# INF841
# Operating Systems Module

M. Sc. in Computer Science

Petr Kuznetsov

petr.kuznetsov@telecom-paristech.fr

*Coauthored with*

*I. Demeure, B. Dupouy, and L. Pautet*

# Licence de droits d'usage

**Cadre privé } sans modification**

***Par le téléchargement ou la consultation de ce document, l'utilisateur accepte la licence d'utilisation qui y est attachée, telle que détaillée dans les dispositions suivantes, et s'engage à la respecter intégralement.***





TELECOM ParisTech

# Operating Systems Module

1. Introduction
2. Processes
3. Basics of System Programming
4. File Management
5. Memory management
6. Bibliography and online resources

# Literature

**Books**:

- Siberschatz, Galvin, Gagne. Operating Systems Concepts, 7th edition, Wiley, 2005
- Tannenbaum. *Modern Operating Systems*, 2nd edition, Prentice-Hall, 2001
- Bach, Maurice J. The Design Of The Unix Operating System. Prentice Hall, Software Series, 1986
- Nutt. *Operating Systems: A Modern Perspective*, 2nd edition, Addison-Wesley (2002)

**Links**:

- University of Surrey. Unix for beginners: http://www.infres.enst.fr/~demeure/SiteCSIC/UNIX TUTORIAL/index.html (Tutorials 1-6)
- R. H. & A. C. Arpaci-Dusseau, U Wisc., Operating Systems: Three Easy Pieces, 2013. http://pages.cs.wisc.edu/~remzi/OSTEP/

# Operating Systems Module

## 1. Introduction

# What is an OS? What is it for?

- Top–down view: **extended machine**
  - Machine-language level is painful to use
  - OS provides a user-friendly interface that hiding the complexity of the hardware

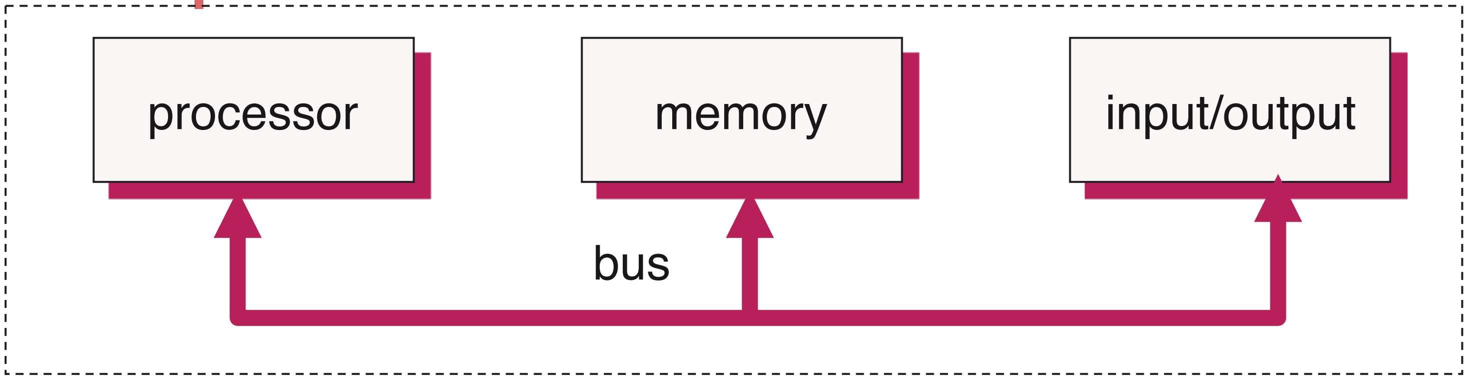
- Bottom-up view: **resource manager**
  - Computer hardware is a complex system run by multiple users
  - OS arbitrates the shared resources

- Simplistic view
  - OS is the one program running at all times (the **kernel**), all the rest are system and application programs

# History of OS

- 1860-1870: Analytical engine. Charles Babbage, Ada Lovelace
  - First programmable machine (purely mechanical, reproduced in 1991)
- 1940-1955: First generation calculation engines. K. Zuse, H. Aiken, J. von Neumann
  - Mechanical relays, vacuum tubes, programmed with plugboards (no OS)
  - Cycle times in seconds
- 1955-1965: Transistors and batch systems
  - Mainframes, programmed by punchcards (FORTRAN, assembly) , later “batched” on a tape
  - Fortran Monitor System (FMS)
- 1965-1980: Integrated Circuits and Multiprogramming
  - IBM System/360 – first multi-purpose machines
  - OS/360 running on all models, multiprogramming, spooling
- 1980-now: Personal computers
  - LSI (Large Scale Integration), microcomputers, general-purpose 8,16,32-bit processors, GUI (Graphical User Interface), distributed OS

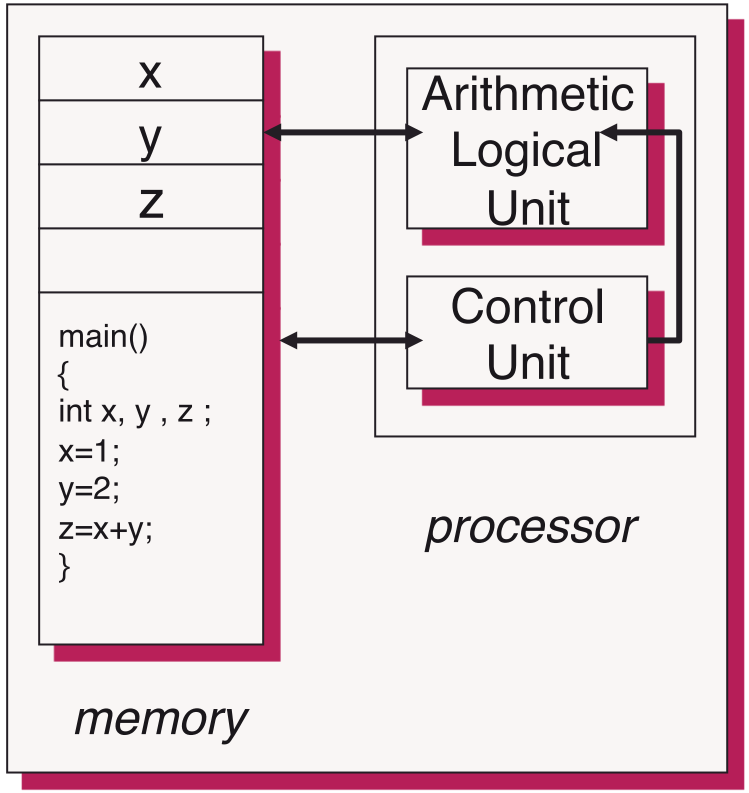
# Computer hardware architecture



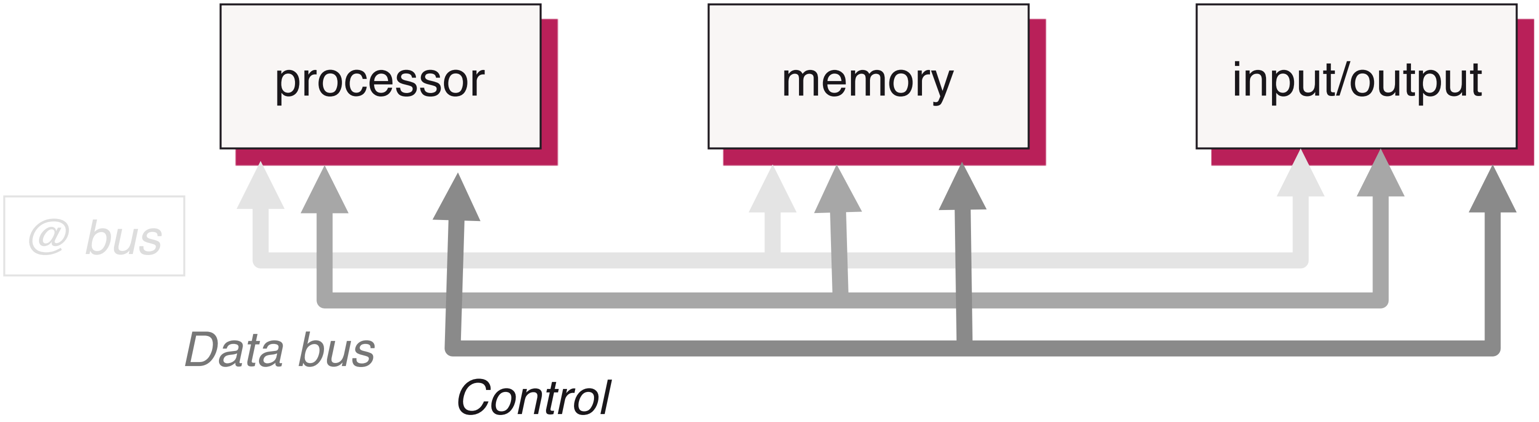

- A computer hardware architecture consists of 4 basic building elements:
  - processor, where program instructions are executed
  - memory where code and data are stored
  - input/output units that establish the interface between the various peripherals (screen, keyboard, mouse, disks, etc.) and the computer itself
- Von Neumann architecture (1945)
  - Both the code and the data are stored in (RAM) memory
  - Most of today's architectures follow the Von Neumann model

# Processor

- Arithmetic Logical Unit (ALU)
  - in which basic instructions are executed
- Control Unit (CU)
  - fetches instructions from memory,
  - controls data fetching & storage,
  - controls the ALU operation
- A number of registers
  - small memories on the processor;
  - eg. Instruction register, program counter.
- Instructions are executed following a fetch / decode / execute / store model.



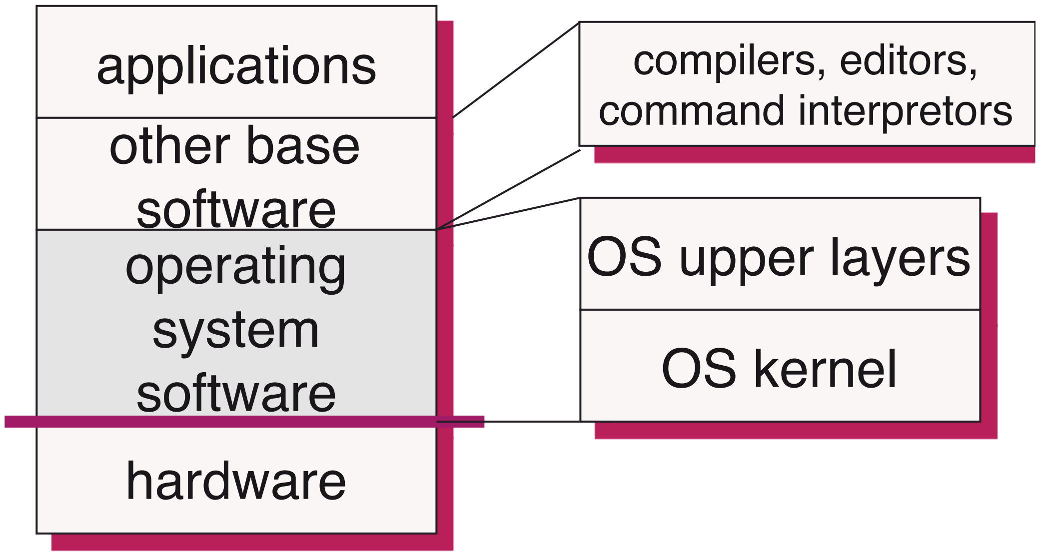
# Bus and memory access




- Memory access (read/write):
  - Put address on address bus
  - Place read/write order on control bus
  - Wait if writing (latency)
  - Read/write data from/to data bus

# Operating system goals

- Build a virtual machine on top of the hardware
  - Abstracts its capabilities
  - Hardware-independent
  - Provides a nice user interface ... if possible




- Manage resources:
  - processor, memory, storage, peripherals, time, communications, etc.

# Operating system characteristics

- Degree of parallelism, single or multiple:
  - tasks,
  - users (multi-user => multi-task),
  - processors,
  - machines.
- Operating mode
  - Time sharing
    - Several simultaneous users
    - Each user has its virtual machine
    - =>Resource sharing
    - =>Time quantum
  - Real time
    - Meet deadlines
    - Interact with environment
    - => resource preallocation
    - => fault tolerance

# Examples

- Mac OS X, Windows 8, iOS, Android …

- Unix (Solaris, Linux, FreeBSD) will be used:
  - Developed by Thompson and Ritchie, Bell Labs, 1969-70
  - Derived from CTSS and MULTICS (MIT)
    - Main characteristics
    - Kernel and user modes
    - 90% Kernel written in C (portability)
    - Many communication tools (good for networks)
    - All resources seen as files, therefore uniform I/O style
    - Powerful command line interpreter (shell), with pipeline functionality

# Interacting with the OS

- Through the command interpretor
  - Eg. Under Unix: ls, ps, exec
  - Interactively or through command programs (scripts).

- From a program,
  - Through a library of system calls
    - E. g.: read, write, sendto, fork, exit

# Operating Systems Module

## 2. Processes

# Processes: virtualization of CPU

# Operating Systems Module

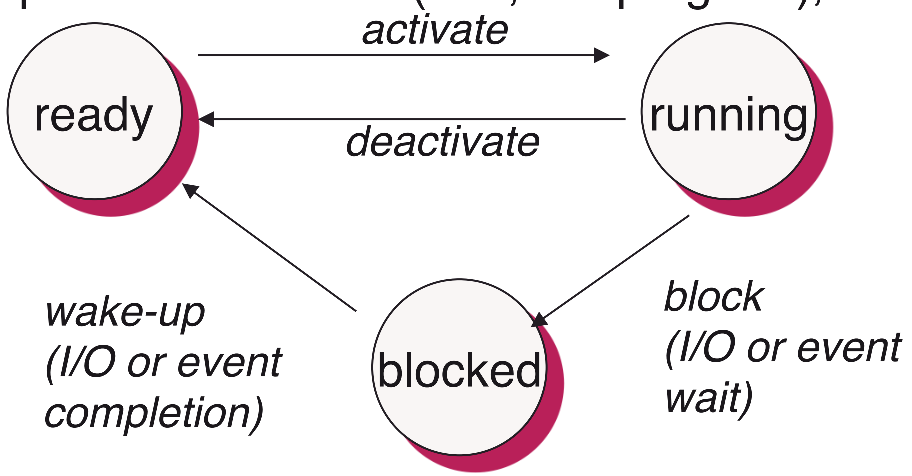
## 2-Processes
## 2.1-Definitions and concepts

# Program versus process

- A program is a set of instructions (a static object)
- A process is a running program and its context (a dynamic object)
- A program context includes the information relative to the process execution:
  - Registers state: program counter, stack pointer, registers describing the process virtual memory space.
  - Resources accessed by the process (e.g., file access rights, open files).
  - Other (e.g., clock value)
- The context must be:
  - saved when a process is deactivated or blocked and
  - restored when a process is activated.
- A process is uniquely identified by a Process IDentifier (PID)

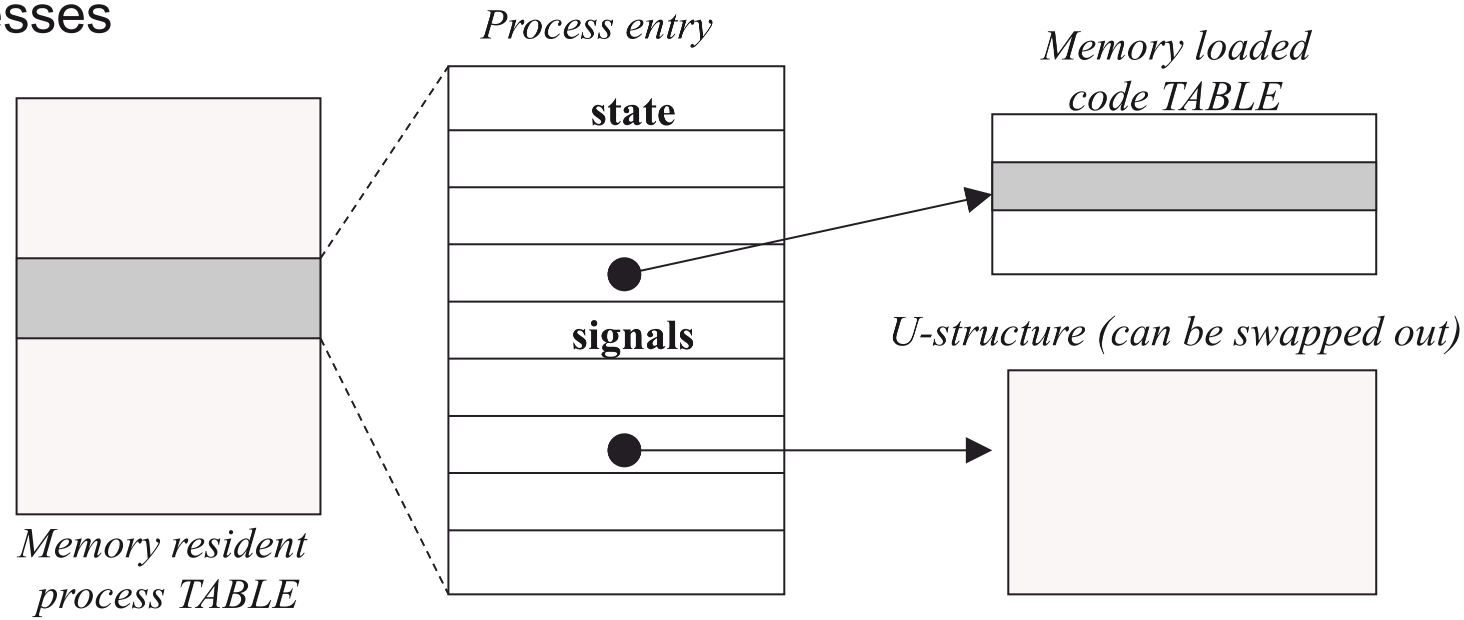
# Process state

- In a multiprocessing system, several processes can be running simultaneously: sharing access to the processor.
- A process may go through several states:
  - **Active** or **running**: the process is the one currently using the processor
  - **Ready** or **eligible**: the process could use the processor if it were ready and if it were its turn to be run.
  - **Blocked** or **waiting**: the process is waiting for a resource (e. g., on-going I/O).
  - Other (e. g., **zombie** or **swapped**)
- Transitions: activate, deactivate, block, wake-up
- Other operations on processes: create (fork, run program), destroy (kill, exit)



# Process table and process space under Unix

- The OS maintains a table containing information relative to ongoing processes




- UNIX allocates 3 memory areas for each process
  - Code area
  - Data area
  - Stack area

# Process creation with Fork under Unix

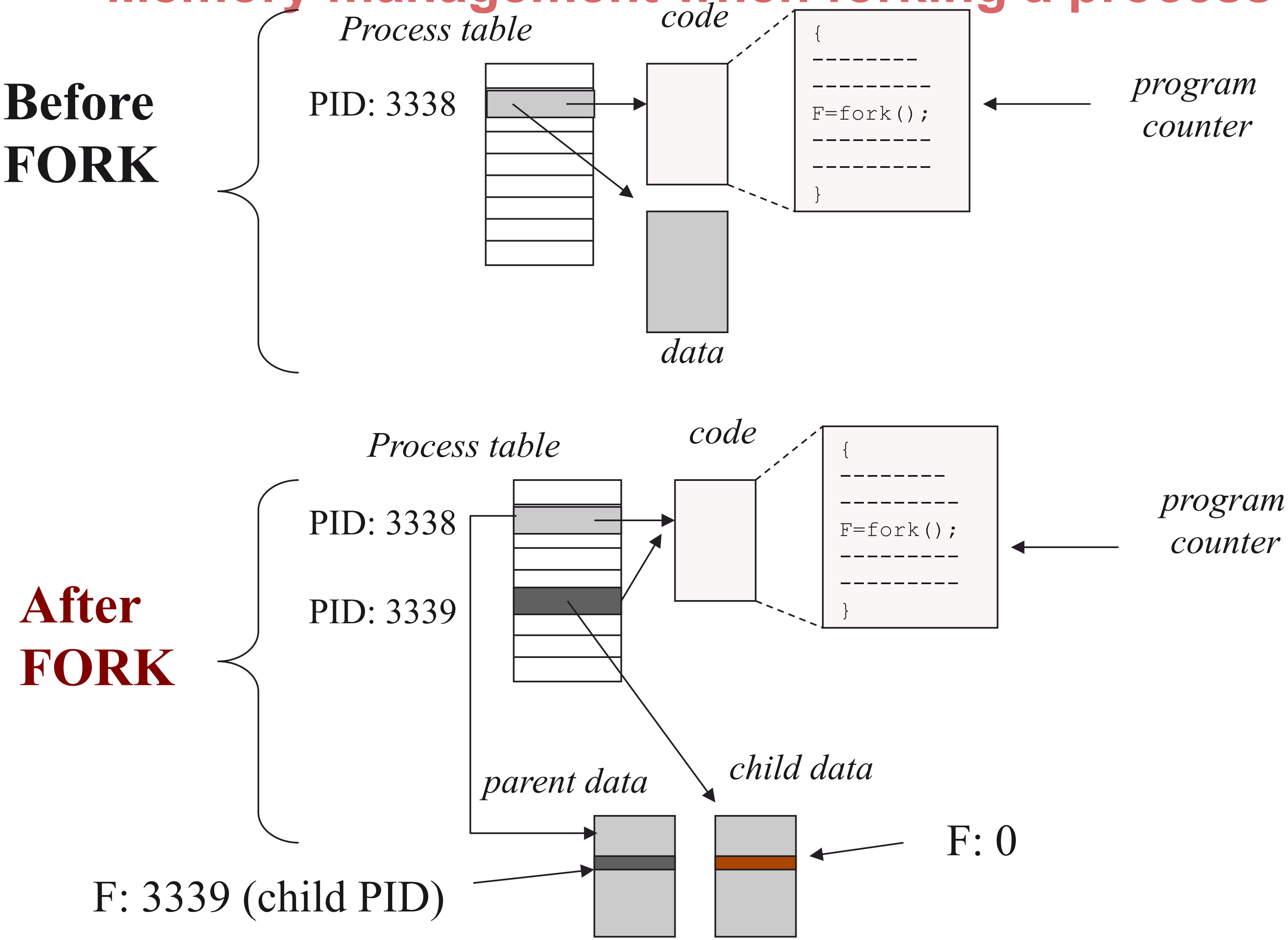
```
int main ( void)
{
    int f;
   •••;
   f = fork ();
   if ( f ==  -1) {
         printf ( "Error : the process cannot be created\n" )
         exit (1);
   }
   if ( f ==  0) {
         printf ("Hi ! I am the child process\n" )
         •••;
   }
   if ( f !=  0) {
         printf ("I am the father process\n" )
         •••;
}
```

- Fork creates a «clone» of the calling process.
- The calling process is called **parent** and the callee is called **child**.
- The parent and the child have different process numbers.
- The values returned to the caller and to the callee are not the same.

