Esercitazioni

Daniele Paolo Scarpazza
daniele.scarpazza@elet.polimi.it
www.elet.polimi.it/~scarpazz

Esercitazione n° 1
Iniziamo a sviluppare in C sotto Linux

Modificare un sorgente con `emacs`

- Lanciare `emacs`
- Funzionalità
  - Funzionalità base (movimento del cursore, modo split-screen)
  - Formattazione automatica del testo
  - Syntax highlighting
  - Demo: creare un programma di esempio in c

Compilare con `gcc`

- File di esempio: `main.c`, `reciprocal.?pp`
- Compilare un singolo file:

```
gcc -c main.c
gcc -c reciprocal.o
```

- Effettuare il linking:

```
gcc main.c reciprocal.c
        -o reciprocal
```

Automatizzare il lavoro con `make`

- Esempio di makefile:

```
reciprocal: main.o reciprocal.o
	g++ $(CFLAGS) -o reciprocal main.o reciprocal.o

main.o: main.c reciprocal.hpp
	gcc $(CFLAGS) -c main.c

reciprocal.o: reciprocal.cpp reciprocal.hpp
	g++ $(CFLAGS) -c reciprocal.cpp

clean:
	rm -f *.o reciprocal
```

Effettuare il debugging con `gdb`

- compilare con le informazioni di debugging:

  `gcc -g`

- eseguire gdb

  `gdb` *`executable`*

- Comandi di base:

  `run, break, where, print`

Documentazione in linea

- Manpages
- Info
- Header files:

  /usr/include

- Importante:

  Per le funzioni, la manpage indica header e librerie che forniscono dichiarazione e implementazione;

  Imparare a recuperare autonomamente queste informazioni;

Accesso alla Documentazione

- Linux fornisce due diversi sistemi di help relativi a comandi e funzioni C: **`man`** e **`info`**
  - **`man [`*`n`*`] `*`cmd`***
    - *n* Sezione dei manuali cui fare riferimento
    - *cmd* Comando di cui mostrare il manuale
  - **`info [`*`cmd`*` [`*`section`*`]]`**
    - *cmd* Comando di cui mostrare il manuale
    - *section* Sezione nel manuale del comando
- storicamente, `info` è arrivato dopo `man` e presenta funzionalità più avanzate;
- comparare il risultato di volta in volta;

La lista degli argomenti

```
#include <stdio.h>

int main (int argc, char* argv[])
{
    printf (“The name of this program is ‘%s’.\n”, argv[0]);
    printf (“This program was invoked with %d arguments.\n”,
              argc - 1);
    /* Were any command-line arguments specified? */
    if (argc > 1) {
           /* Yes, print them. */
          int i;
          printf (“The arguments are:\n”);
          for (i = 1; i < argc; ++i)
          printf (“ %s\n”, argv[i]);
    }
    return 0;
}
```

Uso di `getopt_long`

- Esempi di opzioni sulla riga di comando:

Short Form	Long Form	Purpose
-h	--help	Display usage summary and exit
-o filename	--output filename	Specify output filename
-v	--verbose	Print verbose messages

- Uso di getopt_long

```
next_option = getopt_long(argc, argv,
 short_options, long_options, NULL);
```

Uso di `optind` e `optarg`

- Esempio : `getopt_long.c`
- Ulteriori informazioni: `man getopt`

Standard I/O (§2.1)

- Standard file aperti: stdin, stdout, stderr;
- Stderr non è bufferizzato;
- **`fprintf(stderr,"Error: ...");`**

Uso di `assert` (§2.2)

- Assert permette la verifica a runtime di condizioni:

```
assert (pointer != NULL)
```

se non verificato, genera il messaggio d'errore:

```
Assertion ‘pointer != ((void *)0)’ failed.
```

- Come **non** va usato assert:

```
for (i = 0; i < 100; ++i)
assert (do_something () == 0);
```

- Come si sarebbe dovuto usare:

```
for (i = 0; i < 100; ++i) {
        int status = do_something ();
        assert (status == 0);
}
```

- Demo: relazione fra assert e NDEBUG

Gestione degli errori

- La maggioranza delle funzioni di sistema restituisce zero in caso di successo, e un valore non-zero in caso di errore (+/-; vedere manpage!);
- Inoltre un codice di errore è presente in **`errno`**;
- Usare **`strerror()`** per ottenere un messaggio d'errore testuale:

```
fd = open (“inputfile.txt”, O_RDONLY);
if (fd == -1) {
    fprintf (stderr, “error opening file: %s\n”,
                        strerror (errno));
    exit (1);
}
```

