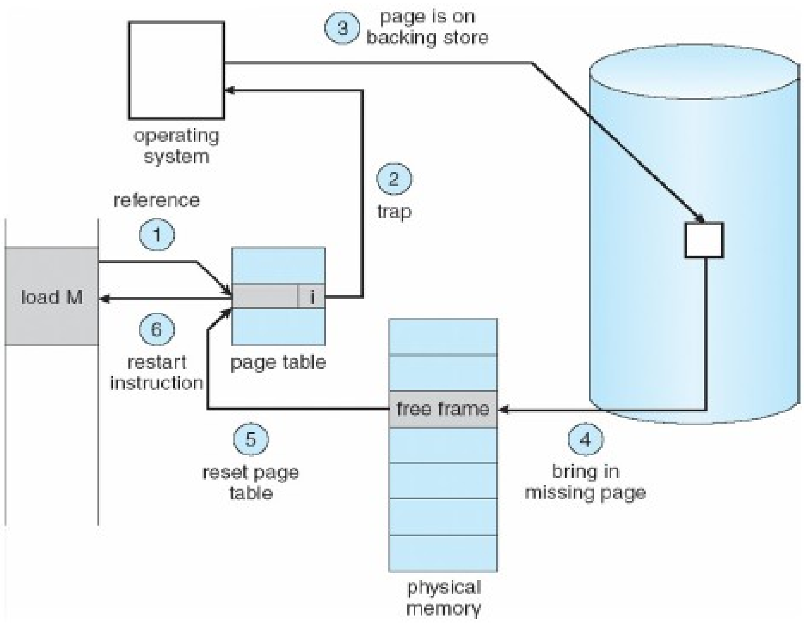


32

Page fault

- Access to an absent page
  - Presence bit
  - Page fault (Trap to OS)
- Page fault management
  - Find a free physical frame
    - If there is a free frame; use it
    - Else, select a page to replace (to free a frame)
    - Save the replaced page on disk if necessary (dirty page)
  - Load the page from disk in the physical frame
  - Update page table
  - Restart instruction
- Require a presence bit, a dirty bit, a disk @ in the page table
- Different page replacement algorithms

On demand paging



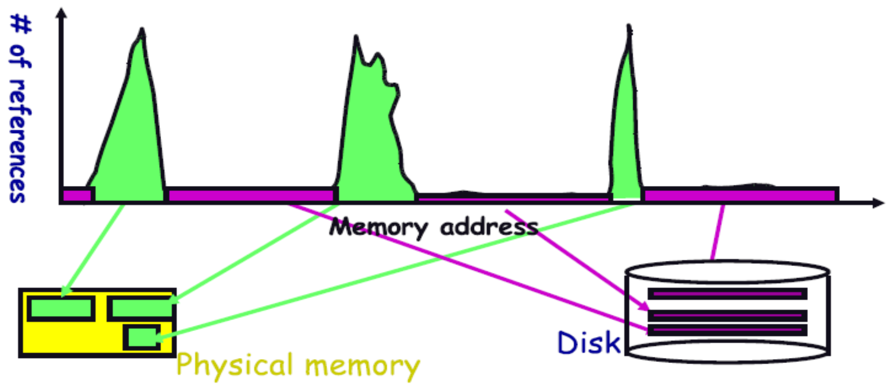
34

Page replacement algorithms

- Working set model
- Algorithms
  - Optimal
  - FIFO
  - Second chance
  - LRU

Working set model

- Disk much much slower than memory (RAM)
  - Goal: run at memory (not disk) speed
- 90/10 rule: 10% of memory gets 90% of memory refs
  - So, keep that 10% in real memory, the other 90% on disk



36

Optimal page replacement

- What is optimal (if you knew the future)?
  - Replace pages that will not be used for longest period of time
- Example
  - Reference string : 0,1,2,3,0,1,4,0,1,2,3,4,1,2
  - 4 physicals frames:

	0	0	0	0	0	3
		1	1	1	1	1
			2	2	2	2
				3	4	4

6 pages faults

FiFo

- Evict oldest page in system
- Example
  - Reference string : 0,1,2,3,0,1,4,0,1,2,3,4,1,2
  - 4 physicals frames:

10 page faults

	0	0	0	0	4	4	4	4	3	3
		1	1	1	1	0	0	0	0	4
			2	2	2	2	1	1	1	1
				3	3	3	3	2	2	2

- Implementation: just a list (updated on page fault)

LRU page replacement

- Approximate optimal with least recently used
  - Because past often predicts the future
- Example
  - Reference string : 0,1,2,3,0,1,4,0,1,2,3,4,1,2
  - 4 physicals frames:

	0	0	0	0	0	0	0	4
		1	1	1	1	1	1	1
			2	2	4	4	3	3
				3	3	2	2	2

8 page faults

39

LRU implementation

- Expensive
  - Need specific hardware
    - Track access without page fault
- Approximate LRU
  - The aging algorithm
    - Add a counter for each page (the date)
    - On a page access, all page counters are shifted right, inject 1 for the accessed page, else 0
    - On a page replacement, remove the page with the lowest counter

Aging : example

Accessed page	Date Page0	Date Page1	Date Page2	Order pages /date
	000	000	000	
Page 0	100	000	000	P0,P1=P2
Page 1	010	100	000	P1,P0,P2
Page 2	001	010	100	P2,P1,P0
Page 1	000	101	010	P1,P2,P0

P0 is the oldest

Second chance

- Simple FIFO modification
  - Use an access bit R for each page
    - Set to 1 when page is referenced
    - Periodically reset by hardware
  - Inspect the R bit of the oldest page (of the FIFO list)
    - If 0 : replace the page
    - If 1 : clear the bit, put the page at the end of the list, and repeat
- Appromixation of LRU
  - don’t have to parse all pages

Page buffering

- Naïve paging
  - Page replacement : 2 disk IO per page fault
- Reduce the IO on the critical path
  - Keep a pool of free frames
    - Fetch the page in an already free page
    - Swap out a page in background

Paging

- Separate linking from memory concern
- Simplifies allocation, free and swap
- Eliminate external fragmentation
- May leverage internal fragmentation

Resources you can read

- http://en.wikipedia.org/wiki/Page_table
  - Wikipedia can always be useful
- Operating System Concepts, 10th Edition, Abraham Silberschatz, Peter B. Galvin, Greg Gagne
  - http://os-book.com/
  - Chapters 9 & 10
- Modern Operating Systems, Andrew Tanenbaum
  - http://www.cs.vu.nl/~ast/books/mos2/
  - Chapter 4