# Example of process creation under Unix

```
$ cat exo1.c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main (void)
{
  int f;
  f = fork();
  printf ("value returned by fork: %d\n", (int)f);
  printf ("I am process number %d\n", (int)getpid());
}
$ gcc exo1.c -o exo1
$ exo1
value returned by Fork: 0
I am process number 3899
value returned by Fork: 3899
I am process number 3898
$
```

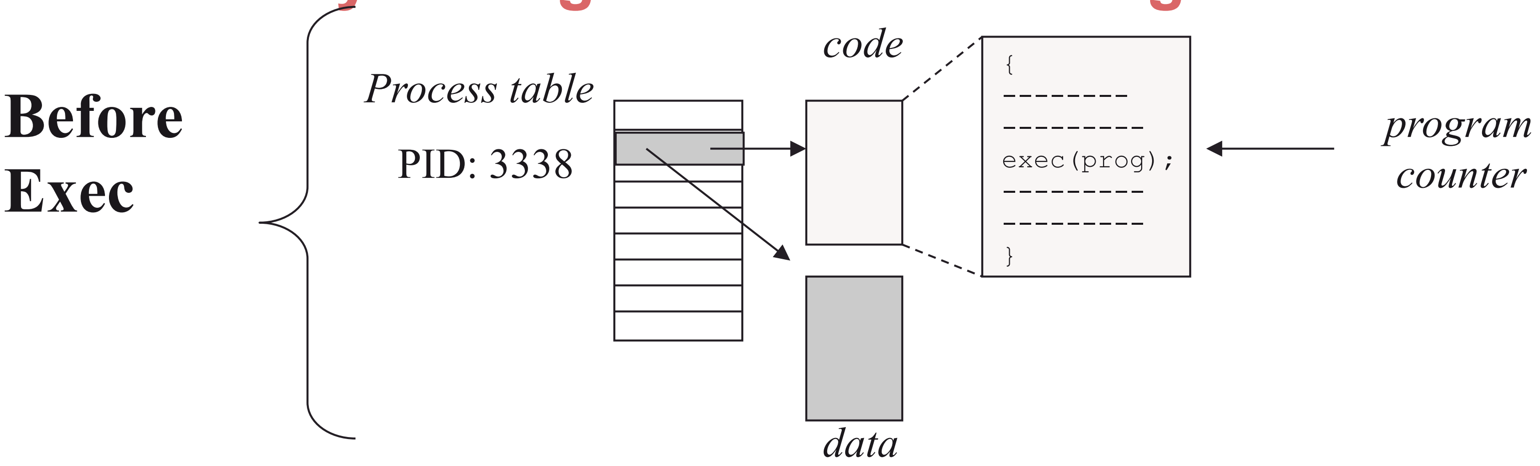
# Memory management when forking a process

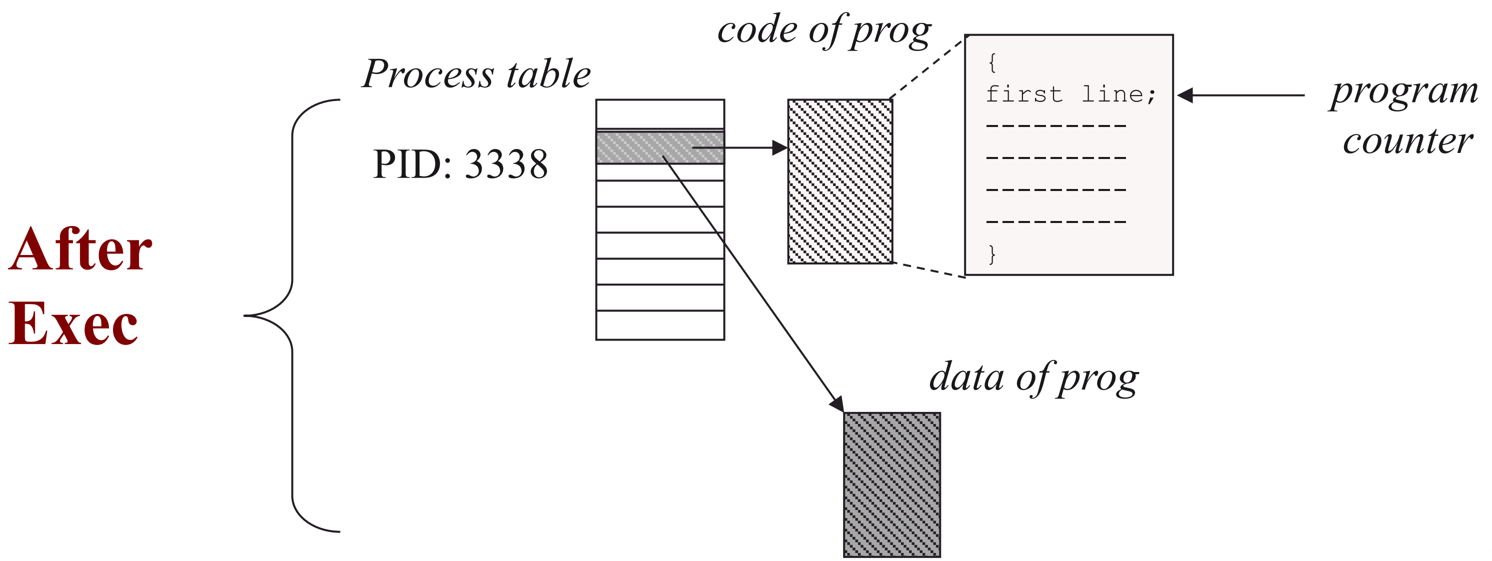


# Fork-exec mechanism

- exec(char *path, char *arg0, ... char * argn, char * /* NULL */) :
  - exec is called by any process that wants its current code replaced by another one specified by path.
  - fork creates a clone child process that executes the father's code.
  - the child process may specify another code to be executed using **exec**

# Memory management when calling exec



# Fork-exec example

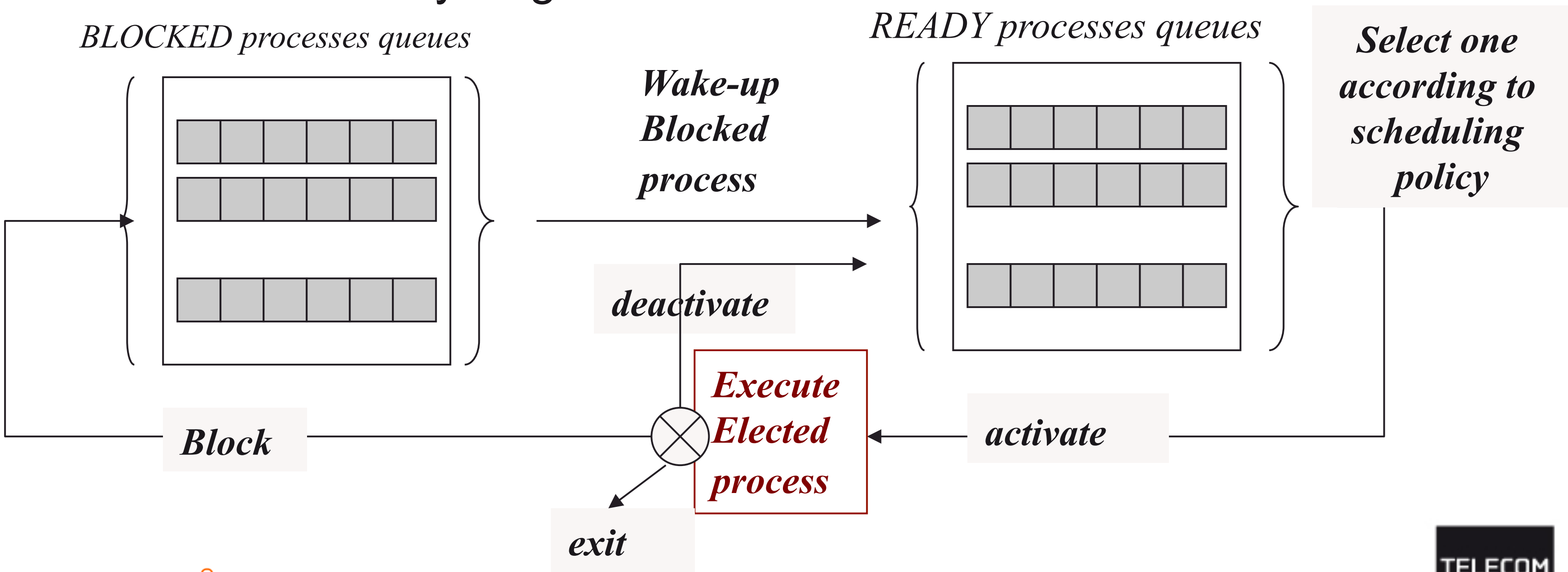
```
int main (int argc, char *argv[])
{•••
  if (argc != 2)
    {printf(" Usage : %s you forgot the argument (name file for exec)! \n",
argv[0]); exit(1);}
  printf (" I am %d, I will fork \n",(int)getpid());
  f=fork();
  switch (f)
    {
    case  0 :
      printf ("Hi ! I am the child %d\n",(int)getpid());
      printf ("%d : Code replaced by %s\n",(int)getpid(), argv[1]);
      execl(argv[1],(char *)0);
      printf (" %d : Error in exec \n", (int) getpid()); exit (2);
    case   -1 :
      printf (" Fork failed "); exit (3) ;
    default : /* parent waits for end of child*/
      printf ("Parent %d waits\n ",(int)getpid());
      child=wait (&State);
      printf ("Child was : %d ", child );
      printf (".. Its state was :%0x (hexa) \n",State);exit(0);}
}
$ ./fexec exo1
I am 846, I will fork
Parent number 846 waits
Hi ! I am the child 847
847 : Code replaced by exo1
 Value returned by fork: 848
I am process number 847
 Value returned by fork: 0
I am process number 848
 Child was : 847 .. Its state was :ff00 (hexa)
```

# Operating Systems Module

## 2. Processes
## 2.2. Processor scheduling

# Processor scheduling

- Several processes may compete for accessing CPU
- The OS **scheduler** decides on the order in which processes will access the processor.
- The scheduler implements a scheduling policy:
  - With or without priorities
  - With or without preemption
  - With or without recycling







TELECOM ParisTech

# Scheduling policies

- What is «the best» scheduling policy ?
  - It depends on what is to be achieved

- Measures
  - Throughput: how many tasks can be executed per time unit.
  - **Turnaround time**: time between task submitted and final result returned (includes: time waiting in queues, memory load time, execution time, I/O time)
  - Waiting time: time spent in the ready queue
  - **Response time**: time between task submitted and first scheduled

- We focus primarily on turnaround time and response since we are interested fairness and interactivity in time-sharing systems.

- If we were in a real-time system a measure would be the number of missed deadlines.

# Examples of scheduling policies

- First Come first Served
- Shortest Job First
- Shortest Time-to-Completion Job First
- Priority
- Round-robin
- Combination, e. g., UNIX scheduling

# First Come First Served

- Processes are executed in the order in which they arrive in the ready queue:
  - no preemption, no recycling, no priority.

- Example:
  - Assume 3 processes P1, P2, P3, arrival order P1, P2, P3 all at time T0
  - Execution times: P1=12 time units, P2=2 time units, P3 =2 time units
  - Arrival order P1, P2, P3 all at time T0:
    - Average turnaround time = (12+14+16) / 3 = 14
    - Gantt chart

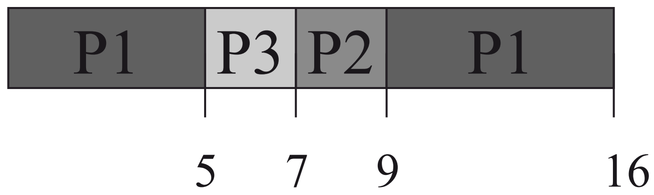
# Shortest Job first

- Execute processes with shortest execution time first
  - No preemption, no priority, no recycling
  - Example: arrival order P1, P2, P3 all at time T0:
    - Average turnaround time = (2+4+16) / 3 = 7.33


- Problem 1: how do we evaluate execution times ?
  - Prediction
  - Ask user to give an upper bound when the job is submitted (as before)
- Problem 2 = Starvation:
  - If short processes keep coming longer processes will never get access to the processor
  - Ageing technique: when making scheduling decision, take into account time already spent waiting.

# Shortest Time-to-Completion First (Preemtion)

- Preempt if less demanding jobs arrive
  - Preemption, no recycling, no priority.
- Example:
  - Assume 3 processes P1, P2, P3, arrival order P1 arrives at time 0,  P2 and P3 arrive at time 5
  - Execution times: P1=12 time units, P2=2 time units, P3 =2 time units




- Average turnaround time = (16+2+4) / 3 = 7.33

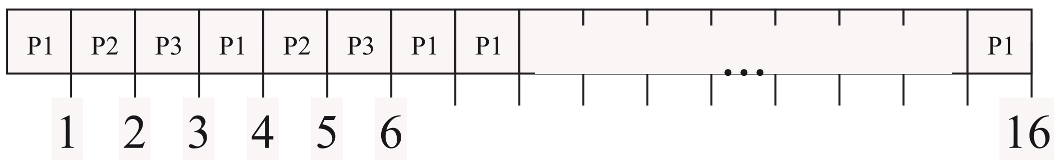
# Priority

- A priority level is attached to each process.
  - Schedule the process with the highest priority first.
  - Priority, preemption, recycling
- Problem: starvation
  - if processes with high priority keep coming, processes with lower priority will never get access to the processor
- Solution: ageing
  - Periodically increase priority to take into account time spent in the ready queue.

# Round Robin

- In order to fairly accommodate several «time-sharing» processes

- Split time into time intervals called **quanta**, and cyclically allocate a time quantum to each process in the ready queue.
  - Preemption, recycling, no priority
  - Great for **response time**!
- Example: same as before, assuming the quantum is equal to one time unit
  - Average turnaround time : (5+6+16) / 3=9
  - Performance depends on quantum size



# Unix scheduling: priorities based on the past utilization

## Multi-level feedback queue (MLFQ)

- Combines round-robin, priority and ageing
  - Priority, preemption, **recycling** (woken-up treatment)

- Processes scheduled according to their priority (0-127)
  - 0 the highest, 127 the lowest
  - Processes with priorities 0-25 and not preemptible (real-time processes)
  - **Round-robin** for each priority level (greater than 25)

- Priority = Pbase+ f(CPU used, waiting time)
- After I/O or signal: Priority = g(I/O type)

# Unix scheduling: example

- A process starts with a base priority (Pbase=60) and maintains a **priority margin** C as a function of CPU consumption (fairness) and time in the ready queue (ageing):
  - Initially, C=CPU time already used by the process
  - Each **time quantum** C=C/2
  - Process priority P=Pbase+C+ nice -20
  - The **nice** parameter adjusts the base priority (application-specific)

- When a process wakes-up after an I/O (recycling)
  - It starts with a priority depending on the I/O event.
  - The current process is preempted if appropriate

# Multiple-processor scheduling

- Load-sharing possible, but also more complexity…

- Centralized solution: all scheduling decisions taken by the **master** processor, other processors only execute user code
- Symmetric multiprocessing (**SMP**), used by (almost) all modern OSs
  - **Shared** READY queues: **synchronization** required (we'll see details later) or **private** READY queues for each processor
  - **Processor affinity**: avoid migration between processors
  - **Load balancing**: distributed the load between private READY queues (push and pull strategies)

# Up-to-date scheduling mechanisms

- Up to Linux 2.5, linear (in the number of tasks) scheduler was used
  - 20% CPU time spent on scheduling!
- **O(1) scheduler** (Linux 2.5-2.6) and **Completely Fair Scheduler** (Linux 2.6+)
  - Two priority ranges**: real-time** (0-99, long quanta) and **nice** (100-140, short quanta)
  - Tasks are stored in a **red-black-tree** (**runqueue**), CPU consumption used as a key
  - Less CPU time used  - faster to find
  - Waiting (sleeping tasks) consume less CPU and thus prioritized (interactivity)
- **RMS** (Rate-Monotonic Scheduling) for **real-time systems**
  - Static-priority policy and preemption
  - Priority inverse to the **period** (processes are assumed periodic)

# Operating Systems Module

## 2. Processes
## 2.3. Signaling

## Signals

- A signal is a form of software interrupt that can originate from:
  - The kernel (divide by zero, illegal instruction, ...)
  - The keyboard (user hits Ctrl-C, Ctrl-D, …)
  - Call to the kill instruction, either through the shell, or by calling the kill system call (from a C program)

- Upon receiving a signal, the most common behaviour for a process is:
  - To exit
  - Possibly to produce a core file that contains an image of the process context when it received the signal, to allow for subsequent debugging

- Signal is a limited form of IPC (Inter Process Communication)

# List of standard signals (1/2)

In /usr/include/bits/signum.h

```
#define   SIGHUP      1         /* Hangup (POSIX).  */
#define   SIGINT      2         /* Interrupt (ANSI), Ctrl-C  */
#define   SIGQUIT     3         /* Quit (POSIX), Ctrl-D  */
#define   SIGILL      4         /* Illegal instruction (ANSI).  */
#define   SIGTRAP     5         /* Trace trap (POSIX).  */
#define   SIGABRT     6         /* Abort (ANSI).  */
#define   SIGIOT      6         /* IOT trap (4.2 BSD).  */
#define   SIGBUS      7         /* BUS error (4.2 BSD).  */
#define   SIGFPE      8         /* Floating-point exception (ANSI).  */
#define   SIGKILL     9         /* Kill, unblockable (POSIX).  */
#define   SIGUSR1     10        /* User-defined signal 1 (POSIX).  */
#define   SIGSEGV     11        /* Segmentation violation (ANSI).  */
#define   SIGUSR2     12        /* User-defined signal 2 (POSIX).  */
#define   SIGPIPE     13        /* Broken pipe (POSIX).  */
#define   SIGALRM     14        /* Alarm clock (POSIX).  */
#define   SIGTERM     15        /* Termination (ANSI).  */
#define   SIGSTKFLT   16        /* Stack fault.  */
#define   SIGCLD      SIGCHLD   /* Same as SIGCHLD (System V).  */
#define   SIGCHLD     17        /* Child status has changed (POSIX).  */
```

# Signals list (2/2)

In /usr/include/bits/signum.h

```
#define SIGCONT    18     /* Continue (POSIX).  */
#define SIGSTOP    19     /* Stop, unblockable (POSIX).  */
#define SIGTSTP    20     /* Keyboard stop (POSIX).  */
#define SIGTTIN    21     /* Background read from tty (POSIX).  */
#define SIGTTOU    22     /* Background write to tty (POSIX).  */
#define SIGURG     23     /* Urgent condition on socket (4.2 BSD).  */
#define SIGXCPU    24     /* CPU limit exceeded (4.2 BSD).  */
#define SIGXFSZ    25     /* File size limit exceeded (4.2 BSD).  */
#define SIGVTALRM  26     /* Virtual alarm clock (4.2 BSD).  */
#define SIGPROF    27     /* Profiling alarm clock (4.2 BSD).  */
#define SIGWINCH   28     /* Window size change (4.3 BSD, Sun).  */
#define SIGPOLL    SIGIO  /* Pollable event occurred (System V).  */
#define SIGIO      29     /* I/O now possible (4.2 BSD).  */
#define SIGPWR     30     /* Power failure restart (System V).  */
#define SIGSYS     31     /* Bad system call.  */
#define SIGUNUSED  31

#define _NSIG      65     /* Biggest signal number + 1 (including real-time signals).  */
```

# Operations on signals

- Upon receiving signal, possible behaviors are:
  - Ignore signals
  - Catch signal and execute treatment
  - Mask signals

- To send a signal:
  - From C: kill(process_nb, signal_nb)
  - From shell: kill –signal_nb process_nb

- To display all the supported signals: `kill –l`

# Receiving a signal

- A table including a bit position for each signal (NSIG), is attached to each process.
  - If the process has received no signal, all bits are set to 0.
  - Otherwise, the positions corresponding to a signal received are set to 1.
  - E. g., If signal number 15 was sent to the process, bit number 15 is set to 1.