Esercitazione n°2
Processi e thread

Processi

- Concetto di processo e albero dei processi
- Uso del comando `ps` e `pstree`
- Process-id

  Demo: `print-pid.c`

Creare processi

- Con `system()`

  Esegue un comando; ritorna al completamento;

  usa internamente `fork()` e **`exec(`**`"/bin/sh"...`**`)`**

  Demo: `system.c`

- Con `fork()`

  il processo figlio è una copia del corrente (salvi PID e PPID); copy-on-write delle pagine;

  la chiamata restituisce il PID del figlio (0 nel figlio);

  Demo: `fork.c`

- Uso di `fork()` ed `exec()`

  Demo: `fork-exec.c`

Terminazione di processi

- Terminazione:
  - autonoma (~ `exit()`), o
  - forzata (~ `kill()`);
- Uso del comando `kill` e della funzione `kill()` per l'invio di segnali:
  - maggiori informazioni nella lezione su IPC;
  - segnali TERM(15), KILL(9) e altri (`kill -l`);
  - demo: **`kill -s KILL pid`**, **`kill -9 pid`**;
- Cleanup e Zombie
  - esempio: `zombie.c`

Attesa sulla terminazione

- Con **`wait(&status)`**
  - Attende la terminazione di un figlio qualsiasi;
  - Restituisce il pid del figlio terminato;
  - Effettua il clean-up
  - In status viene scritta l'autopsia; da interrogare con le macro apposite;
  - Demo: `fork2.c`
- Con `waitpid(pid, &status, opts)`
  - Come wait, ma attende la terminazione del solo figlio il cui PID viene indicato;
  - Demo: `fork+exec.c`

Thread

- Concetto di thread
  - stesso spazio di indirizzamento, nessuna copia
  - scheduling non prevedibile
- Libreria: libpthread (`gcc -lpthread`)
- Creazione di un thread
  - Definire: `void* func(void *)`
  - `pthread_create(&id,&attr,&func,arg)`
  - Esempio: `thread-create.c`
- Passaggio di parametri a un thread
  - Esempio: `thread-create2.c`
- Valore di ritorno

Novità sull'implementazione

- L'implementazione dei thread descritta in §4.5, pag. 92 (1 thread > 1 processo) è valida fino al kernel 2.4.18;
- Il kernel 2.4.20 e i successivi implementano la NPTL (native posix thread library), quindi N thread > 1 processo;
- Esempio: `thread-pid.c`

Terminazione di thread

- Terminazione autonoma:

  con `return` o `pthread_exit(retval)`
- Terminazione forzata (cancellazione):

  con `pthread_cancel(thread_id)`

  “cancellabilità” e sezioni critiche (vedi §4.2)
- Attesa di terminazione (join)

  `pthread_join(thread_id, &retval)`

  demo: `thread-create2.c`

Esercitazione n°3
Sincronizzazione e IPC

Sincronizzazione per thread e processi

- Sincronizzazione fra processi e IPC
  - Segnali
  - Semafori
  - Memoria condivisa
  - Mapped memory
  - Pipe
  - Socket
- Sincronizzazione fra thread e ItC
  - Variabili globali
  - Mutex
  - Semafori
  - Condition variables

Segnali (§3.3)

- Concetto di *disposizione* verso un segnale
  default, ignore, handler; vedere signal(7)
- Segnali standard (§Appendice C)
- Installazione di un handler:
  definizione: **`void handler(int i);`**
  registrazione: **`signal(SIGxxx, handler);`**
  attenzione: ri-registrarsi se desiderato!
- Invio di un segnale: **`kill(pid, SIGxx);`**
- Attesa di un segnale: **`pause();`**
- Libro: §3.3, usa **`sigaction();`**

Semafori per processi (§5.2)

- Permettono l'accesso concorrente a risorse finite, evitando le corse critiche; ex.: produttore-consumatore asincroni;
- Operazioni canoniche:
    - "wait" ~ tentativo di occupazione risorsa
        - tenta di decrementare il valore del semaforo;
        - se si scende sotto lo zero, il processo va in attesa;
    - "post" ~ liberazione risorsa
        - incrementa il valore del semaforo;
        - risveglia i processi che erano in attesa;

Semafori per processi (§5.2)

