Modulo 5: System call per il controllo dei processi (fork, exit, wait)

Laboratorio di Sistemi Operativi I
Anno Accademico 2007-2008

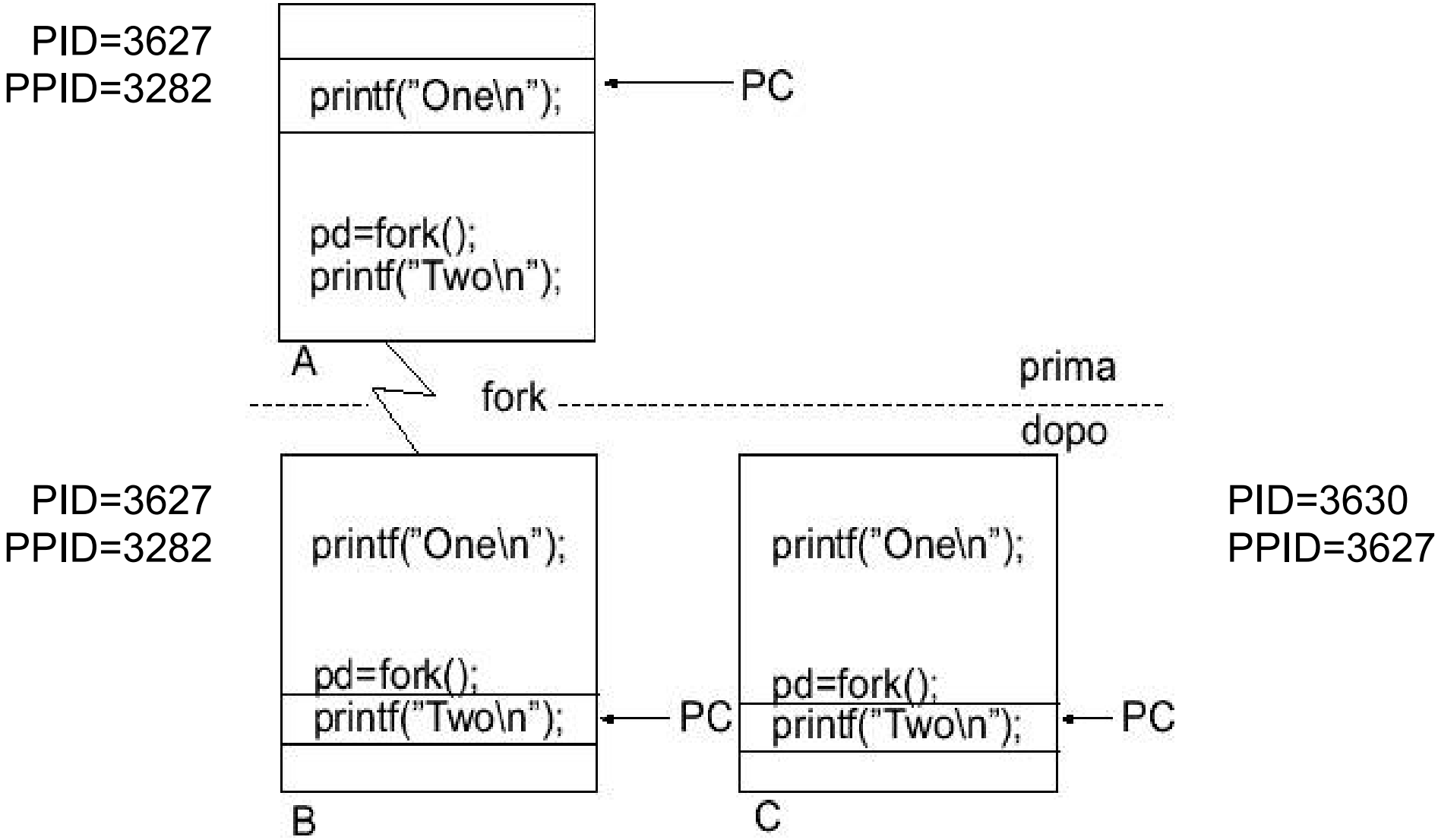
Francesco Pedullà
(Tecnologie Informatiche)

Massimo Verola
(Informatica)

Creazione di processi

- `pid_t fork();`
  crea un nuovo processo figlio, copiando completamente l'immagine di memoria del processo padre:
  - data, heap, stack vengono copiati
  - per ottimizzare le prestazioni, il codice viene spesso condiviso e in alcuni casi, si esegue *copy-on-write*, cioè le pagine di memoria del processo padre vengono effettivamente copiate solo al momento in cui sono modificate
- Sia il processo figlio che il processo padre continuano ad eseguire l'istruzione successiva alla **`fork`**
- **`fork`** viene chiamata una volta dal processo padre, ma ritorna due volte
  - al processo figlio ritorna con return code=0
  - al processo padre ritorna il PID del processo figlio
  - in caso di errore ritorna un valore negativo