- A process checks if signals are received when it runs in the **user mode**. The signal processing is deferred if the process:
  - Executes a system call (running in the **kernel mode**)
  - Is in **uninterrubptiple sleep** (e.g., waiting for disk I/O)

# What to do when receiving a signal

- 3 possibilities:
  - Ignore the signal:
    - by calling `signal(signal_nb, SIG_IGN)` in a program
  - Go back to the default treatment (one is defined for each signal) if some other behaviour was defined:
    - `signal(signal_nb, SIG_DFL)`
  - Define a specific treatment (function) to be executed:
    - `signal(signal_nb, function)`
- `signal()` **does not** send a signal, only defines the treatment when receiving it
  - In the **U-structure** part of the process entry, the system maintains a table in which it stores the behavior to adopt when receiving each signal

# Example: ignoring all signals

```
#include <signal.h>

int main(void)
{
        short int  SigNum;
        long       SigVal;

        /* signal returns -1 if it cannot ignore the signal, 0 otherwise */

        for (SigNum = 1; SigNum <= NSIG ; SigNum ++)
        {
                SigVal = signal(SigNum, SIG_IGN);
                printf (" value returned for: %d.-> %d\n",
                 SigNum , SigVal);
        }

...
}
```

# Example: using the alarm signal

```
/* If no key is hit a message is displayed on the screen every 5
seconds */
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
int main (void)
{
  void SigTreat (int Signal);
  int Charac;
  signal (SIGALRM, SigTreat);
  alarm(5);
  do
    {
      Charac = getchar ();
      putchar (Charac);
      alarm(5);
    } while (Charac != EOF);   /* CTRL D to exit */
}

void SigTreat (int Signal)
{
  printf ("\7\7 type a key!\n");
  signal (Signal, SigTreat);
  alarm(5);
}
```

# Example: SIGFPE signal

- SIGFPE :
  - Floating Point Exception signal.
  - Generated, for example, when division by zero occurs
- Program skeleton

```
long int Tab[NLIG, NCOL];
short int NumLig, NumCol;
int main (void)
{
   ...
   init(...);
   calcul(...);
}
```

- Init and calcul functions:
  - Init: reads inputs from keyboard or file
  - Calcul:
    - computes Tab[NumLig, NumCol] the elements of Tab.

# Example: ignore SIGFPE signal

```
long int Tab[NLIG, NCOL];
short int NumLig, NumCol;
void main (void)
{
   ...
   signal(SIGFPE, SIG_IGN);
   ...
   init(...);
   calcul(...);
}
```

- Advantage:
  - If there is an error on the computation of one element, it does not prevent the computation of the others
- Problem:
  - Nothing tells you which elements cannot be computed

# Example: process SIGFPE signal

```
long int Tab[NLIG, NCOL];
short int NumLig, NumCol;
void main (void)
{
    void Treat_FPE(int NumSig);
    ...
    signal(SIGFPE, Traite_FPE);
    ...
    init(...);
    calcul(...);
}
void Treat_FPE(int NumSig)
{
    printf(« signal %d : tab[%d, %d]\n », Numsig, NumLig, NumCol);
    signal(SIGFPE, Traite_FPE);
}
```

- Problem:
  - Does not reset initial conditions that caused the error

# Example: process SIGFPE signal (2)

```
#include <setjmp.h>
long int Tab[NLIG, NCOL];
short int NumLig, NumCol;
jmp_buf context;
int main (void)
{
   void Treat_FPE(int NumSig);
   signal(SIGFPE, Traite_FPE);
   ...
   /* save current context in « context » data structure */
   setjmp(context);
   init(...);
   calcul(...);
}
void Treat_FPE(int NumSig)
{
   printf("signal %d : tab[%d, %d]\n", Numsig, NumLig, NumCol);
   /* reload context previously saved in "context" data structure */
   longjmp(context, 0);
}
```

# Signal processing

- In Unix system V versions, when a process receives a signal, it executes the treatment specified and goes back to the default treatment (it is not true in other versions of Unix, e.g., BSD).

- It is therefore necessary to specify the behavior to adopt next time the same signal is received in the treatment function.

```
void Traite_FPE(int NumSig)
{
  printf(« signal %d : tab[%d, %d]\n, Numsig, NumLig, NumCol);
  signal(SIGFPE, Treat_FPE);
  /* reload context previously saved in « context » data structure
  */
  longjmp(context, 0);
}
```

# Operating systems module

## 2. Processes
## 2.4. Process synchronization

# Why synchronize ?

- Concurrent access to a shared resource may lead to an inconsistent state
  - E. g., concurrent file editing
  - Non-deterministic result (**race condition**): the resulting state depends on the scheduling of processes

- Concurrent accesses need to be **synchronized**
  - E. g., decide who is allowed to update a given part of the file at a given time

- Code leading to a race condition is called **critical section**
  - Must be executed sequentially

- **Synchronization problems**: mutual exclusion, readers-writers, producer-consumer

# Mutual exclusion

- No two processes are in their critical sections (CS) at the same time

+

- Deadlock-freedom: **at least one** process eventually enters its CS
- Starvation-freedom: **every** process eventually enters its CS
  - Assuming no process **blocks** in CS

- Originally: implemented by reading and writing
  - Peterson’s lock, Lamport’s bakery algorithm
- Currently: in hardware (mutex, semaphores)

# Readers-writers problem

- Writer updates a file
- Reader  keeps itself up-to-date
- Reads and writes are non-atomic!

Why synchronization? Inconsistent values might be read

| **Writer** | **Reader** |
| --- | --- |
| T=0: write(“sell the cat”) | |
| | T=1: read(“sell …”) |
| T=2: write(“wash the dog”) | |
| | T=3: read(“… the dog”) |

**Sell the dog?**

# Producer-consumer (bounded buffer) problem

- Producers **put** items in the buffer (of bounded size)
- Consumers **get** items from the buffer
- Every item is consumed, no item is consumed twice

(Client-server, multi-threaded web servers, pipes, …)

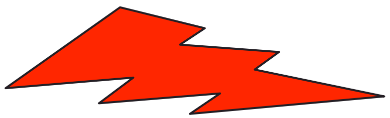
Why synchronization? Items can get lost or consumed twice:

**Producer**

```
/* produce item */
while (counter==MAX);
buffer[in] = item
in = (in+1) % MAX;
counter++;
```

**Consumer**

```
/* to consume item */
while (counter==0);
item=buffer[out];
out=(out+1) % MAX;
counter--;
/* consume item */
```



Race!

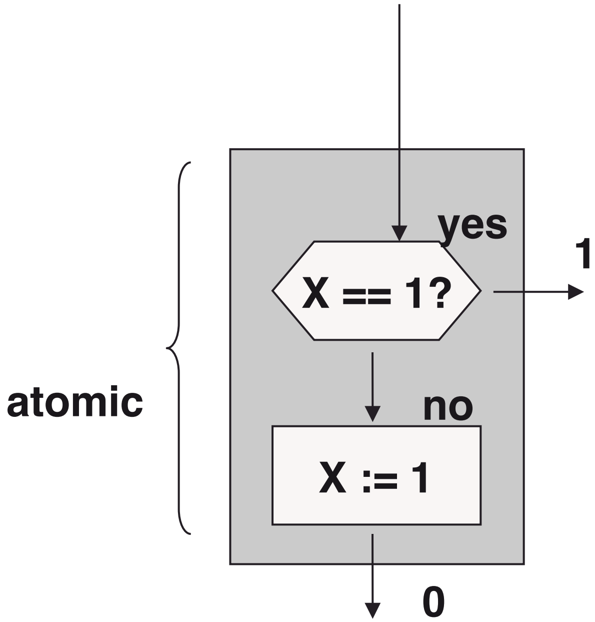
# Synchronization tools

- Test-and-Set (TAS) or Compare-and-Swap (CAS) instructions
- Interrupt masking (requires privileged execution mode).
- Semaphores (locks), monitors

- Nonblocking synchronization
  - No locks used

# Test and Set (TAS)

- TAS(X) **tests** if X = 1, **sets** X to 1 if not, and returns the old value of X
  - Instruction available on almost all processors:




shared X:=0

| Producer | Consumer |
|---|---|
| `while(counter==MAX);` | `while (counter==0);` |
| `. . .` | `. . .` |
| `buffer[in] = item;` | `item = buffer[out];` |
| `. . .` | `. . .` |
| `while TAS(X);` | `while TAS(X);` |
| `counter++;` | `counter--;` |
| `X:=0;` | `X:=0;` |
| `. . .` | `...` |

Problems:

- busy waiting
- no record of request order (for multiple producers and consumers)

# Semaphores [Dijkstra 1968]: specification

- A semaphore S is an integer variable accessed (apart from initialization) with two atomic operations P(S) and V(S)
  - Stands for “passeren” (to pass) and “vrijgeven” (to release) in Dutch

- The value of S indicates the number of resource elements available (if positive), or the number of processes waiting to acquire a resource element (if negative)

```
Init(S,v){ S := v; }

P(S){
      while S<=0;     /* wait until a resource is available */
      S--;            /* pass to a resource */
}

V(S){
      S++;            /* release a resource */
}
```

# Semaphores: implementation

S is associated with a composite object:

- S.counter: the **value** of the semaphore
- S.wq: the **waiting queue,** memorizing the processes having requested a resource element

```
Init(S,R_nb) {
  S.counter=R_nb;
  S.wq=empty;
}
P(S) {
  S.counter--;
  if S.counter<0{
   put the process in S.wq
        until READY;}
}
V(S) {
  S.counter++
  if S.counter>=0{
        mark 1st process in S.wq
  as    READY;}
}
```

# Lock

- A semaphore initialized to 1, is called a **lock** (or **mutex)**
- When a process is in a critical section, no other process can come in

shared semahore S := 1

| Producer | Consumer |
| --- | --- |
| `while (counter==MAX);` | `while (counter==0);` |
| `. . .` | `. . .` |
| `buffer[in] = item;` | `item = buffer[out];` |
| `. . .` | `. . .` |
| `P(S);` | `P(S);` |
| `counter++;` | `counter--;` |
| `V(S)` | `V(S);` |
| `. . .` | `...` |

**Problem**: still waiting until the buffer is ready

# Semaphores for producer-consumer

- 2 semaphores used :
  - **empty:** indicates empty slots in the buffer (to be used by the producer)
  - **full:** indicates full slots in the buffer (to be read by the consumer)

```
shared  semaphores empty := MAX, full := 0;
```

| Producer | Consumer |
| --- | --- |
| `P(empty)`<br>`buffer[in] = item;`<br>`in = (in+1) % MAX;`<br>`V(full)` | `P(full);`<br>`item = buffer[out];`<br>`out=(out+1) % MAX;`<br>`V(empty);` |

# Potential problems with semaphores/locks

- **Blocking**: progress of a process is conditional (depends on other processes)
- **Deadlock** (no progress ever made)

```
X1:=1; X2:=1
```

| Process 1 | Process 2 |
|---|---|
| ... | ... |
| P(X1) | P(X2) |
| P(X2) | P(X1) |
| **critical section** | **critical section** |
| V(X2) | V(X1) |
| V(X1) | V(X2) |
| ... | ... |

- **Starvation** (waiting in the waiting queue forever)

# Non-blocking algorithms

A process makes progress, regardless of the other processes

shared **buffer**[MAX]:=empty; **head**:=0; **tail**:=0;

| Producer `put(item)` | Consumer `get()` |
|---|---|
| `if (tail-head == MAX){`<br>`        return(full);`<br>`}`<br>`buffer[tail%MAX]=item;`<br>`tail++;`<br>`return(ok);` | `if (tail-head == 0){`<br>`        return(empty);`<br>`}`<br>`item=buffer[head%MAX];`<br>`head++;`<br>`return(item);` |

**Problems**:

- works for 2 processes but hard to say why it works ☺
- multiple producers/consumers?

(go to a distributed computing class to learn more)

# Multiple producers-consumers: transactions

- Mark sequences of instructions as an **atomic transaction**, e.g., the resulting producer code:

```
atomic {
        if (tail-head == MAX){
        return full;
        }
        items[tail%MAX]=item;
        tail++;
}
return ok;
```

- A transaction can be either **committed** or **aborted**
  - Committed transactions are **serializable**
  - Let the transactional memory (TM) care about the conflicts
  - Easy to program, but performance may be problematic in software

# More on synchronization

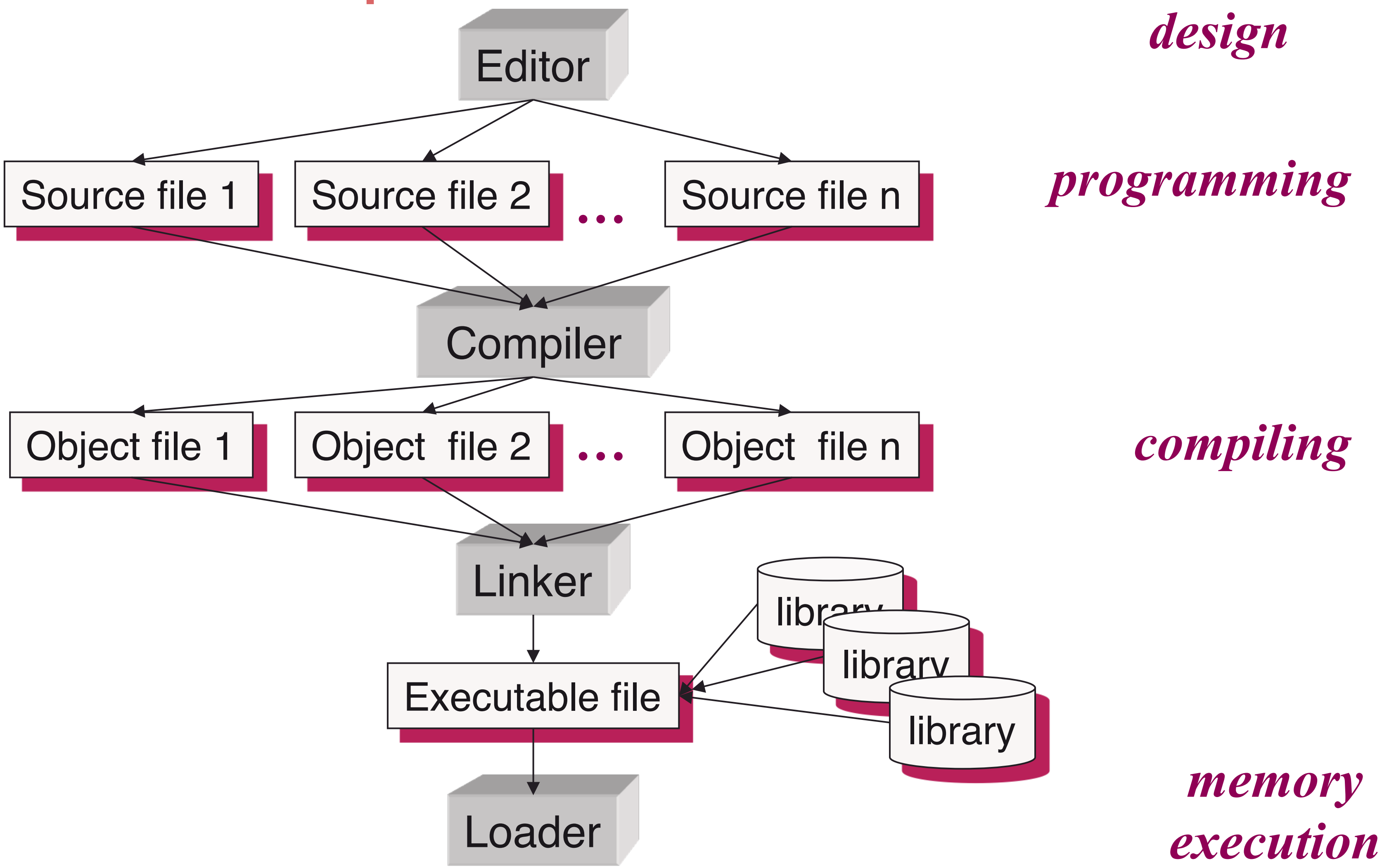
- Concurrency is indispensable in programming:
  - Every system is now concurrent
  - Every parallel program needs to synchronize
  - Synchronization cost is high (“Amdahl’s Law”)
- Tools:
  - Synchronization libraries (e.g., `java.util.concurrent`)
  - Synchronization primitives (e.g., TAS, CAS, LL/SC)
  - Transactional memory, also in hardware (Intel Haswell, IBM Blue Gene,…)
- More on blocking and nonblocking concurrent algorithms:
  - M. Herlihy, N. Shavit. The art of multiprocessor programming. Morgan Kaufman. 2008

# Operating Systems Module

## 3. Software life cycle

## 3.1. Compiling, Linking, Executing

# Software production tools



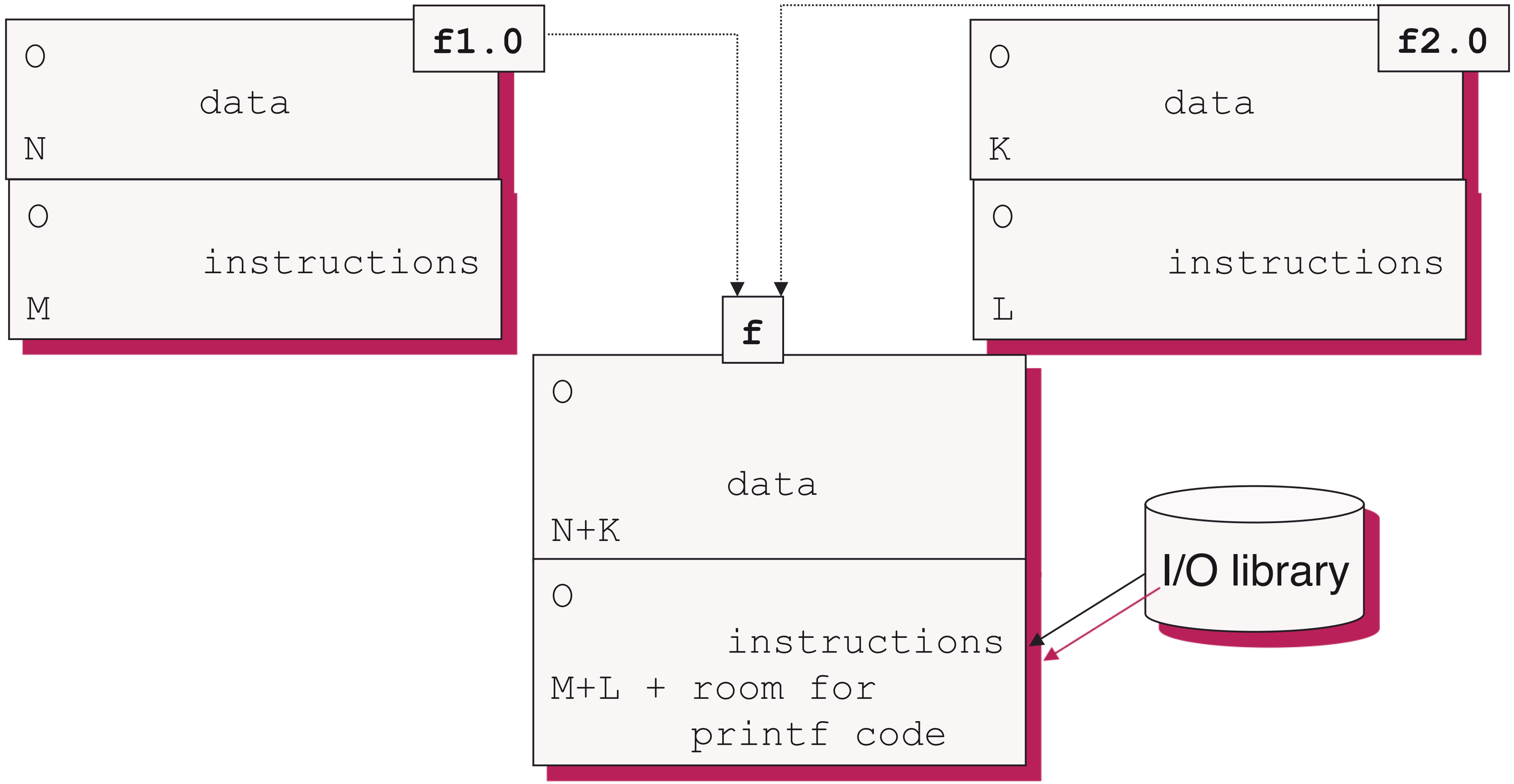
# Separate compilations

```
f1.c
int main(void)
{
        int i ;
        i=square(4) ;
        printf("i=%d\n", i);
}
```

```
f2.c
double square(double f)
{
        return(f*f);
}
```