- Vanno allocati e deallocati esplicitamente
  - Con **`semget()`** e **`semctl()`**
  - Esempio: **`sem_all_deall.c`**
- Vanno inizializzati
  - Esempio: **`sem_init.c`**
- Operazioni canoniche
  - Esempio: **`sem_pv.c`**

Semafori per processi (§5.2)

- Dichiarare una union come segue (per le operazioni sul semaforo):

```
union semun {
    int val;
    struct semid_ds *buf;
    unsigned short int *array;
    struct seminfo *__buf;
};
```

- Allocare il semaforo:

```
int sem_id = semget(IPC_PRIVATE, 1,
                    O_CREAT | S_IRWXU );
```

(alloca banchi di semafori, nel caso di un solo semaforo)

Semafori per processi (§5.2)

- Inizializzazione del banco di semafori (assegnazione dei valori iniziali dei semafori; nel caso un solo semaforo nel banco):

```
unsigned short values[1];
values[0] = 1;
argument.array = values;
semctl(sem_id, 0, SETALL, argument);
```

- Distruzione del banco di semafori:

```
semctl(sem_id, 0, IPC_RMID);
```

(entrambe le chiamate agiscono su tutto il banco di semafori, quindi il secondo argomento -numero del semaforo- è ignorato)

Semafori per processi (§5.2)

Operazione “wait”

```
void wait (int semid)
{
  struct sembuf operations[1];
  operations[0].sem_num = 0;
  operations[0].sem_op = -1;
  operations[0].sem_flg = SEM_UNDO;
  if (semop (semid, operations, 1))
   fprintf(stderr,"%s\n",strerror(errno));
}
```

`semop()` esegue un numero di operazioni a piacere su un banco di semafori: per ogni operazione **`sem_num`** indica il semaforo, **`sem_op`** l'operazione e **`sem_flg`** i flags.

Semafori per processi (§5.2)

Operazione "post":

```
void post (int semid)
{
  struct sembuf operations[1];
  operations[0].sem_num = 0;
  operations[0].sem_op  = 1;
  operations[0].sem_flg = SEM_UNDO;
  if (semop (semid, operations, 1))
   fprintf(stderr,"%s\n",strerror(errno));
}
```

Semafori per processi (§5.2)

- Demo: `sem.c`
- Esemplifica il problema di sincronizzazione produttore-consumatore:
  - Il produttore lavora a ritmo costante (1 job/s);
  - Il consumatore lavora a ritmo casualizzato (1 job in 0,1,2 secondi; in media 1 job/s);
  - Sincronizzazione affidata ad un semaforo;

Memoria condivisa (§5.1)

- Segmenti di memoria che vengono collegati allo spazio di indirizzamento di più processi;
- Prima dell'uso è necessario esplicitamente:
  - Allocare un segmento condiviso con **`shmget()`**
  - Attaccare il segmento allo spazio di indirizzamento del processo corrente con **`shmat()`**;
- Dopo l'uso è necessario:
  - Distaccare il segmento con **`shmdt()`**;
  - Disallocare il segmento condiviso con **`shmctl();`**
- Demo: **`shm.c`**

Memoria condivisa (§5.1)

- Uso di **`shmget()`**:

```
segment_id = shmget(IPC_PRIVATE, shared_segment_size,
                    IPC_CREAT | IPC_EXCL | S_IRUSR |S_IWUSR);
```

- Uso di **`shmat()`**:

```
address = shmat(segment_id, desired_address, flags);
```

- Uso di **`shmdt()`**:

```
shmdt(address);
```

- Uso (banale) di **`shmctl()`**:

```
shmctl(segment_id, IPC_RMID, 0);
```

Esempio: memoria condivisa e segnali

- Demo: **`shmem+signal.c`**
- Funzionamento:
  - un processo padre, che funge da “controller”
  - due processi figli, che fungono da “worker”
  - il controller distribuisce il “job” ai worker per mezzo di una struttura conservata in memoria condivisa;
  - il controller dà inizio ai lavori inviando un segnale utente SIGUSR1 a ciascun worker;
  - i worker recuperano il job dalla memoria condivisa, lo svolgono e lasciano i risultati in memoria condivisa;
  - job: calcolo di seno e coseno di un angolo; il controller verifica la correttezza assicurandosi che $sin^2+cos^2=1$;