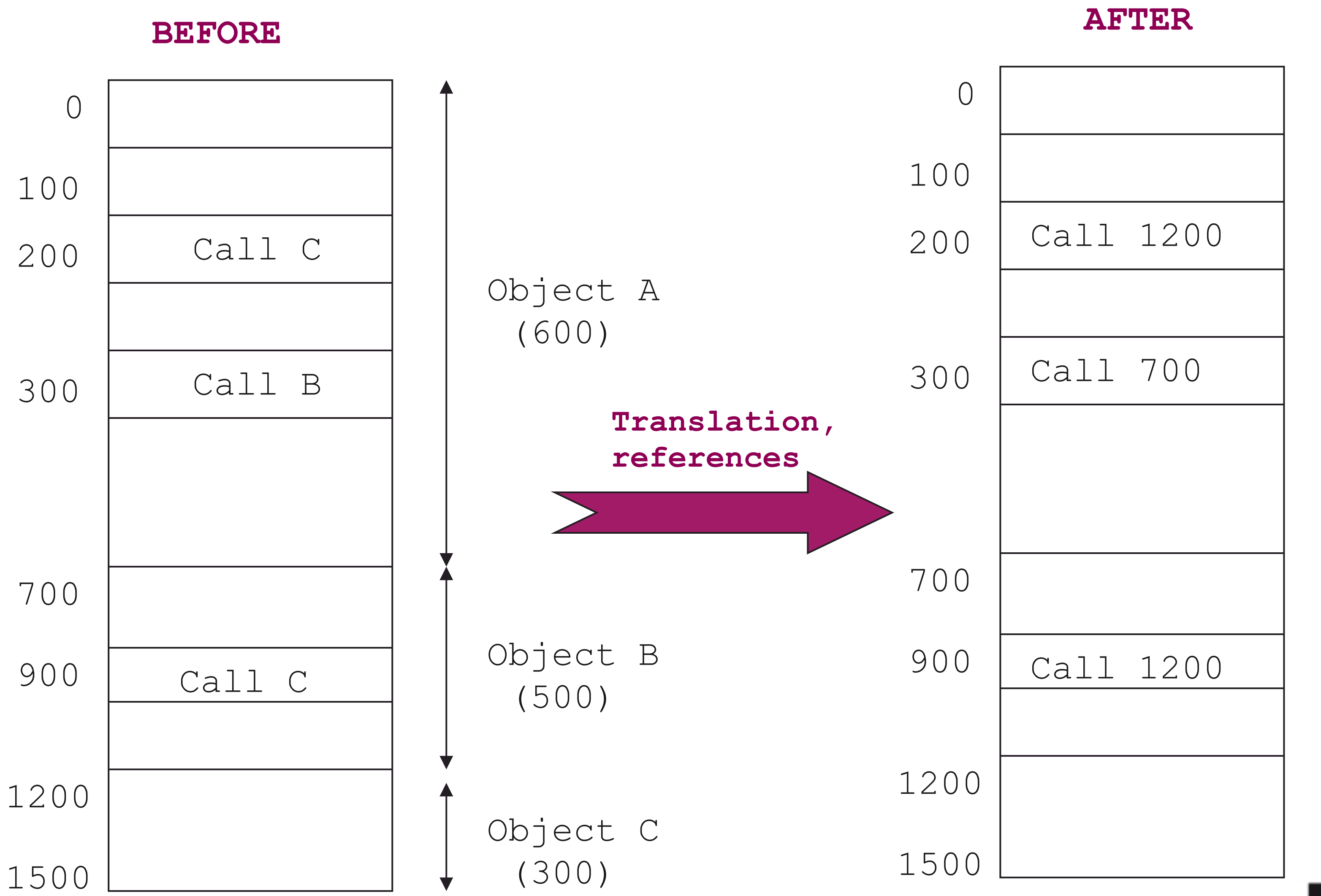
- Separate compilations:
  - gcc -c f1.c (=> produces object file f1.o)
  - gcc -c f2.c (=> produces object file f2.o)
- Linking
  - gcc -o f f1.o f2.o (=> produces executable file f)
- Executing
  - ./f
  - **i=0 ????**

# From object files to executable file




- Here: **static linking**
- In **dynamic linking,** only a **pointer** to modules is kept. They are only loaded at **execution time** if needed.

# Link editing



# Compilation errors

- Going back to the previous example (f1, f2)
  - execution of: `gcc -Wall -c f1.c`
  - returns: `warning implicit declarations: square, printf`
  - Why?
    - Prototyping poorly done
    - Missing prototype of `square` and `#include <stdio.h>` for `printf`

- Execution of: `gcc -o f f1.o`

  returns:
  ```
  undefined symbol: square
    ld fatal error
  ```

- Why? missing object file f2.o containing code of `square`

# Preprocessor prototyping and #include

- `f` cannot return the expected result because the use of `square` does not correspond to the way it is defined (treats the output as int).

```
i (format d) = 0             i (format lf) = 16.000000
i (format d) = 320000        i (format ld) = 1076887552
```

- To avoid problems:
  - Compile with the `-Wall` option
  - Give prototype of all functions called (in .h files)
- Example corrected

**f1.c**

```
#include <stdio.h>
#include "f2.h"
int main(void)
{
    int i ;
    i=square(4) ;
    printf("i=%d\n", i);
}
```

**f2.h**

```
extern double square(double f);
```

**f2.c**

```
double square(double f)
{
        return(f*f);
}
```

# Libraries and include files

**f1.c (modified)**

```
#include <stdio.h>
#include "f2.h"
int main(void)
{
    int i ;
    i= square(4) ;
    printf("i=%d\n", i);
    i=sqrt(i);
    printf("i=%d\n", i);
}
```

- result of:

```
gcc -o f f1.c f2.c
```

- will be:

```
warning: type mismatch
ld: undefined symbol sqrt
```

- To avoid the `warning`, cast the variable appropriately (i=(int) square(4))
- To avoid the `ld: undefined ...`,
  - #include <`math.h`>
  - load math library:

```
gcc -o f f1.c f2.c -lm
```

# Libraries and include files (2)

**f1.c (modified)**

```
#include <stdio.h>
#include <math.h>
#include "f2.h"
int main(void)
{
     int i ;
     i= (int) square(4) ;
     printf("i=%d\n", i);
     i=sqrt(i);
     printf("i=%d\n", i);
}
```

- result of:
  - `gcc -Wall -o f f1.c f2.c`
- will be:
  - `ld: undefined symbol sqrt`
- result of:
  - `gcc -Wall -o f f1.c f2.c -lm`
- will be OK

# Prototyping and library errors

| Action | Effect |
|---|---|
| No `#include`<br>No library loading | No executable produced |
| No `#include`<br>Library loading | Executable produced but potential problems at runtime |
| `#include`<br>No library loading | No executable produced |

# Operating Systems Module

## 3. Software life cycle

## 3.2 Make Tool

# Make tool

- **make** is a tool that provides support
  - to maintain an up-to-date executable file from various modules
  - by recompiling, linking, etc. what is necessary..
- To this purpose, **make** uses:
  - the dates at which the files were last modified (/compiled ?)
  - the dependencies among the various modules
- The dependencies among the modules and the actions to undertake in order to generate the executable file are described in a "makefile".

- Usage:

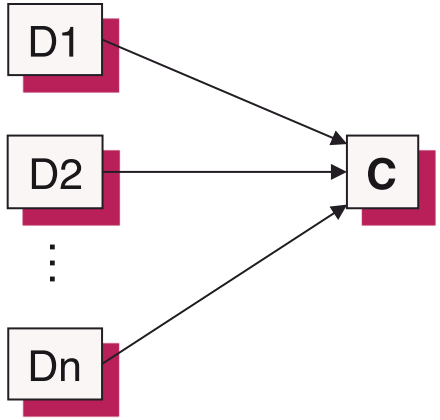
`make -f makeFileName`

or, simply

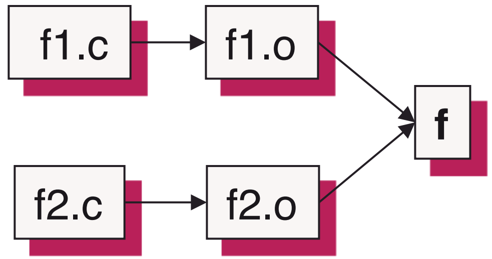
`make` (looks for "`makefile`" or "`Makefile`" in the current directory)

# Makefile, target and dependencies

- Dependency graph
  - If file A is dependent on file B there will be an arc from B to A.
  - E.g., C depends on D1, D2, …, Dn, i. e., C is a target that depends on N1, N2, Nn.




- Example dependency graph
  - f depends on f1.o and f2.o




- The structure of a makefile reflects the structure of the corresponding dependency graph.
- A makefile is a series of target, action lines:

- `target:      dependency 1 ... dependency n`
- `<TAB>action`
- The action describes what is to be done to obtain the target from the dependency files.

# First version of makefile

```
### f depends on f1.o and f2.o
f:      f1.o f2.o
<TAB>   gcc f1.o f2.o -o f
### f1.o depends on f1.c
f1.o:   f1.c
<TAB>   gcc -c f1.c
### f2.o depends on f2.c
f2.o:   f2.c
<TAB>   gcc -c f2.c
```

- If f1.c is modified then the f1.o target and the f targets will be redone.
- f2.o will remain unchanged

# Second version of makefile

```
f:      f1.o f2.o
        gcc f1.o f2.o -o f

### f1.o depends on f1.c & f2.h
f1.o:   f1.c f2.h
        gcc -c -Wall f1.c

f2.o:   f2.c
        gcc -c -Wall f2.c

clean:
        rm *.o core
```

NB:

- the target does not have to be a file.
- `make clean` will clean the current directory.

# Makefile variables

- In the table below, we show some of the commonly used makefile variable

| | Name | example |
|---|---|---|
| Compiling option | CFLAGS | CFLAGS=-c -g -Wall |
| Linking options | LDFLAGS | LDFLAGS= -g -lm |
| Object files | OFILES | OFILES=f1.o f2.o |
| Sources files | CFILES | CFILES=f1.c f2.c |
| Compiler name | CC | CC=gcc |
| Linker name | LD | LD=gcc |
| Rm command | RM | RM=/bin/rm |
| Program name | PROG | PROG=f |

# Third version of makefile

- With variables

```
############################################################
 BINDIR          =       /usr/local/bin
 CFLAGS          =       -c -g -Wall
 LDFLAGS         =       -g -lm
 OFILES          =       f1.o f2.o
 CC              =       $(BINDIR)/gcc
 LD              =       $(BINDIR)/gcc
 RM              =       /bin/rm -f
 PROG            =       f
############################################################
 f:      $(OFILES)
         $(LD) $(LDFLAGS) $(OFILES) -o $(PROG)
 f1.o:   f1.c f2.h
         $(CC) $(CFLAGS) f1.c
 f2.o:   f2.c
         $(CC) $(CFLAGS) f2.c
############################################################
 clean:
         $(RM) $(OFILES) core
```

# Suffix rules

- To go from a source to an object file, there is an implicit suffix rule:

```
.c.o:
        $(CC) $(CFLAGS) $<
```

- Thus:

```
BINDIR          =       /usr/local/bin
CFLAGS          =       -c -g -Wall
LDFLAGS =       -g -lm
OFILES          =       f1.o f2.o
CC              =       $(BINDIR)/gcc
LD              =       $(BINDIR)/gcc
RM              =       /bin/rm -f
PROG            =       f
#########################################################
#
f:      $(OFILES)
        $(LD) $(LDFLAGS) $(OFILES) -o $(PROG)
f1.o:   f1.c f2.h
        $(CC) $(CFLAGS) f1.c
#########################################################
#
clean:
        $(RM) $(OFILES) core
```

# Organizing the directories

- A separate directory with all the “include” (*.h) files can be created.
- The `gcc -I` option can then be used to tell the compiler where to look for the include files

```
################################################
CFLAGS=       -c -g -Wall -I ../includeDir
```

# Operating Systems Module

## 4. File System

# Definitions

- A **file** is a named collection of related information that is recorded on secondary storage (disks)
  - but can be mapped to the primary (main) memory
- File’s attributes:
  - **Name:** symbolic file name
  - **Identifier**: unique tag, identifies the file within the file system
  - **Type:**
    - Program files: source, object, executable
    - Data files: ASCII, binary, media files (various format), ...
    - System files, such as /etc/passwd, /var/spool/mail
  - **Location:** a pointer to a device and a **path** on the device
  - **Size:** the current size (bytes, words, or **blocks**) and, possibly, the max allowed size
  - **Protection:** access control information
  - **Time, date, and user:** for creation, last modification, last use

# File operations

- Access operations exported to the OS:
  - `File operations:`
    - `create, open, read, write, seek, close, delete, truncate,…`
  - Directory operations:
    - `create, delete, opendir, closedir, readdir, link, …`
- System commands: `cat, ls, file, rm, mv, cp, …`

# File system manager

The **file system manager** is a part of the operating system

- It is in charge of all operations on file, of file storage, protection and integrity

- It establishes a **mapping** between the **logical** organization seen by the users and the **physical** organization of storage devices.

# Operating Systems Module

## 4. File System
## 4.1. Physical organization

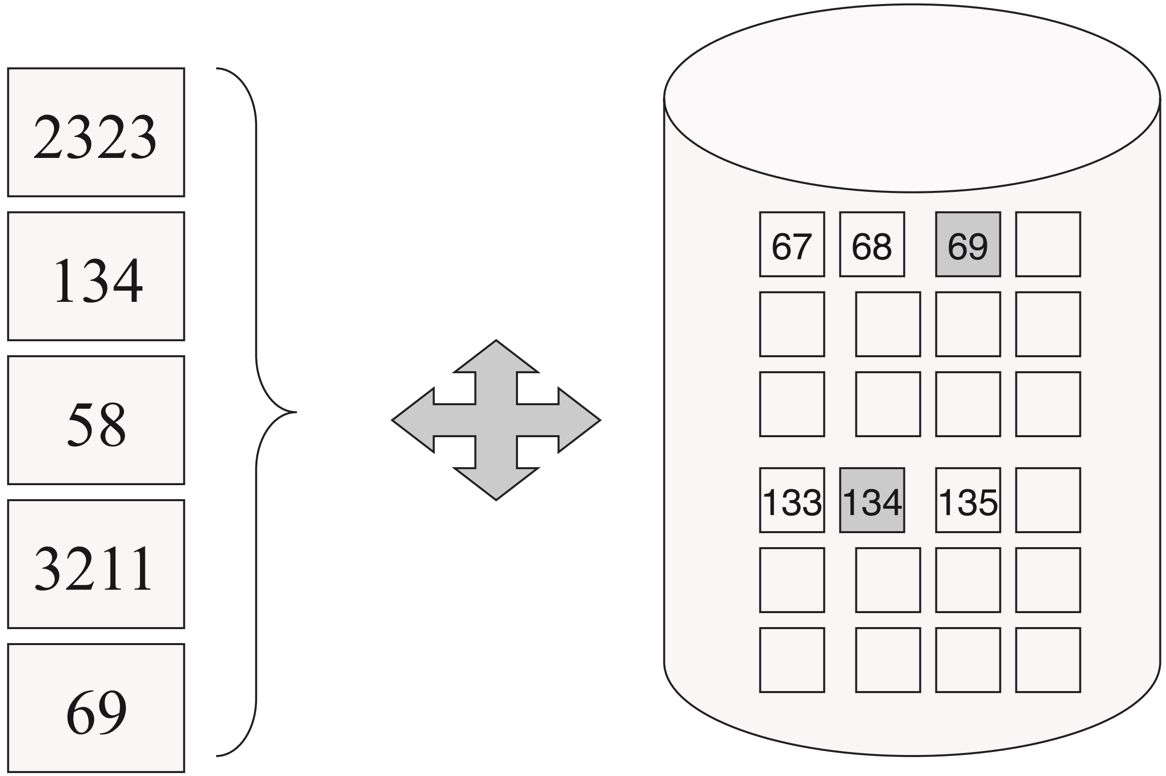
# File system implementation under Unix

- On each storage disk, a special file called **i-list** (**index-list**) describes all files stored on disk.

- An i-list entry is called an **i-node**
  - Each i-node describes one file
  - An i-node is stored on 64 bytes (if file too large, **indirection** is used)

- i-list size
  - Set when the disk is initialized
  - If the list contains a large number of entries (i-nodes), many files can be created, but the i-list is large
  - If the i-list contains few i-nodes, then the i-list is small but only a small number of files can be created, even if there is available space on disk.

# File storage under Unix




File composed of 5 blocks

Blocks on Unix disk

# Super block ... and disk organization

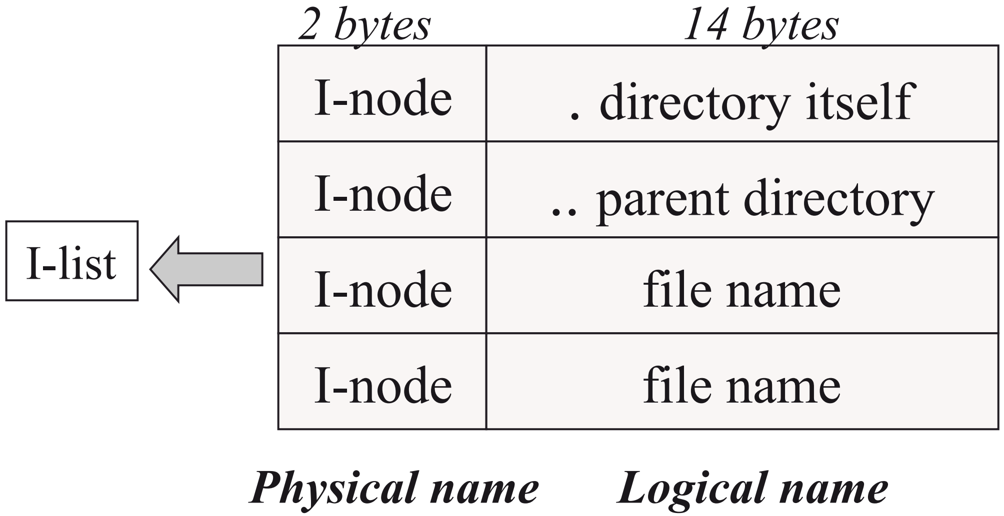

- The super block contains the following information:
  - Number of blocks reserved for i-list
  - Total number of blocks in disk
  - List of free blocks
  - List of free i-nodes
  - ...
  - Date of last modification
  - Number of free blocks
  - Number of free i-nodes
  - File system name

# I-node: example

- Protection: access rights
- UID and GID are the creator ids
- Disk @1 to disk @10 contain the addresses of the first 10 blocks composing the file.
- Disk @11, addresses a block that contains the addresses of the 128 following blocks (assuming that a block is 512 octets long)
- Disk @12, addresses a block that addresses 128 blocks that each contain the addresses of 128 file blocs (2 levels of indirection)
- Disk @13, addresses a block that addresses 128 blocks that each address 128 blocs that each contain the addresses of 128 file blocs (3 levels of indirection)

| |
|---|
| Protection |
| Number of links |
| UID-GID |
| Number of characters |
| ... |
| Disk @ 1 |
| Disk@ 2 |
| ... |
| Disk @ 10 |
| Disk @ 11 |
| Disk @ 12 |
| Disk @ 13 |

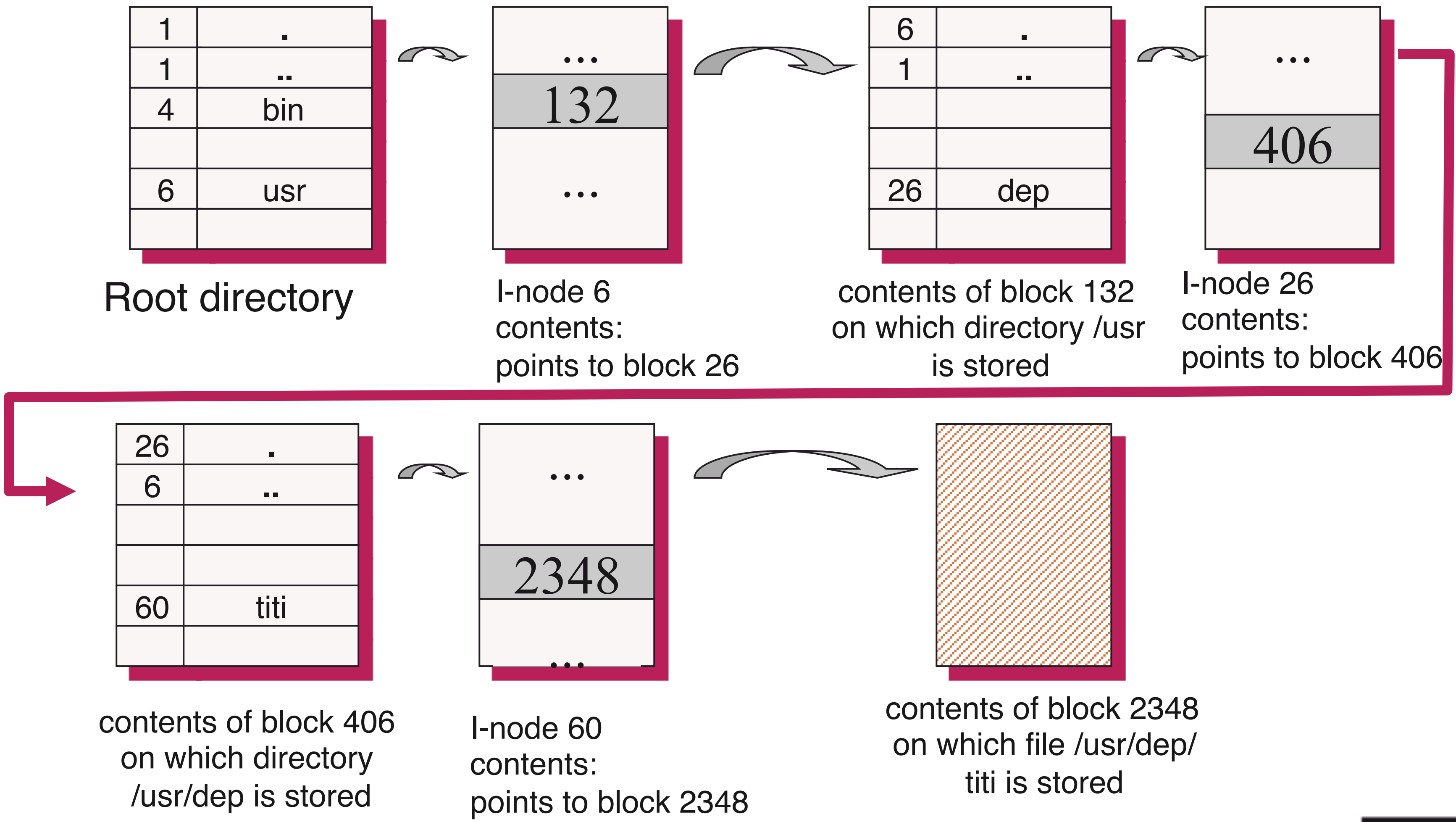
# Unix directory




- The above figure shows the directory « historical structure » (now names can be longer than 14 bytes).
- Under Unix, directories establish the mapping between logical and physical structures, that are completely separated.
- Some directory commands:
  - `pwd` (print working directory), gives the name and path of current directory
  - `cd` (change directory), to move in the directory graph
  - `ls`, to list files contained in a directory

# Unix file access: example

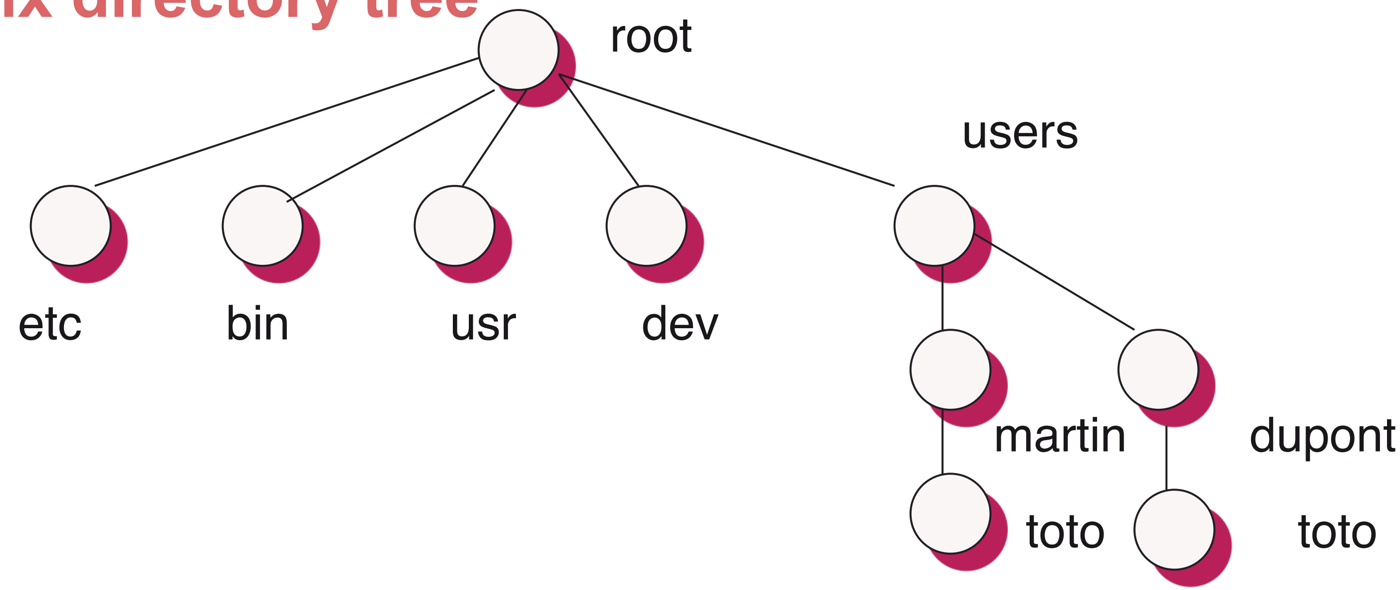
- Assume looking for /usr/dep/titi



# Operating systems module

## 4. Files
## 4.2. Logical organization

# Unix directory tree



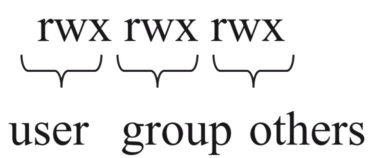

- /dev: device files (note that under unix devices are considered as files)
- /etc: management files such as passwd, group, hosts
- /bin and /usr/bin: shell commands
- /usr/include: header files (.h)
- /var/spool/mail: for mail
- /tmp: a useful directory to which anyone may write

# File structure and file access under Unix

- Under Unix a file is a sequence of bytes with no other structure
  - It is ended by a special EOF (End Of File) character
  - Advantage:
    - Small, portable system
    - Universal: all files and devices are managed in the same way
  - Drawback:
    - Many functions to be written by the user when more complex file access schemes are needed
- Sequential access
  - Used by default: the access functions (`read`, `write`) use a cursor that is moved each time an access is performed.
- Direct or random access
  - Can be performed using functions such as `lseek` that can be used to explicitly move the cursor to the desired position.

# Access rights

- Upon file creation:
  - the file inherits the UID and GID of the file owner (as specified in /etc/passwd).
  - In the i-node, the access rights are set using the umask found in the owner's environment.
- The access rights are coded over 9 bits
- Example 1:




  - A file with rights rw- r-- ---
  - Can be read and written by the owner and can be read by members of the same group.
- To change a file access rights use 'chmod' that has two « modes »
  - `chmod 644 file`
    - Gives read/write access to the user, and read access to group and other
  - `chmod g+w  file`
    - Adds 'write' access to the group.

# Access rights (2)

Example 2: making a directory private

```
mkdir private
chmod 711 private
cd private
mkdir dir1
chmod 755 dir1
```

No one (other than the owner) can read private (through ls) everybody can access dir1 (cd dir1).

**NB** You need the ‘x’ right to execute ‘cd’, and the ‘r’ right to execute ‘ls’

```
chmod a+x toto
chmod -R 755 dir-name/
```

# Standard files

| | High level | Low level (descriptor) | Default |
|---|---|---|---|
| Standard input | stdin | 0 | keyboard |
| Standard output | stdout | 1 | screen |
| Error output | stderr | 2 | screen |

# Operating systems module

## 4. Files
## 4.3. C input/output library

# C input/output library

- 2 libraries are available
  - Low level (open, read, write, etc)
  - High level (fopen, fread, fwrite, etc)
- File access: associate a file local name to a file global name:
  - Low level:
    - a file descriptor is used.
    - It is an integer value returned by the ‘open’ call.
    - It corresponds to an index in the table where the list of open files is maintained.
  - High level:
    - A FILE pointer is used
    - It is returned by the ‘fopen’ call.
    - It is a pointer to a file structure describing the file (FILE *)
- We now focus on the low level library

# Opening a file

- Before accessing a file you need to ‘open’ it:

```
int desc
  ...
desc=open(" toto ", O_RDWR);
if(desc == -1)
{
      perror(" open toto ");
      exit(1)
}
```

- An entry is created in the table corresponding to the opened files.
- ‘desc’ is the index (**descriptor**) of the newly created file in the table
- O_RDWR: Open for reading (RD) and writing (WR)
- If the file does not exist, it is created.

- File creation: `int creat(char *nom, int mode);`
- ‘mode’ denotes the file access rights.

# Reading, writing, closing

```
int write(int filydesc, char *buffer, unsigned nbyte)
int read(int filydesc, char *buffer, unsigned nbyte)
int close(int filydesc)
```

Example

```
int main ( void )
{
  int MyFile, Ret_Read, Ret_Write;
  char MyArray [512] ;
  MyFile= open ("toto", O_RDONLY) ;
  if (MyFile== -1)
  {
    perror ("open");
    •••
  }
  while((Ret_Read = read (MyFile, MyArray, 512)) > 0)
  {
     •••
     Ret_Write = write(1, MyArray, Ret_Read);
     if (Ret_Write == -1)
       { /* error processing */ };
  }
  close (MyFile) ;
}
```

# Moving the cursor

- A cursor keeps track of the byte last read or written
- To move the cursor: 'lseek'
- ```
  long lseek(int fildes, long offset, int from)
  ```
- 'from'
  - 0, from beginning of file
  - 1 from current position
  - 2 from the end of file
- Lseek returns:
  - cursor value
  - -1 if error.
  ```

# Other I/O functions not detailed here

- Character mode: getc, putc, getchar, putchar
- Formatted I/O: scanf, printf, fscanf, fprintf, sscanf, sprintf
- .....

# Synchronizing file accesses

- To lock
  - lockf(filedesc, F_LOCK, nb_octets)
  - locks the nb_octets following the current cursor position (use lseek if need to set the cursor)
- To unlock
  - lockf(filedesc, F_ULOCK, nb_octets)
  - unlocks the nb_octets following the current cursor position (use lseek if need to set the cursor)

- If nb_octets=0 locks/unlocks all file (**recommended**)

# Synchronizing file accesses (2)

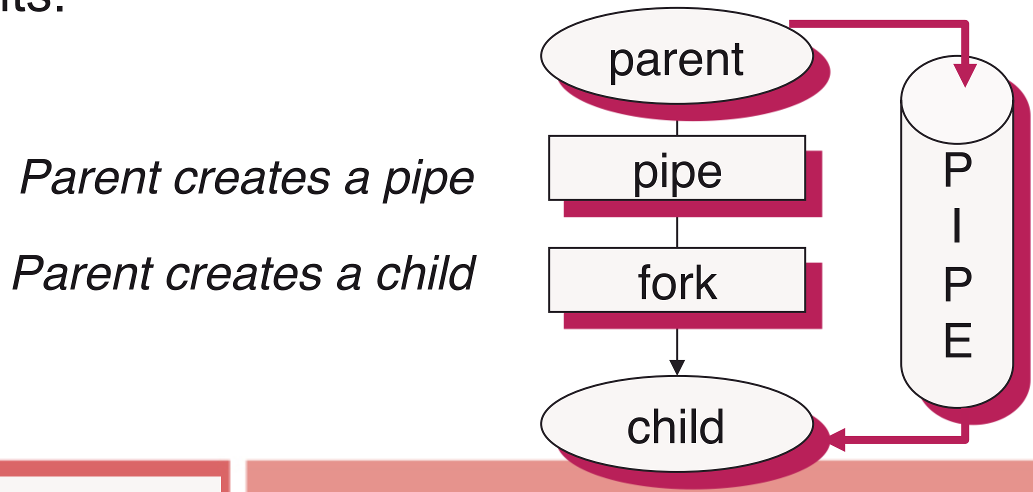
```
lseek(sortie, 0, 0);
ilock=lockf(sortie, F_LOCK, 0);
printf("Proc %d entering critical section\n", (int)getpid());
/* begin critical section */
...
/* end critical section */
lseek(sortie, 0, 0);
ilock=lockf(sortie, F_ULOCK, 0);
printf("Proc %d exited critical section\n", (int)getpid());
```

# Operating systems module

## 4. Files
## 4.4. Using pipes

# Pipes

- Pipes are a special type of producer/consumer files
- A standard pipe
  - is declared using the ‘pipe’ call
  - used by processes with a common ancestor
  - unidirectional (bidirectional on Unix System V)
- A named pipe
  - Is declared using the ‘mknod’ call
  - It can be accessed by any process who knows its name and have the proper access rights.



# Using standard pipes

```
int main(void)
{
  int Ret_fork, Pipe[2], State;
  char c;
  pipe(Pipe);
  Ret_fork=fork();
  if (Ret_fork!=0)
  {
    close(Pipe[0]);
    printf("send characters!\n");
    while((c=getchar())!=EOF)
      write(Pipe[1], &c, 1);
    wait(&State);
  }
  if (Ret_fork == 0)
  {
     printf("Child ready to read.\n");
     close(Pipe[1]);
     while(read(Pipe[0], &c, 1))
       printf("characters received=%c\n", c);
     exit(0);
   }
}
```

# Using named pipes

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

int main(int nb_arg, char **argv)
{
  int Ret_mknod;
  if (nb_arg!=2)
  {
     printf("argument was expected");
     exit(1);
  }

  /* create a named pipe accessible by everybody */
  Ret_mknod=mknod(argv[1], S_IFIFO|0666, 0)
  if (Ret_mknod!=0)
  {
     perror("mknod");
     exit(1);
  }
  printf("Pipe %s created.\n", argv[1]);
  ...
}
```

# Using pipes from the shell

- Examples
  - ps -ax | grep dupont
  - ypcat passwd | grep isabelle

# Operating systems module

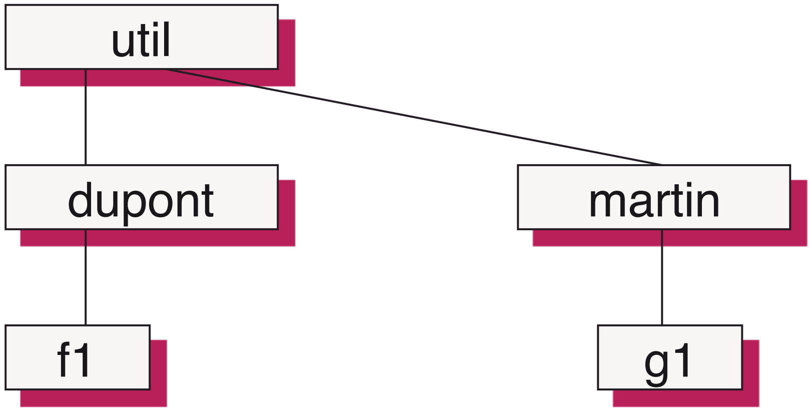
## 4. Files
## 4.5. Useful commands
## cp, mv, rm, ln

# cp command

- Assume that, from his home directory, user dupont types:

```
cp f1 f2
```

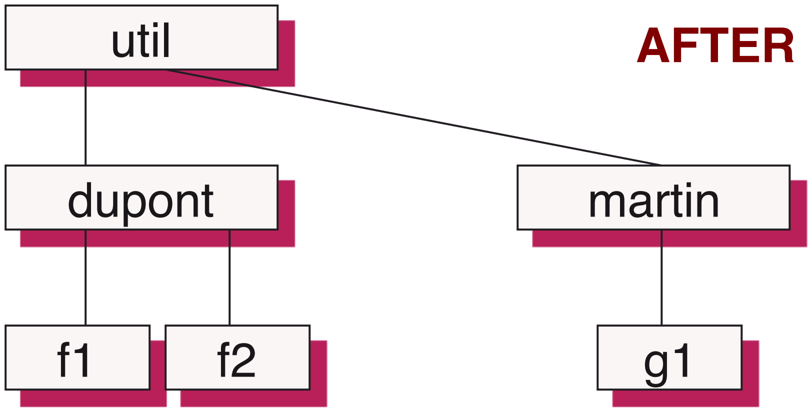
**BEFORE**




| | |
|---|---|
| 29 | . |
| 12 | .. |
| 50 | f1 |
| | |

***dupont directory***

**AFTER**



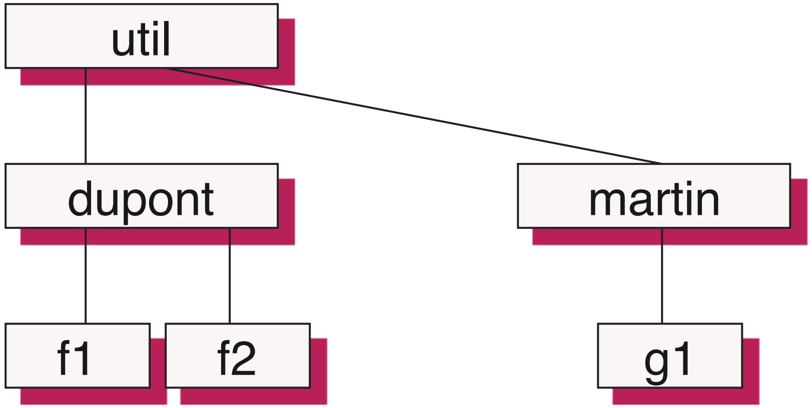

| | |
|---|---|
| 29 | . |
| 12 | .. |
| 50 | f1 |
| 62 | f2 |

# mv command

- Assume that, from his home directory, user dupont types:

```
mv f2 f50
```

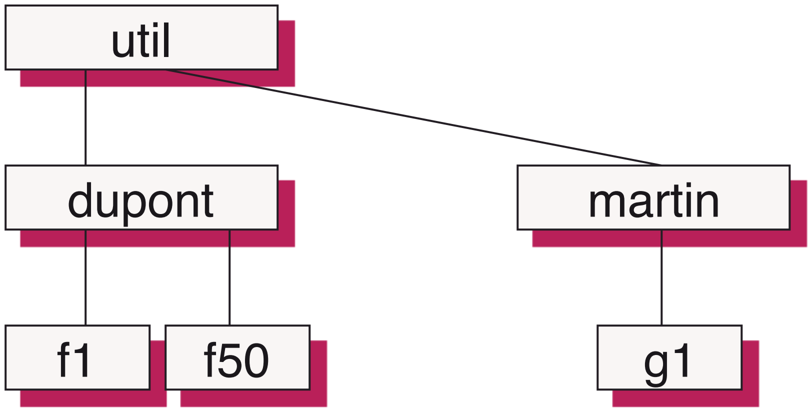
**BEFORE**




| | |
|---|---|
| 29 | . |
| 12 | .. |
| 50 | f1 |
| 62 | f2 |

***dupont directory***

**AFTER**



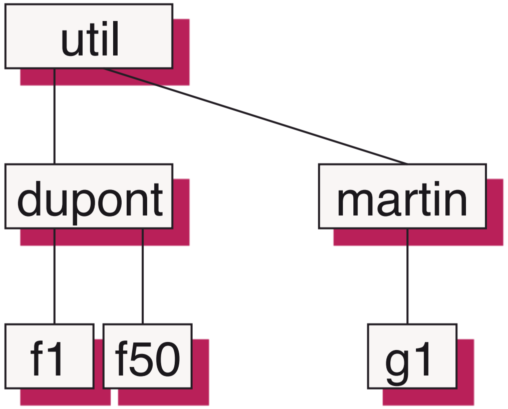

| | |
|---|---|
| 29 | . |
| 12 | .. |
| 50 | f1 |
| 62 | f50 |

# ln command

- Assume that, from his home directory, user martin types:

```
ln ../dupont/f50 g2
```

**BEFORE**



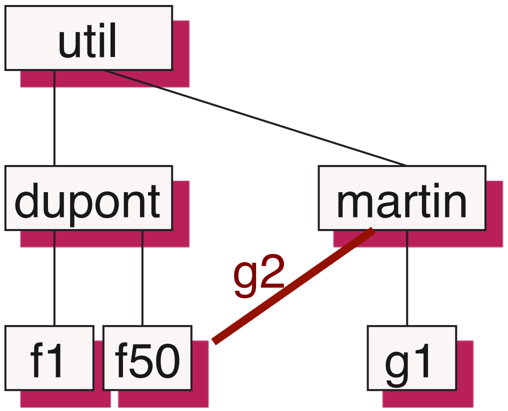

| | |
|---|---|
| 29 | . |
| 12 | .. |
| 50 | f1 |
| 62 | f50 |

*dupont dir*

| | |
|---|---|
| 38 | . |
| 12 | .. |
| 47 | g1 |
| | |

*martin dir*

**AFTER**




| | |
|---|---|
| 29 | . |
| 12 | .. |
| 50 | f1 |
| 62 | f50 |

*dupont dir*

| | |
|---|---|
| 38 | . |
| 12 | .. |
| 47 | g1 |
| 62 | g2 |

*martin dir*

File not duplicated; link counter incremented by one !