Esempio: memoria condivisa e segnali

- Dettagli:
  - il controller effettua due **`fork()`**, due **`kill()`** e due **`wait()`**;
  - il controller [dis/]alloca la memoria condivisa con: **`shmget()`**, **`shmat()`**, **`shmdt()`**, **`shmctl()`**;
  - ciascuno dei figli installa un handler del segnale **`SIGUSR1`** (che svolge i calcoli)
  - gli handler scrivono il risultato in memoria condivisa;
  - i figli si mettono in attesa del segnale di inizio lavori invocando un **`pause()`**;

Pipe (§5.4)

- Concetto di pipe
  - Figli con stdout>stdin, esempio: ls | less
  - Una pipe offre un buffering con capacità limitata => bloccante => sincronizzazione
- Per creare un pipe:

```
int pipe_fds[2], read_fd, write_fd;
pipe(pipe_fds);
read_fd = pipe_fds[0];
write_fd = pipe_fds[1];
```

- I file descriptor sono ereditati:
  - ideale per comunicazione padre-figlio o figlio-figlio;
  - esempio: **`pipe.c`**

Variabili condivise e corse critiche

- Esempio di problema: accesso concorrente a lista di job:

  `job-queue1.c`

- Non deve essere possibile per più thread operare contemporaneamente sulla lista;

Mutex (~ sezioni critiche) (§4.4.2)

- Inizializzazione:
```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```
- Ingresso nella sezione critica:
(potenzialmente bloccante)
```
    pthread_mutex_lock(&mutex);
```
- Uscita dalla sezione critica:
(sblocca i thread in attesa del mutex)
```
    pthread_mutex_unlock(&mutex);
```
- Esempio: `job-queue2.c`

Semafori per thread (§4.4.5)

- Inizializzazione:

```
sem_t semaphore;
sem_init(&semaphore);
```

- Operazioni canoniche:

```
sem_wait(&semaphore);
sem_post(&semaphore);
```

- Distruzione:

```
sem_destroy(&semaphore);
```

- Esempio: `job-queue3.c`

Esercitazione n°4
Socket e Filesystem

Socket (§5.5)

- Dispositivo per la comunicazione fra processi residenti sulla stessa macchina o su macchine diverse;
- Caratterizzato da:

  Spazio dei nomi:
  AF_INET, AF_UNIX, AF_IRDA, AF_APPLETALK
  ...

  Stile di comunicazione:
  SOCK_DGRAM, SOCK_STREAM, SOCK_RAW, SOCK_RDM, SOCK_SEQPACKET, ...

  Protocollo (...)

Uso dei socket (§5.5.5, §5.5.6)

- Lato server

```
socket_fd = socket(PF_xxx, SOCK_DGRAM,0);
bind(socket_fd, ..., ...);
listen(socket_fd, 5);
while(...) {
    client_sock_fd = accept(socket_fd, ...);
    ... write(client_socket_fd,...);
    ... read(client_socket_fd, ...);
    close(client_sock_fd);
}
```

- Lato client

```
socket_fd = socket(PF_xxx, SOCK_DGRAM,0);
connect(socket_fd, ..., ...);
... write(socket_fd,...);
... read(socket_fd, ...);
close(client_sock_fd);
```

Socket locali e socket internet

- Socket locali

  Esempio: **`socket-server.c, socket-client.c`**

- Socket TCP/IP

  Maggiori informazioni nella prossima esercitazione

  Esempio: **`socket-inet.c`**

Socket locale – server - inizializzazione

```
const char* const   socket_name = argv[1];
int                 socket_fd;
struct sockaddr_un  name;

/* Create the socket.  */
socket_fd = socket(PF_LOCAL, SOCK_STREAM, 0);

/* Indicate this is a server.  */
name.sun_family = AF_LOCAL;
strcpy (name.sun_path, socket_name);
bind(socket_fd, &name, SUN_LEN (&name));

/* Listen for connections.  */
listen(socket_fd, 5);
```

Socket locale – server - connessioni

```
do {
  struct sockaddr_un client_name;
  socklen_t client_name_len;
  int client_socket_fd;

  client_socket_fd =
       accept(socket_fd, &client_name, &client_name_len);

  /* Handle the connection.  */
  client_sent_quit_message =
       server (client_socket_fd);
  /* Close our end of the connection.  */
  close (client_socket_fd);
}
while (!client_sent_quit_message);

close (socket_fd);
unlink (socket_name);
```

Socket locale – server – ricezione dati

```
int server (int client_socket)
{
  while (1) {
    int length;
    char* text;
    /* Read length of the text message;  */
    if (read(client_socket, &length, sizeof (length)) == 0)
      return 0;
    /* Allocate a buffer to hold the text.  */
    text = (char*) malloc (length);
    /* Read the text itself, and print it.  */
    read(client_socket, text, length);
    printf ("%s\n", text);

    free (text);
    if (!strcmp (text, "quit")) return 1;
  }
}
```

Socket locale – client - connessione

```
int main (int argc, char* const argv[])
{
  const char* const   socket_name = argv[1];
  const char* const   message     = argv[2];
  int                 socket_fd;
  struct sockaddr_un  name;

  socket_fd = socket(PF_LOCAL, SOCK_STREAM, 0);
  /* Store the server's name in the socket address.  */
  name.sun_family = AF_LOCAL;
  strcpy (name.sun_path, socket_name);
  /* Connect the socket.  */
  connect(socket_fd, &name, SUN_LEN (&name));
  /* Write the text on the command line to the socket.  */
  write_text (socket_fd, message);
  close(socket_fd);
  return 0;
}
```

Socket locale – client – trasmissione dati

```
void write_text (int socket_fd, const char* text)
{
  /* Write the number of bytes in the string, including
     NUL-termination.  */
  int length = strlen (text) + 1;

  write (socket_fd, &length, sizeof (length));
  /* Write the string.  */
  write (socket_fd, text, length);
}
```

Socket TCP/IP – esempio di client

```
int                        socket_fd;
struct sockaddr_in         name;
struct hostent *           hostinfo;

socket_fd = socket(PF_INET, SOCK_STREAM, 0);
/* Store the server's name in the socket address.  */
name.sin_family = AF_INET;
/* Convert from strings to numbers.  */
hostinfo = gethostbyname(argv[1]);
if (hostinfo == NULL) { /* errore */ return 1; }
name.sin_addr = *((struct in_addr*)hostinfo->h_addr);
name.sin_port = htons (80);
/* Connect to the web server  */
if (connect(socket_fd, &name, sizeof(struct sockaddr_in))
== -1) { /* errore */ return 1; }
get_home_page (socket_fd);
```

Socket TCP/IP – client – scambio dati

```
void get_home_page (int socket_fd)
{
  char buffer[10000];
  ssize_t number_characters_read;