# ln –s command: symbolic link

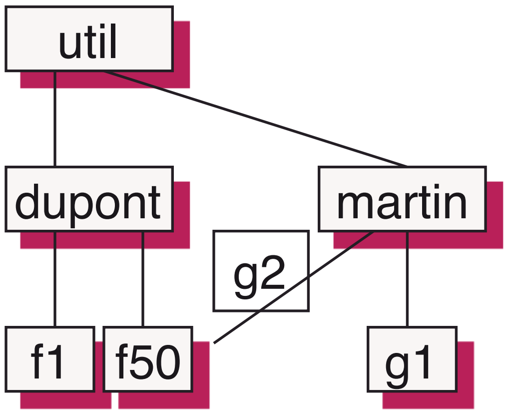
- This command creates a new file that contains the input string
- The created file is of type 'l' (linked file)
- Example
  - ln –s ../dupont/f50 g3
  - creates in dupont a new file named 'g3' containing '../dupont/f50'
- 'g3' is a pointer to f50
- Watch out:
  - If dupont types
    - mv f50 f60
  - Then cat g3 will return 'file not found'

```
> ls -l
> -rw-r--r-- 2 martin staff 31 Sep 25 11:11 g2
> lrwxr-xr-x 1 martin staff 20 Sep 25 11:20 g3->../dupont/f50
```

# ln –s command

- Assume that, from his home directory, user martin types:
  - ln –s ../dupont/f50 g3

**BEFORE**



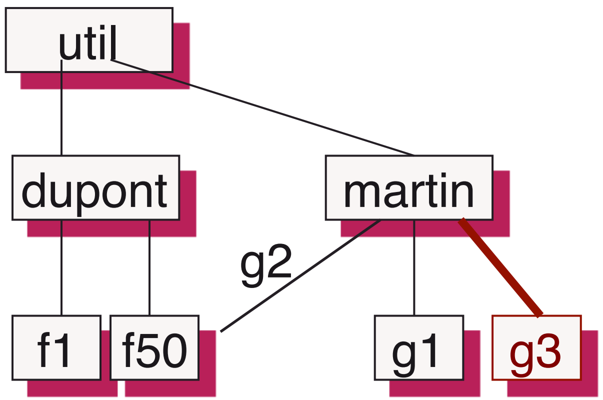

| | |
|---|---|
| 29 | . |
| 12 | .. |
| 50 | f1 |
| 62 | f50 |

***dupont dir***

| | |
|---|---|
| 38 | . |
| 12 | .. |
| 47 | g1 |
| 62 | g2 |

***martin dir***

**AFTER**




| | |
|---|---|
| 29 | . |
| 12 | .. |
| 50 | f1 |
| 62 | f50 |

***dupont dir***

| | |
|---|---|
| 38 | . |
| 12 | .. |
| 47 | g1 |
| 62 | g2 |
| 101 | g3 |

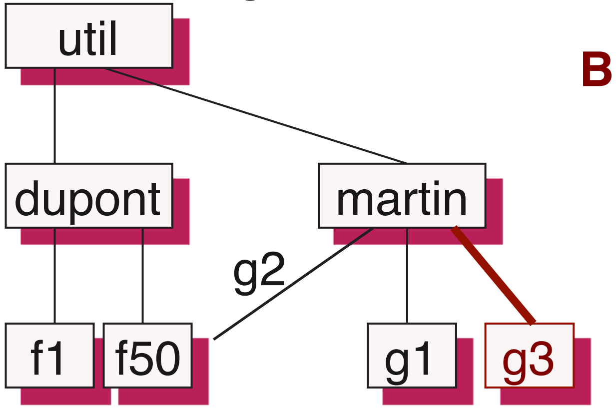
***martin dir***

Contents of g3 file is ../dupont/f50

# rm command

- rm /util/martin/g1

**BEFORE**



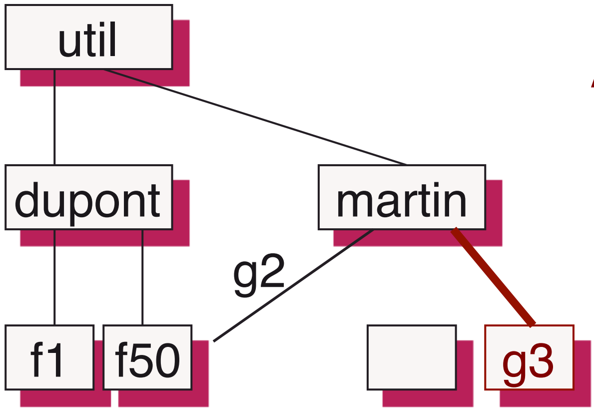

| | |
|---|---|
| 29 | . |
| 12 | .. |
| 50 | f1 |
| 62 | f50 |

*dupont dir*

| | |
|---|---|
| 38 | . |
| 12 | .. |
| 47 | g1 |
| 62 | g2 |
| 101 | g3 |

*martin dir*

**AFTER**




| | |
|---|---|
| 29 | . |
| 12 | .. |
| 50 | f1 |
| 62 | f50 |

*dupont dir*

| | |
|---|---|
| 38 | . |
| 12 | .. |
| 0 | g1 |
| 62 | g2 |
| 101 | g3 |

*martin dir*

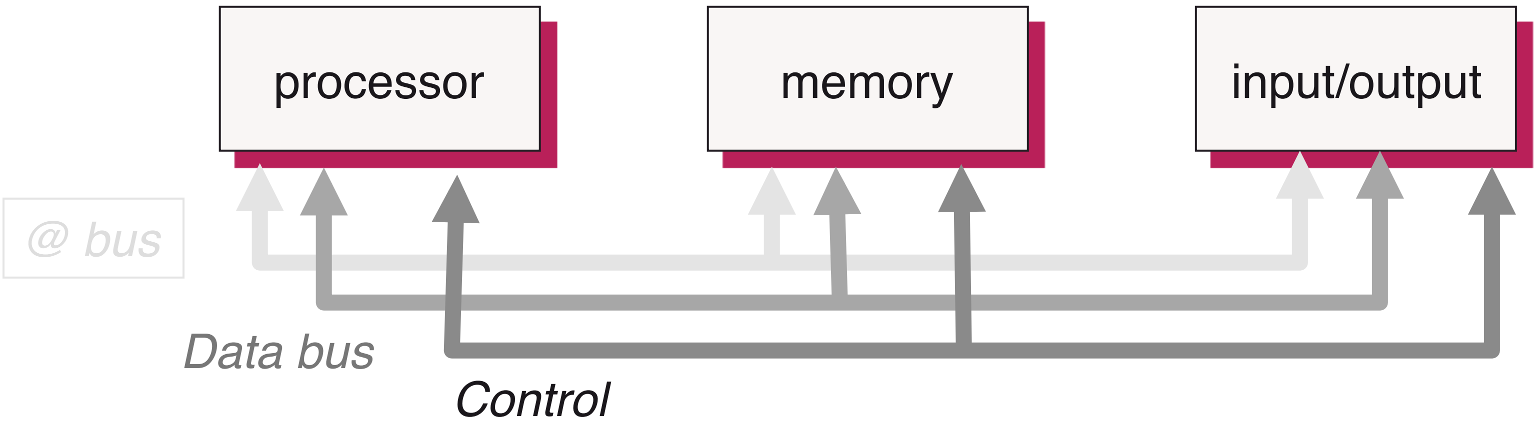
Zeros the i-node reference of g1

# Operating Systems Module

## 5. Memory Management
## 5.1. Definitions and concepts

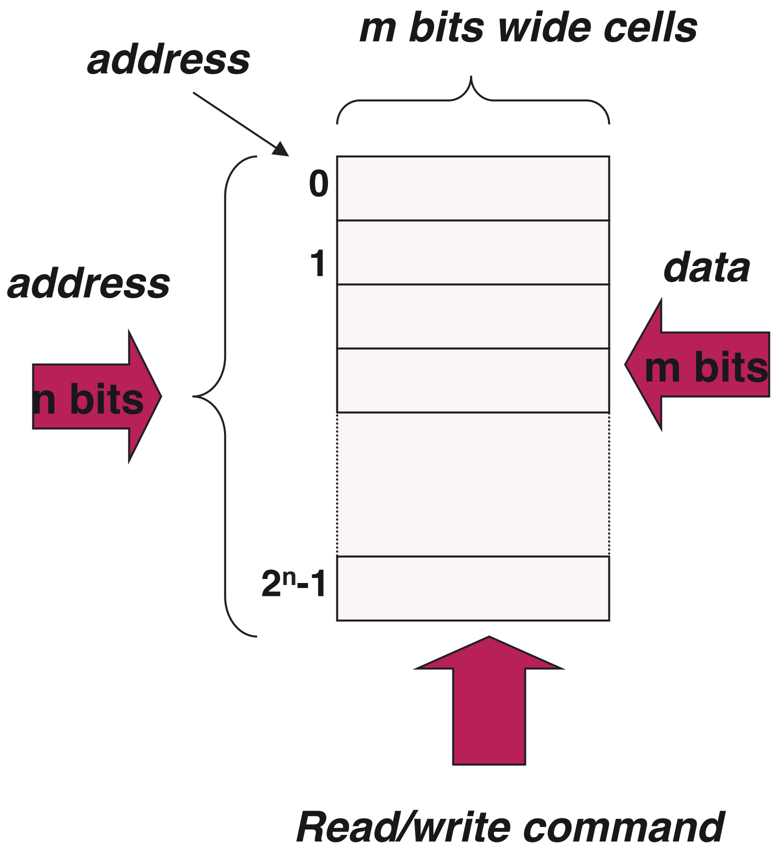
# Definition

- Memory is an array of cells each having an address.
- The memory manager is in charge of memory allocation to the operating system and to runing processes.
- Memory access (read/write):
  - Put address on address bus
  - Place read/write order on control bus
  - Wait if writing (latency)
  - Read/write data from/to data bus

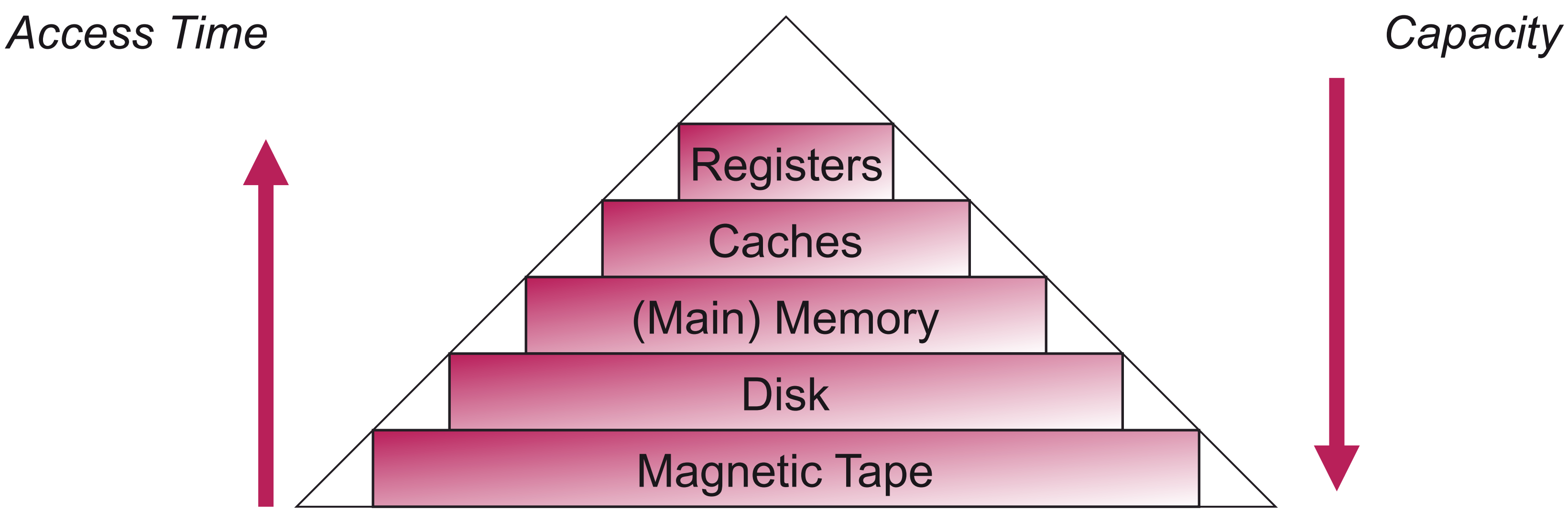


# Memory access

- If the cell has m bits (Binary digIT), each cell can encode $2^m$ values.
- m: data bus width
- n: address bus width



# Memory Hierarchy



# Memory management

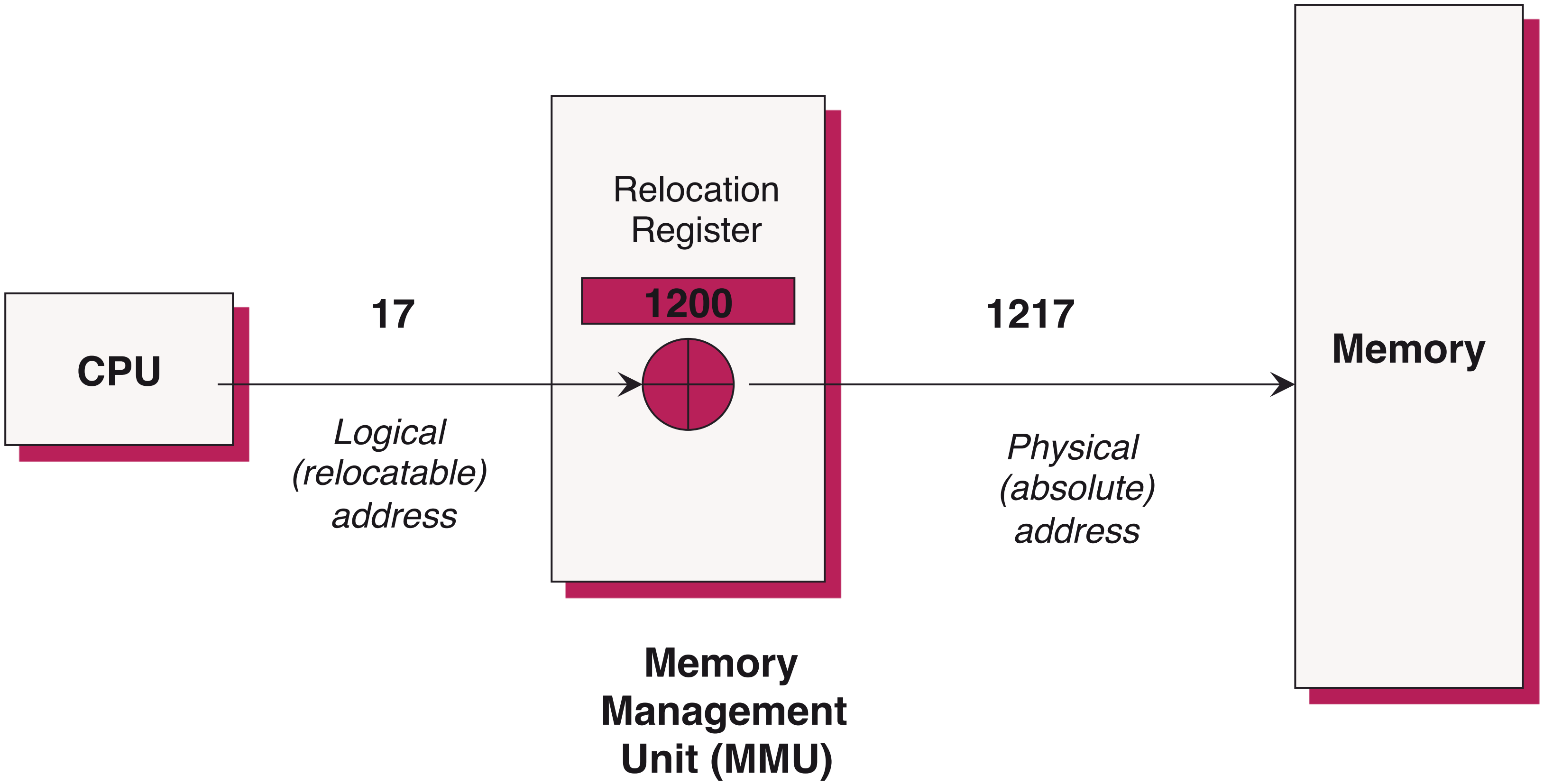
- **Logical** and **physical** memory organization
- **Loading** / **Relocation**:
  - Programmer does not know where the program will be placed in memory when executed
  - During execution, a program may be swapped to disk and returned to main memory at a different location (**relocated**)
  - Memory references must be translated to actual physical memory address
- **Protection**:
  - Processes should not be able to reference memory locations in another process without permission
  - Must be checked during execution
    - Impossible to check absolute addresses in programs since the program could be relocated
    - Operating system cannot anticipate all of the memory references a program will make
- **Sharing**:
  - Allow several processes to access the same portion of memory
  - Better to allow access to the same copy rather than have many replicas of the same program.

# Binding of Instructions and Data to Memory

Address binding of instructions and data to memory addresses can happen at three different stages.

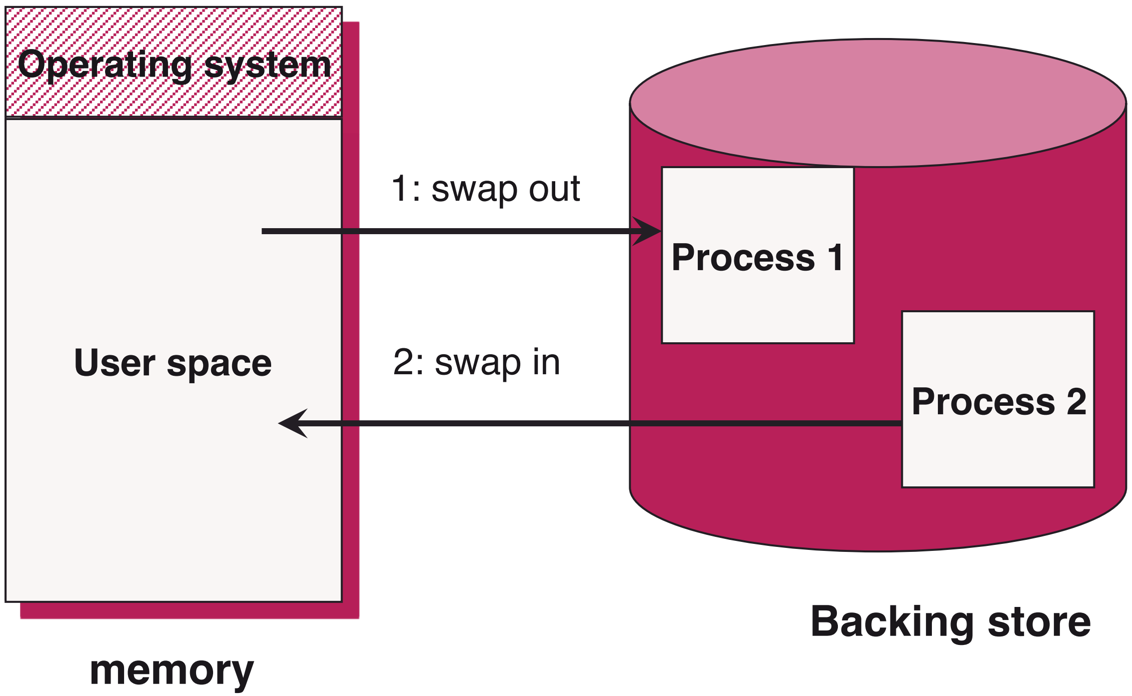
- Compile time:
  - if memory location known a priori, **absolute code** can be generated;
  - must recompile code in case of location changes
- Load time:
  - must generate **relocatable code** if memory location is not known at compile time
- Execution time:
  - binding delayed until run time if the process can be moved during its execution
  - need hardware support for address maps (e.g., base and limit registers).

# Dynamic relocation using a relocation register



# Swapping

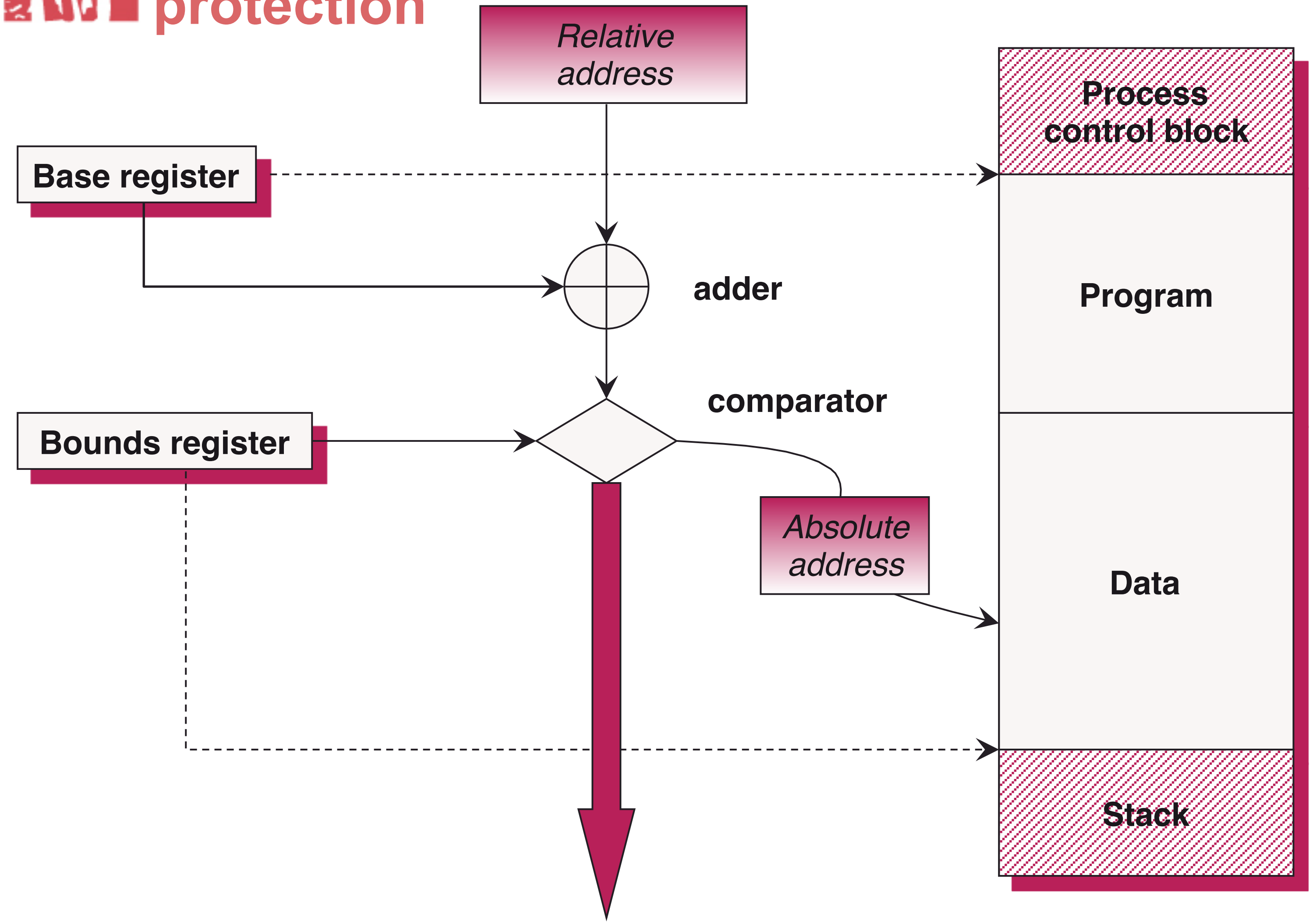
- A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution.
- Backing store
  - fast disk large enough to accommodate copies of the memory images for all users;
  - must provide direct access to these memory images.