  sprintf (buffer, "GET /\n");
  write(socket_fd, buffer, strlen (buffer));
  while (1) {
    number_characters_read = read(socket_fd, buffer,10000);
    if (number_characters_read == 0)  return;
    /* Write the data to standard output.  */
    fwrite(buffer, 1, number_characters_read, stdout);
  }
}
```

Filesystem

- Il **File System** fornisce all'utente una *visione logica* di tutti i dispositivi: hard disk, CD, floppy, ramdisk, mouse, scanner, porte varie, ...
- La struttura di un file system si basa su due concetti fondamentali: file e directory
- Sistemi operativi diversi utilizzano diverse implementazioni di file system

File

- Ad un file sono associati gli attributi:

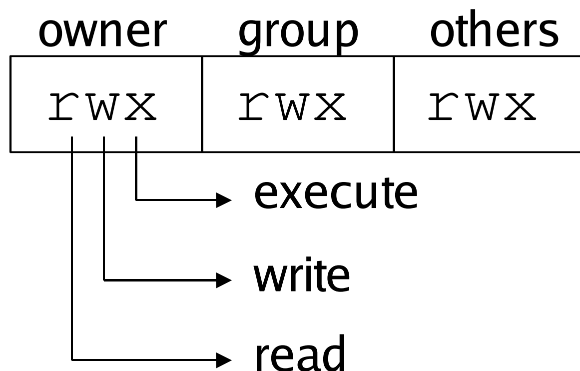
Nome:	E' un nome simbolico con cui ci si riferisce ad esso, può contenere una estensione che ne indica il tipo
Locazione:	E' un puntatore alla posizione fisica dei dati sul dispositivo
Dimensione	Dimensione dei dati
Date:	Indicano il momento della creazione, ultima modifica o ultimo accesso al file
Proprietari:	L'utente e il gruppo che possiedono il file
Diritti:	Indica quali operazioni possono essere eseguite su quel file

Protezione: diritti

- I **diritti** sono formati da 3 gruppi di 3 bit:

  In ciascun gruppo:
  - il primo bit (r) indica se il file può essere letto
  - il secondo bit (w) indica se il file può essere scritto
  - il terzo bit (x) indica se il file può essere eseguito

  i tre gruppi:
  - il primo gruppo (u) indica i diritti dell'utente proprietario
  - il secondo (g) i diritti degli appartenenti al gruppo proprietario
  - il terzo (o) indica i diritti di tutti gli altri utenti



owner	group	others
rwx	rwx	rwx
111	101	101
7	5	5

Filesystem e Dispositivi

Un device è un file speciale che fornisce una interfaccia comune a diversi dispositivi o dati. I device sono raccolti in `/dev` e nelle sue sottodirectory:

`stdin, stdout, stderr`	Standard input, output, error
`ttyX, ptyX, console`	Terminale fisico/virtuale *X*, terminale corrente
`null`	Null device (pozzo senza fondo)
`fd0, fd1,...`	Primo, secondo floppy disk, ...
`hda, hdb, hdc, hdd`	Dischi IDE: primary master, primary slave, secondary master, secondary slave
`hda0, hda1, hda2, ...`	Prima, seconda, terza, partizione sul disco hda
`psaux`	Mouse di tipo “PS/2”
`sda, sdb, sdc, ...`	Primo, secondo, terzo disco SCSI
`sda1, sda2, sda3, ...`	Prima, seconda, terza partizione sul disco sda
`scd0, scd1, ...`	Primo, secondo, ... lettore CD SCSI
`sg0, sg1, ...`	Primo, secondo, ... dispositivo generico SCSI
`ttyS0, ttyS1, ...`	Prima, seconda, ... porta seriale

Il filesystem `procfs`

Attraverso `/proc` è possibile accedere a numerose informazioni sullo stato corrente del sistema, esempio:

`/proc/cpuinfo`	contiene informazioni sulla CPU;
`/proc/devices`	major e minor di ogni dispositivo;
`/proc/tty/driver/serial`	stato della seriale;
`/proc/sys/kernel/version`	versione del kernel;
`/proc/sys/kernel/hostname`	nome di rete della macchina;
`/proc/filesystem`	tipi di filesystem noti al kernel;
`/proc/ide/ide1/hdc/media`	dispositivo IDE sec/slave;
`/proc/ide/ide1/hdc/model`	marca e modello del “ “ ;
`/proc`	contiene una sottodirectory per

ogni processo: raccoglie informazioni sul processo (immagine dell’eseguibile, stato, memoria, paginazione, directory di lavoro, descrittori file aperti, thread);

Esercitazione n°5
Una applicazione completa

Panoramica dell'applicazione (§11.1)

- Si tratta di un web server minimale
- Serve pagine web dinamiche, generate al momento
- Ad ogni richiesta viene caricato “al volo” il modulo corrispondente
- Vengono forniti come esempio quattro moduli (time, issue, diskfree, processes);
- Demo: compilazione e uso (§11.4)

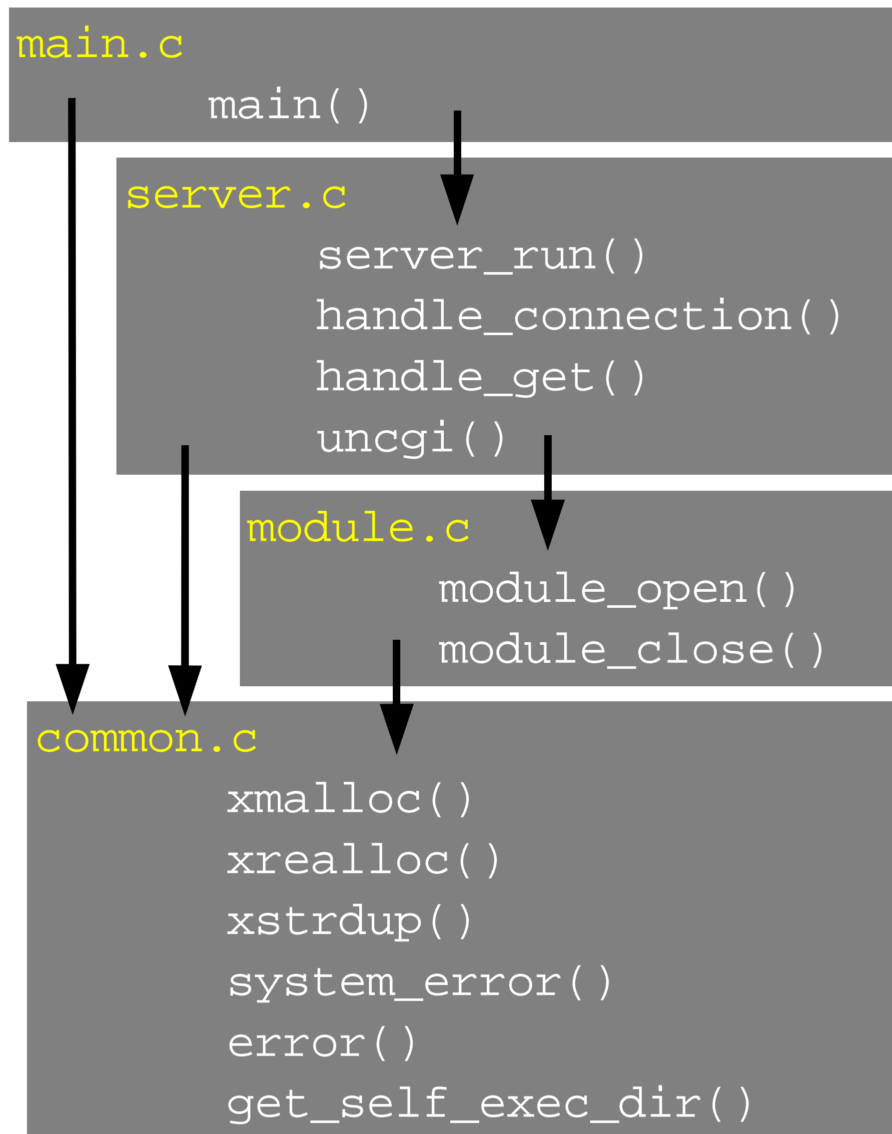
Implementazione del server (§11.2)

- Funzioni comuni
  ([ri]allocazione sicura, errori, percorso)
  File: `common.c`
- Caricamento dinamico moduli (cfr. §2.3.6)
  File: `module.c`
- Server
  (socket, fork, gestione connessioni e richieste)
  File: `server.c`
- Main
  (parsing opzioni, lancio server)
  File: `main.c`

Moduli d'esempio disponibili (§11.3)

- Modulo “orologio”
  File: `time.c`
- Modulo “informazioni sulla distribuzione”
  File: `issue.c`
- Modulo “spazio su disco”
  File: `diskfree.c`
- Modulo “processi”
  File: `processes.c`

Implementazione del server (§11.3)



File `common.c`

- Allocazione sicura della memoria:

```
void* xmalloc(size_t size)
void* xrealloc(void* ptr, size_t size)
char* xstrdup(const char* s)
```

- Visualizzazione di errori fatali:

```
void system_error(const char* operation)
void error(const char* cause, const
char* message)
```

- Altro:

```
char* get_self_executable_directory()
```

File `module.c`

```
struct server_module* module_open (const char* module_name)
{ /*...*/
  void (* module_generate) (const char*, int);
  struct server_module* module;

  module_path = (char*) xmalloc(strlen(module_dir)+strlen(module_name)+2);
  sprintf(module_path, "%s/%s", module_dir, module_name);
  handle = dlopen(module_path, RTLD_NOW);
  module_generate = (void (*) (const char *, int))
             dlsym(handle, "module_generate");
  module = (struct server_module*) xmalloc(sizeof (struct server_module));
  module->handle             = handle;
  module->name               = xstrdup (module_name);
  module->generate_function = module_generate;
  return module;
}

void module_close (struct server_module* module)
{
  dlclose (module->handle);
  free ((char*) module->name);
  free (module);
}
```

File `server.c`

```
void server_run (struct in_addr local_address, uint16_t port)
{
  sigaction(SIGCHLD, &sigchld_action, NULL);
  server_socket = socket(PF_INET, SOCK_STREAM, 0);
  bind(server_socket, &socket_address, sizeof (socket_address));
  rval = listen(server_socket, 10);
  ...
  while (1) {
  ...
  connection = accept(server_socket, &remote_address, &address_length);
  child_pid = fork();
  if (child_pid == 0) {
      close(STDIN_FILENO);
      close(server_socket);
      handle_connection(connection);
      close(connection);
      exit (0);
    }
    else if (child_pid > 0) {
      close (connection);
    }
  }
}
```

File `server.c`