# Basic memory organization

- Main memory usually into two partitions:
  - Resident operating system, usually held in low memory with interrupt vector.
  - User processes then held in high memory.
- Single-partition allocation
  - Relocation-register scheme used to protect user processes from each other, and from changing operating-system code and data
  - Base (or relocation) register contains value of smallest physical address
  - Bounds (or limit) register specifies the size of the range of logical addresses

# Dynamic relocation and protection




System **trap** (interrupt) if limit exceeded

# Dynamic Code Loading

- **Routine** is not loaded until it is called
- Better memory-space utilization; unused routines are never loaded.
- Useful when large amounts of code are needed to handle rarely occurring cases.
- No special support from OS
- Implemented through program design.

# Dynamic Code Linking

- Linking postponed until execution time.
- Small piece of code, *stub*, used to locate the appropriate memory-resident library routine.
- Stub replaces itself with the address of the routine, and executes the routine.
- OS needs to check if routine is already in processes' memory space. If not, the routine is **loaded**.
- Dynamic linking is particularly useful for **sharing libraries and library updates**.

# Operating systems module

## 5. Memory Management

## 5.2. Memory partitioning
## Contiguous allocation

# Memory Partitioning Schemes

- **Monoprogramming** (no swapping)
  - One process in memory at a time
- **Fixed-size partitioning**: partitioning is done in advance
  - Multiprogarmming bounded by the number of partitions (**MFT** - Multiprogramming with a Fixed number of Tasks)
  - A process must be loaded into a partition of equal or greater size => unused space in a partition cannot be claimed by another process

    => **internal fragmentation**
- **Variable-size partitioning** (**MVT** - Multiprogramming with Variable number of Tasks)
  - Leftovers may be of no use (even though there is enough total space to satisfy a request)

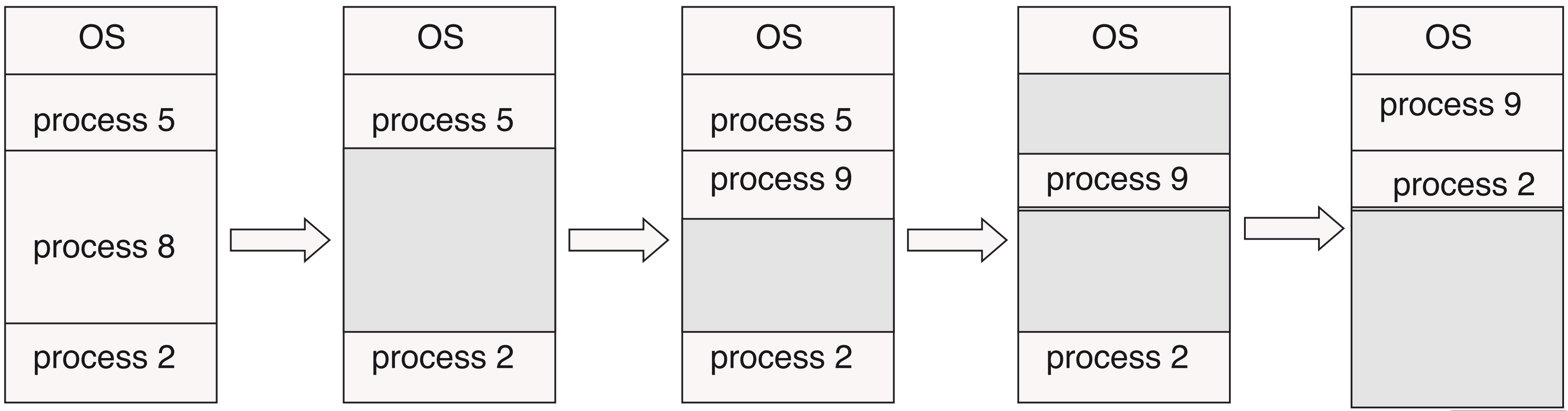
    => **external fragmentation**

# Fixed-size partitioning

- Equal-size partitions
  - any process whose size is less than or equal to the partition size can be loaded into an available partition
  - if all partitions full, the OS can swap a process out
  - a program may not fit in a partition (must be designed with overlays)
- Main memory use is inefficient.
  - Any program, no matter how small, occupies an entire partition (**internal fragmentation**)
- Placement Algorithm with Partitions
  - Equal-size partitions
    - it does not matter which partition is used
  - Unequal-size partitions
    - assign each process to the smallest partition within which it fits
    - queue for each partition
    - processes assigned to minimize wasted memory within a partition

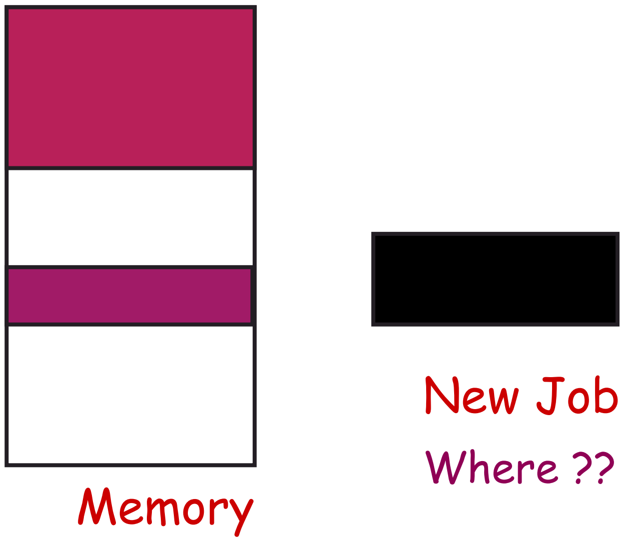
# Variable-size partitions

- Holes (blocks) of available memory of various size scattered throughout memory (**external fragmentation**)
- When a process arrives, allocate memory in a hole large enough
- OS maintains information about:
  - allocated partitions
  - free partitions (holes)
- May use **compaction** to shift processes' partitions so they are **contiguous**
  - all free memory is in one block



# Variable-size partitioning: Dynamic Storage-Allocation Strategies

- **First fit**: allocate the first hole large enough
- **Next fit**: same as first fit but start where finished last time
- **Best fit**: allocate the smallest hole big enough
- **Worst fit**: allocate largest hole



# Performance of placement strategies

- **First-fit**
  - Fastest
  - Often many processes loaded in the front end that must be searched over when trying to find a free block.
- **Next-fit**
  - Rather allocate a block of memory at the end of memory where the largest block is found and broken up into smaller blocks
  - Compaction is required to obtain a large block at the end of memory
  - Slightly worse performance than first-fit
- **Best-fit**
  - The slowest!
  - Surprisingly, also more memory wasted (external fragmentation).
- **Worst-fit**
  - As slow as best-fit
  - Worst use of memory too (largest block is typically small)

# Fragmentation

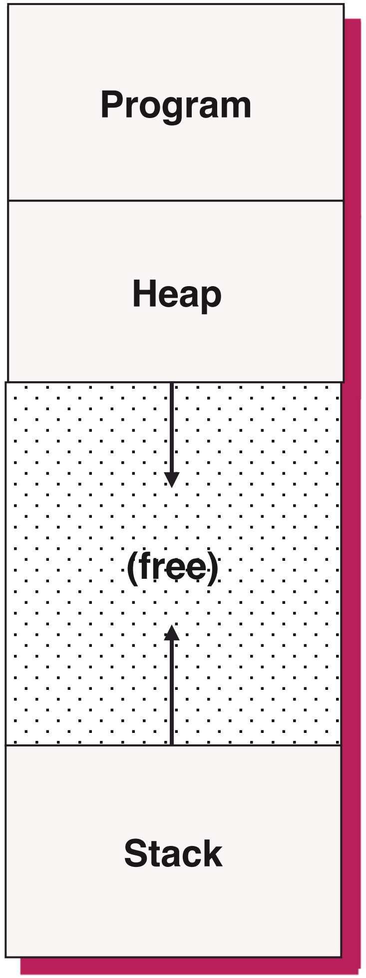
- External Fragmentation
  - total memory space exists to satisfy a request, but it is not contiguous.
- Internal Fragmentation
  - allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used.
- Reduce external fragmentation by compaction
  - Shuffle memory contents to place all free memory together in one large block.
  - Compaction possible only if relocation is dynamic, and done at execution time.

Anything else (better?) to fight external fragmentation?

- Non-contiguous address spaces: **segmentation** and **paging**

# Segmentation: generalized base-and-bounds

- Recall that address space is split into **logical segments**:
  - Program code
  - Stack (position in the function call chain and local variables)
  - Heap (dynamically allocated memory)
- Maintain a **separate** base-and-bounds register per segment
  - **Segment base** and **segment bounds**
- **Still not general enough…**



# Operating systems module

## 5. Memory Management

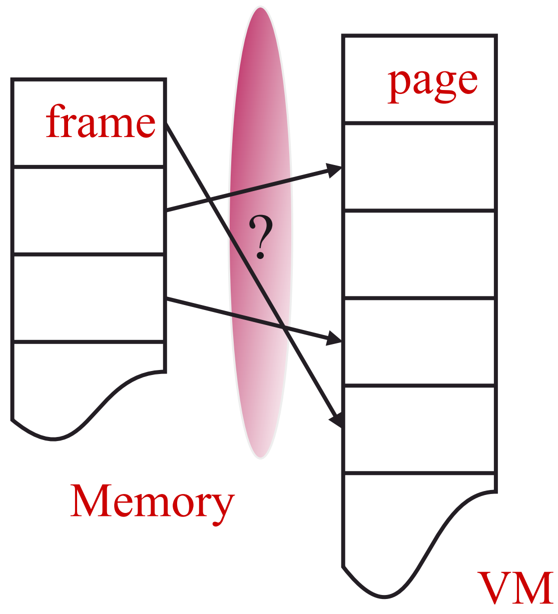
## 5.3. Memory virtualization: paging

# Virtual Memory

- **Virtual memory** is the OS abstraction that gives the programmer the illusion of an address space that may be larger than the physical address space
- Most commonly implemented using **paging**
- Motivated by:
  - Convenience:
    - No need to care about the actual amount of physical memory
  - Higher degree of multiprogramming:
    - (parts of) processes are loaded on demand

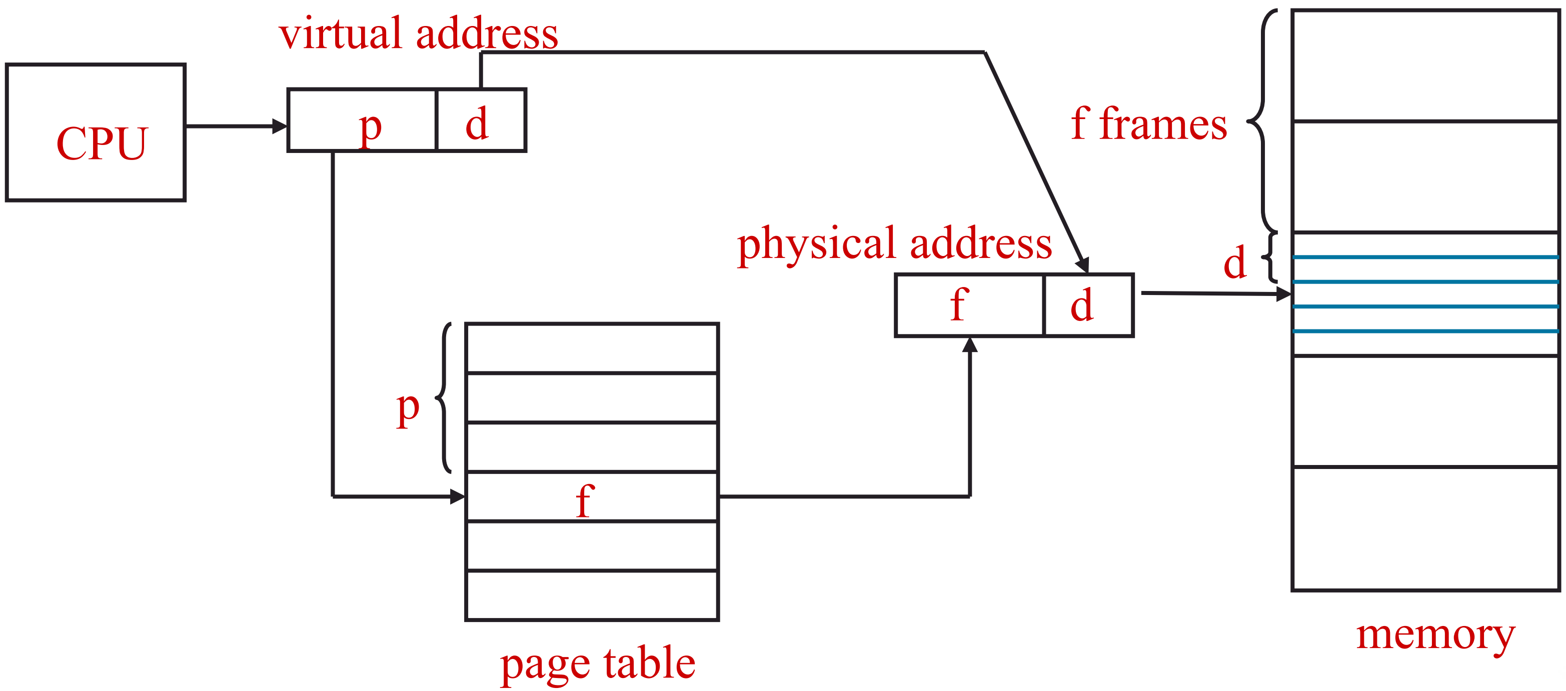
# Virtual Memory: Paging

- Physical memory divided into equal-sized **frames**, logical (virtual) memory into **pages**, power of 2 bytes (512, 1024, 8192)
- OS keeps track of all free frames.
- To run a process of n pages, need to find n free frames, **not necessarily contiguous**
- Larger pages: more internal fragmentation.
- **Page table** translates logical to physical addresses.
  - maintained by OS, one per process
  - one-to-one page-frame mapping
  - memory address = page number + offset within the page

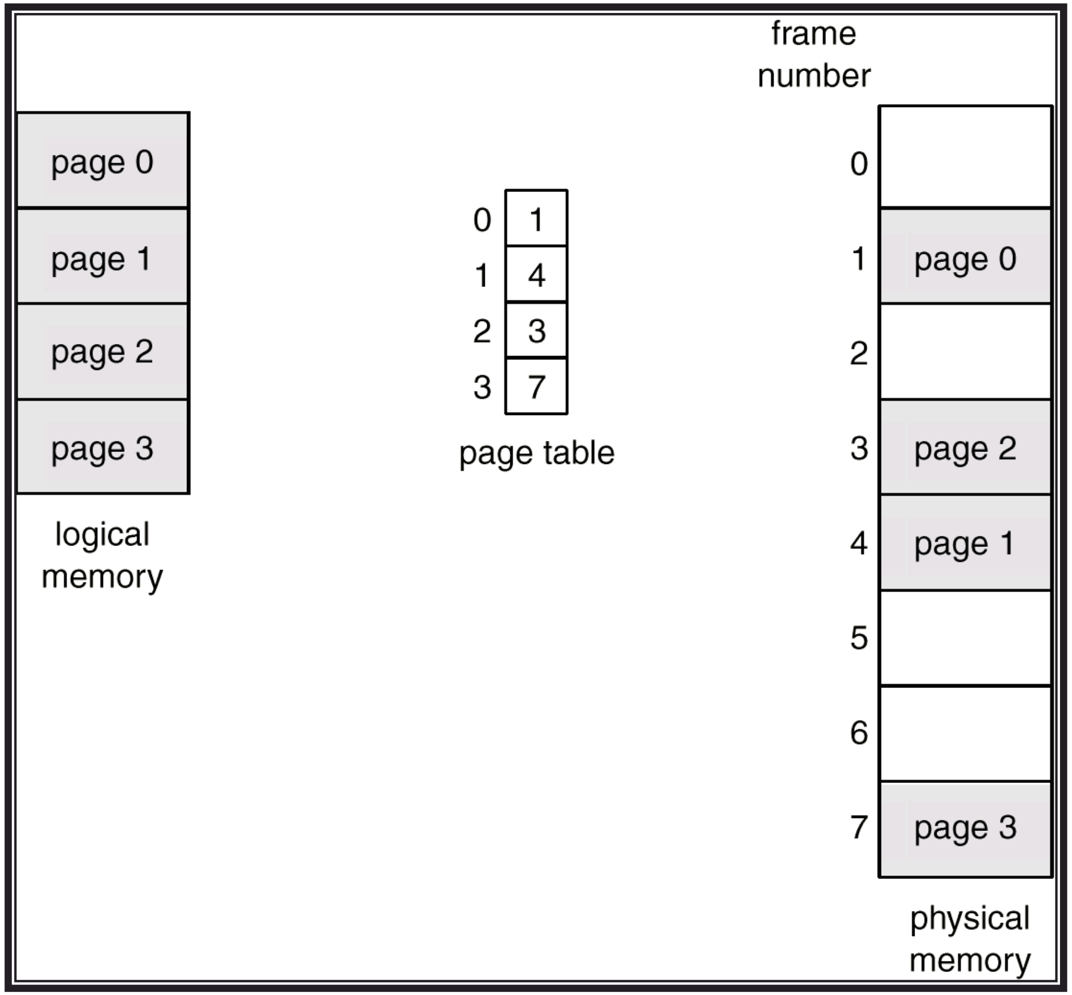


# Paging: Address Translation

- Address generated by CPU is divided into:
  - Page number (p) – used as an index into a page table which contains base address of each page in physical memory.
  - Page offset (d) – combined with base address to define the physical memory address that is sent to the memory unit.

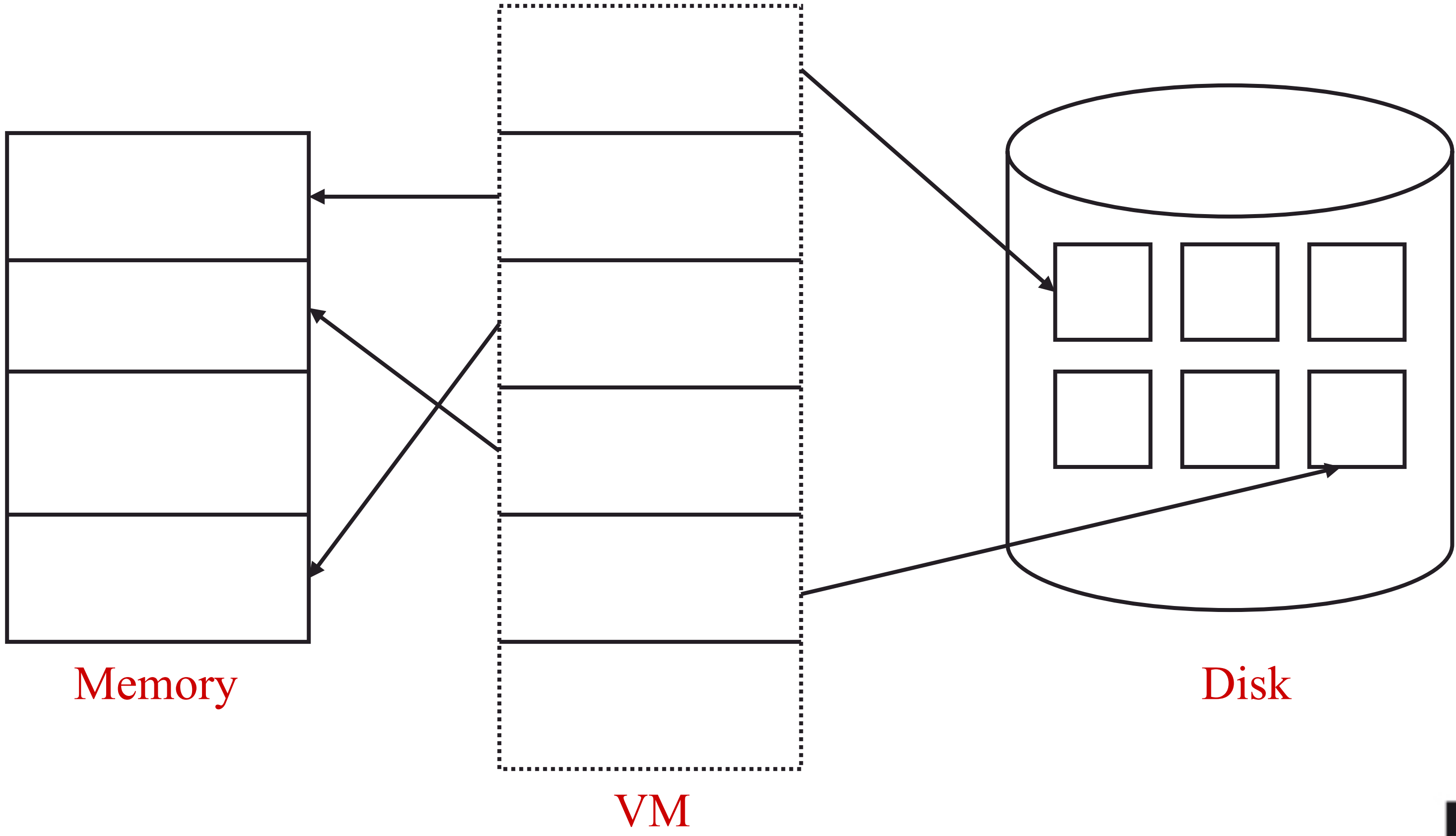


# Paging Example



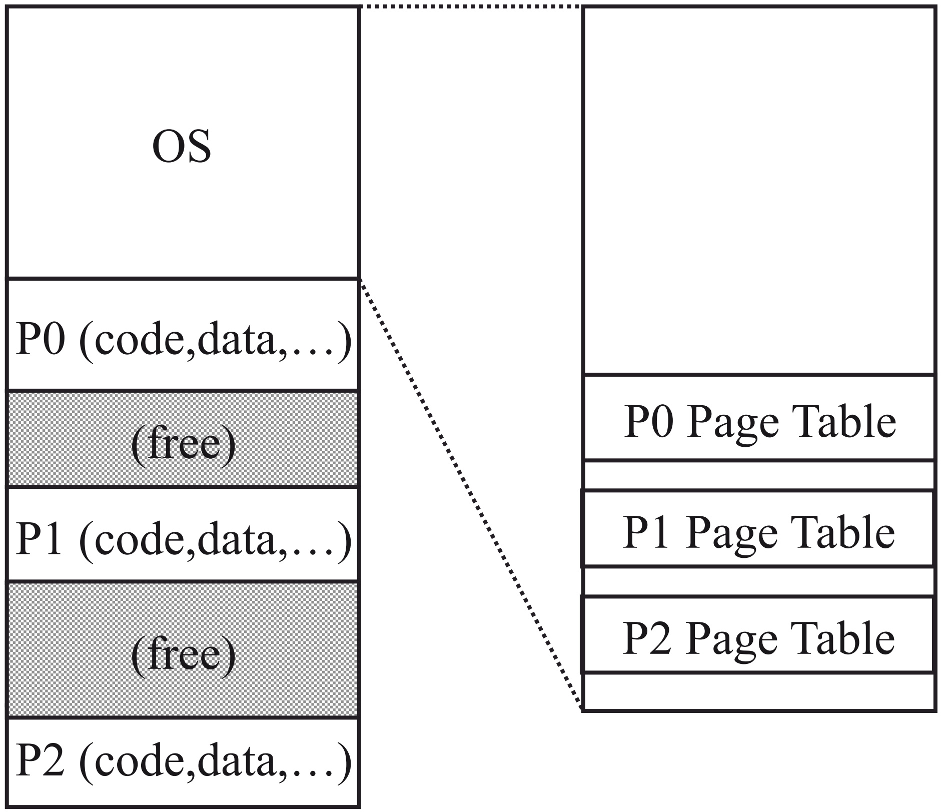
# Where to Store Address Space?

- Virtual address space may be larger than physical memory
- Where do we keep it?
  - On the next device down our storage hierarchy.



# Where to Store Page Table?

- Where do we keep the page table?
  - In memory …



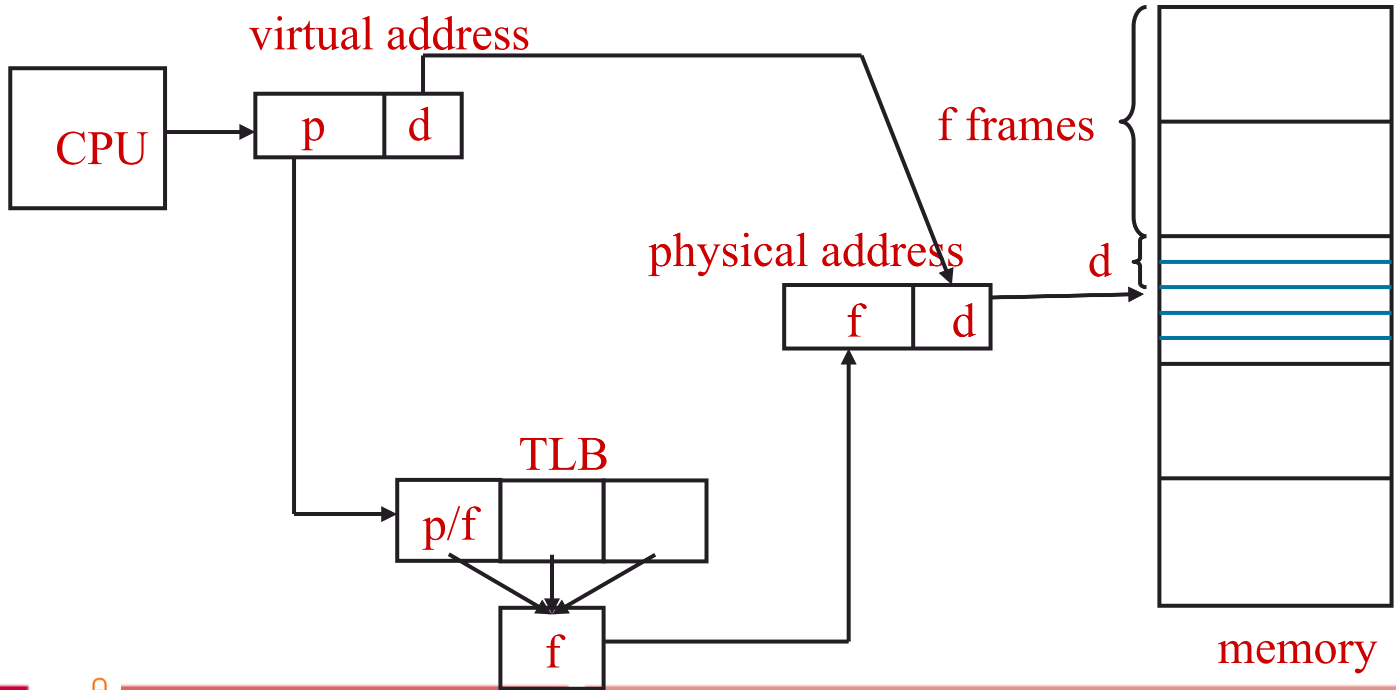
# Paging: two problems

- Page lookups take **too slow**, per memory reference:
  - extra memory reference (in the page table)
  - extra calculation to compute the physical address
  - **slows down each process by two or more!**

- Page tables (PT) grow **too big**:
  - 32-bit address space
  - 4KB ($2^{12}$) page size
  - 4 byte page-table entry (PTE)
  - Roughly 1 million ($2^{32}/2^{12}$) pages
  - 4MB page table
  - 100 processes
  - **400MB only for page tables!**

# Paging: faster translation

- Use **caching** for popular page-frame translations
  - Cache for page table entries is called the **Translation-Lookaside Buffer** (TLB)
  - Typically fully associative
  - Relatively small number of entries (e.g., 64 entries)
- On every memory access, first look for the page-frame mapping in the TLB
- Works thanks to **temporal locality**!



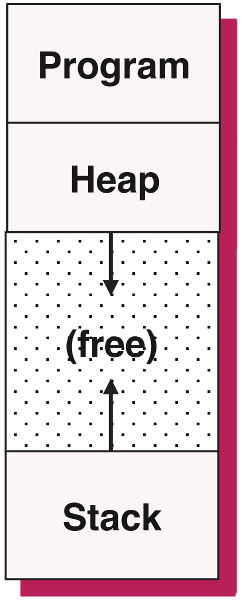
# TLB Miss

- What if the TLB does not contain the right PT entry?
  - **TLB miss**
  - Evict an existing entry if does not have any free ones
    - **Replacement policy**: Least-Recently-Used (LRU), random,…
  - Bring in the missing entry from the PT

- TLB misses can be handled in hardware or software
  - Software allows application/OS to assist in replacement decisions

# Paging: handling the page-table size

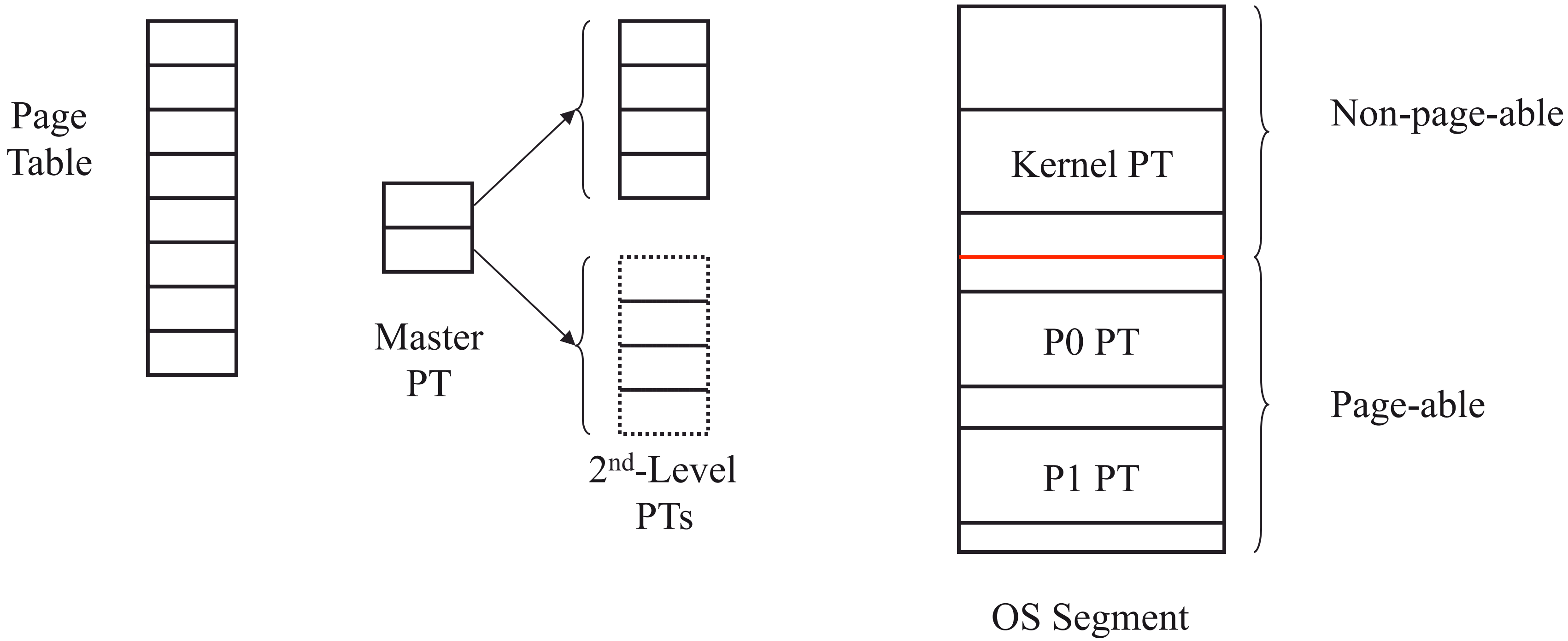
- Solution one: combine segmentation and paging
  - Address space is typically **sparsely used**
  - So why keeping the PTE in the table?
  - One page table for each segment (contiguously used) of the address space




- Solution two: **inverted page tables** (one entry per physical frame)
  - no separate page table per process
  - map each frame to (PID,page)
  - lookups are challenging: done via hash tables

- Solution three: **multi-level paging**

# Paging: two-level page tables



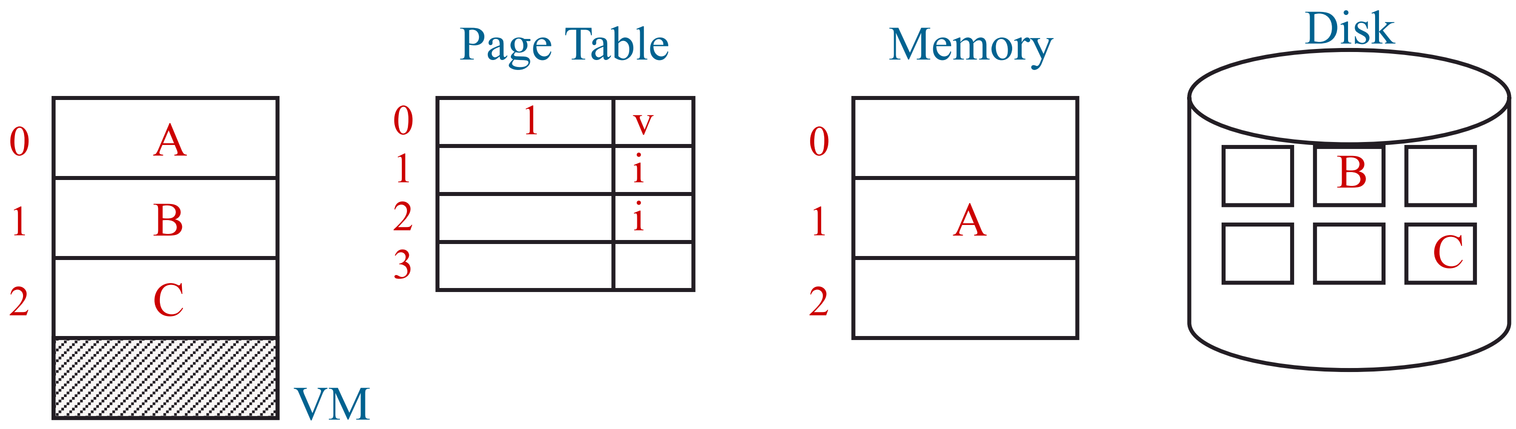

- Page the page tables: the PT is not allocated contiguously
- More efficient use of memory but two lookups per memory reference

# What if virtual memory > physical memory?

- If address space of each process is ≤ size of physical memory, then no problem
  - Still need to deal with fragmentation
- When virtual memory gets larger than physical memory
  - Partially stored in memory
  - Partially stored on disk
- Again we have to deal with misses: **page misses**

# Demand Paging

- To start a process (program), just load the code page where the process will start executing
- As process references memory (instruction or data) outside of loaded page, bring in as necessary
- How to represent fact that a page is not yet in memory?



# Page Fault

- What happens when process references a page marked as invalid in the page table?
  - **Page fault** exception
  - Check that reference is valid
  - Find a free memory frame
  - Read desired page from disk
  - Change valid bit of page to v
  - Restart instruction that was interrupted by the exception
- What happens if there is no free frame?

# Cost of Handling a Page Fault

Check page table, find free memory frame (or find victim) … about 200 - 600 μs

Disk seek and read … **about 10 ms**

Memory access … about 100 ns

- **Page fault degrades performance by a factor of 100000!!!!!**
  - And this doesn't even count all the additional things that can happen along the way
- **Avoid page faults at all cost! ☺**
  - If want no more than 10% degradation, can only have 1 page fault for every 1,000,000 memory accesses
  - And this is up to the OS: **page replacement policy**

# Page Replacement

- What if there's no free frame left on a page fault?

  Free a frame that's currently being used

  1. Select the frame to be replaced (the **victim**)
  2. Write the victim back to disk
  3. Mark the victim as invalid in the PT
  4. Read the desired page into the freed frame
  5. Mark the page as valid
  6. Restart failed instruction

- Optimization:

  do not need to write victim back if it has not been modified (check the **dirty** bit of the page)

- How do we choose the best victim in order to minimize the page fault rate?

# Optimal Page Replacement

- Suppose we only have 3 memory frames
- Suppose we know the access pattern in advance
  - 7, 0, 1, 2, 0, 3, 0, 4, 2, 3

- Optimal algorithm is to replace the page that **will not** be used for the longest period of time
  - What's the problem with this algorithm?

- Realistic policies try to predict future behavior on the basis of past behavior
  - Use of **temporal locality** again

# FIFO

- First-in, First-out
  - Be fair, let every page live in memory for about the same amount of time.
- What’s the problem?
  - Is this compatible with what we know about behavior of programs?
- How does it do on our example?
  - 7, 0, 1, 2, 0, 3, 0, 4, 2, 3

# Belady's anomaly: FIFO replacement

Assume 3 frames and the following page access sequence: **012301401234**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Frame 1: | 0 | 3 | 3 | 3 | 4 | 4 | 4 |
| Frame 2: | 1 | 1 | 0 | 0 | 0 | 2 | 2 |
| Frame 3: | 2 | 2 | 2 | 1 | 1 | 1 | 3 |
| **Page faults:** | **3** | **0** | **1** | **4** | **2** | **3** | |

**-> 9 page faults ( 3 to initially fill memory then 6)**

If 4 free frames:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Frame 1 : | 0 | 4 | 4 | 4 | 4 | 3 | 3 |
| Frame 2 : | 1 | 1 | 0 | 0 | 0 | 0 | 4 |
| Frame 3 : | 2 | 2 | 2 | 1 | 1 | 1 | 1 |
| Frame 4 : | 3 | 3 | 3 | 3 | 2 | 2 | 2 |
| **Virtual page fault:** | | **4** | **0** | **1** | **2** | **3** | **4** |

**-> 10 page faults ( 4 to initially fill memory then 6)**

More memory, more page faults !!!
- the page request *order* is an important factor
- not just the size of memory

# Fixing the anomaly: stack algorithms

- Least Frequently Used (LFU) Replacement
  - Have a reference bit (set whenever the page is referenced) and (software) counter for each page frame
  - At each clock interrupt, the OS adds the reference bit to the counter and then clears the reference bit
  - When need to evict a page, choose frame with lowest counter
  - No notion of time: may be hard to evict a page referenced long time ago
- Least Recently Used (LRU) Replacement
  - On **referencing** a page, timestamp it
  - When need to evict a page, choose the one with the oldest timestamp
    - What's the motivation here?
    - Is LRU optimal?
    - In practice, LRU is quite good for most programs
  - Is it easy to implement?

# LRU approximation:
# Second-Chance (Clock) Algorithm

- Arrange physical pages in a circle (with a "clock hand")
- Hardware keeps a **use bit (reference bit)** per page, set each time the page is referenced
  - If use bit is not set, the frame has not been used for a while
- On page fault:
  1. Advance clock hand
  2. Check ***use bit***
     - If 1, has been used recently, <u>clear</u> and go on
     - If 0, this is our victim
- Can we always find a victim? At what cost?

# LRU Approximation: Nth-Chance Algorithm

- Similar to Clock except:
  - maintain a **counter** in addition to the use bit
- On page fault:
  1. Advance clock hand
  2. Check use bit
     - If 1, clear and set counter to 0
     - If 0, increment counter, if counter < N, go on, otherwise, this is our victim
- What’s the problem if N is too large?

# Page Replacement summary

- FIFO suffers from Belady's anomaly

- LRU avoids Belady's anomaly
  - Exploits *locality of reference*

- *But* while it works well, it is hard to implement in software
  - **Aging** and various clock algorithms are the most common in practice

# References

- "Operating Systems", William Stallings, Prentice Hall, 4th ed. 2001, Chapter 7, http://williamstallings.com/OS4e.html
- Lectures notes from the text supplement by Siberschatz and Galvin, Modified by B. Ramamurthy, Chapter 8

# Operating Systems Module

## 7- Conclusion, bibliography and on-line resources

# So what did we learn?

OS is a virtual machine **and** resource manager

- Virtualization
  - CPU: processes and scheduling
  - Memory: relocation, segmentation and paging
- Concurrency
  - TASs, locks, semaphores,…
- Persistent storage
  - file system

- ‘In allocating resources, strive to avoid disaster, rather than to attain the optimum.’

  Bulter Lampson, "Hints for Computer System Design"

# Useful URLs

- See « site pédagogique » INF841
- University of Surrey, Unix tutorial: Unix for beginners": http://www.infres.enst.fr/~demeure/SiteCSIC/UNIX TUTORIAL/index.html
- Mark Burgess, A short introduction to operating systems, October 3, 2001: http://www.iu.hio.no/~mark/os/os.html
- Eric Steven Raymond, The Art of Unix Programming: http://catb.org/~esr/writings/taoup/html/
- Advanced Linux Programming: http://www.advancedlinuxprogramming.com/alp-folder"
- Memory Management Reference: www.memorymanagement.org

# Operating systems books

- Siberschatz, Galvin, Gagne. Operating Systems Concepts, 7th edition, Wiley, 2005
- Tannenbaum, *Modern Operating Systems*, 2nd edition, Prentice-Hall, 2001
- Nutt, *Operating Systems: A Modern Perspective*, 2nd edition, Addison-Wesley (2002),
  - includes material on windows
- Bach, Maurice J. The Design Of The Unix Operating System. Prentice Hall, Software Series, 1986
  - Ins and out s of linux
- Operating Systems", William Stallings, Prentice Hall, 4th ed. 2001, http://williamstallings.com/OS4e.html

# More:

- Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau, Operating Systems: Three Easy Pieces, 2013. http://pages.cs.wisc.edu/~remzi/OSTEP/
  - OS lecture notes, very much in preparation, but very easy read

- M. Herlihy, N. Shavit. The art of multiprocessor programming. Morgan Kaufman, 2008.
  - Concurrency with and without locks. Excellent combination of details and intuition.