```
static void handle_connection (int connection_fd)
{
  char buffer[256];
  ssize_t bytes_read;

  bytes_read = read(connection_fd, buffer, sizeof (buffer) - 1);
  if (bytes_read > 0) {
    buffer[bytes_read] = '\0';
    sscanf (buffer, "%s %s %s", method, url, protocol);
    while (strstr (buffer, "\r\n\r\n") == NULL)
      bytes_read = read(connection_fd, buffer, sizeof (buffer));
    if (strcmp(protocol, "HTTP/1.0") && strcmp(protocol, "HTTP/1.1")) {
      write(connection_fd, bad_request_respon, sizeof(bad_request_respon));
    }
    else if (strcmp (method, "GET")) {
      snprintf (response, sizeof (response), bad_method_template, method);
      write(connection_fd, response, strlen (response));
    } else
      handle_get (connection_fd, url);
  }
  else if (bytes_read == 0) ;
  else system_error ("read");
}
```

File `server.c`

```
static void handle_get (int connection_fd, const char* page)
{
  if (*page == '/' && strchr (page + 1, '/') == NULL) {
    char module_file_name[256];

    if (parameters = strchr(page, '?')) {
        parameters[0] = 0;
        parameters++;
    }
    uncgi(parameters);
    snprintf (module_file_name, sizeof(module_file_name), "%s.so", page+1);
    module = module_open (module_file_name);
  }

  if (module == NULL) {
    snprintf (response, sizeof (response), not_found_template, page);
    write(connection_fd, response, strlen (response));
  } else {
    write(connection_fd, ok_response, strlen (ok_response));
    (*module->generate_function) (parameters,connection_fd);
    module_close(module);
  }
}
```

File `main.c`

```
int main (int argc, char* const argv[])
{
  module_dir = get_self_executable_directory ();
  do {
    next_option = getopt_long(argc,argv, short_options, long_options, NULL);
    switch (next_option) {
    case 'a':
      local_host_name = gethostbyname(optarg);
      local_address.s_addr = *((int*) (local_host_name->h_addr_list[0]));
      break;
    case 'h':  print_usage (0);
    case 'm':  module_dir = strdup (optarg);          break;
    case 'p':  port = (uint16_t) htons (value);       break;
    case 'v':  verbose = 1;                           break;
    case '?':  print_usage (1);
    case -1:   /* Done with options.  */              break;
    default:   abort ();
    }
  } while (next_option != -1);
  server_run (local_address, port);
  return 0;
}
```

Modulo di esempio `time.c`

- Genera una pagina HTML con:
  - data e ora corrente;
  - un form HTML che richiama la stessa pagina con metodo GET, contenente:
    - un drop-down che permette di cambiare il formato;
    - un pulsante "submit";
- Il formato di data e ora viene:
  - estratto dalla URL;
  - le stringhe CGI vengono convertite;
  - passato a `strftime()`

Elaborati per l'esenzione dalle domande

- Per ottenere l'esenzione dalle due domande relative all'esercitazione è possibile svolgere un elaborato.
- Regole e modalità sono descritte nel file `RegoleALP.pdf` sul sito del Prof. Fornaciari.
- Decisione da comunicare entro 15 gg prima dell'appello (~ 20 gennaio 2bc).
- Consegna entro 10 gg dall'appello.

Esempi di elaborati possibili:

- Gestione dei processi
  - lista processi; pstree; kill; suspend; resume; dettagli
- Gestione della posta elettronica su POP3:
  - connetti; lista messaggi; visualizza; elimina; disconnetti
- Gestione degli utenti locali:
  - lista utenti; dettagli; crea; elimina; chsh; chfn; last; passwd
- ...

Esempi di elaborati possibili:

- Gestione delle code di stampa:
  - lista stampanti; status stampante; lista jobs; dettagli job; kill job
- Gestione delle quote su disco:
  - report; dettagli utente; modifica dettagli utente
- Informazioni di sistema:
  - un ragionevole subset di procfs + last + df + ...
- Gestione di un album fotografico:
  - richiede estensione del server
- ...

Esempi di elaborati possibili:

- Gestione dei servizi in stile SystemV:
  vedere funzionalità di chkconfig sotto RedHat;
- Gestione del servizio samba:
  vedere funzionalità samba;

  Usare Webmin (www.webmin.com)
  come fonte di ispirazione per:
  - le funzionalità da fornire;
  - altri temi di elaborato possibili;