Creazione di processi



Esempio di utilizzo di fork

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
main()
{
    pid_t pid;
    printf("Un solo processo con PID %d.\n",(int)getpid());
    printf("Chiamata a fork...\n");
    pid=fork();
    if(pid == 0)
        printf("Sono il processo figlio (PID: %d).\n",(int)getpid());
    else if(pid>0)
        printf("Sono il genitore del processo con PID %d.\n",pid);
    else
        printf("Si e' verificato un errore nella chiamata a fork.\n");
}
```

Proprietà ereditate dal processo figlio

- real user ID, real group ID, effective user ID, effective group ID
- group IDs supplementari
- ID del gruppo di processi
- session ID
- terminale di controllo
- set-user-ID flag e set-group-ID flag
- directory corrente
- directory root
- maschera di creazione file (umask)
- maschera dei segnali
- flag close-on-exec per tutti i descrittori aperti
- environment
- segmenti di memoria condivisi (*shared memory segments*)
- limiti sulle risorse

Proprietà NON ereditate dal processo figlio

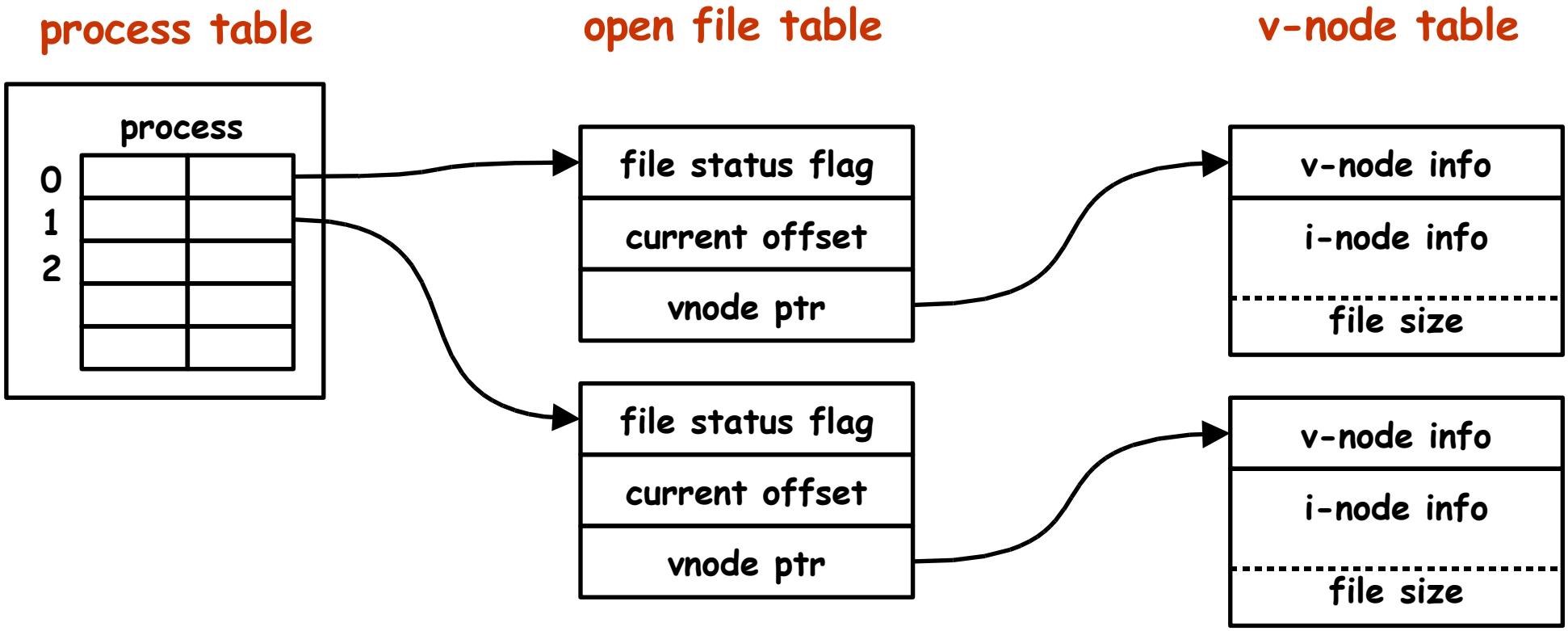
- valore di ritorno di **`fork`**
- process ID
- process ID del processo parent
- valori di timings del processo
- file locks
- allarmi in attesa (annullati nel figlio)
- insieme di segnali in attesa (svuotato)

Errori nella `fork`

- **Ragioni per il fallimento di `fork`**
  - il numero massimo di processi nel sistema è stato raggiunto
  - il numero massimo di processi per user id è stato raggiunto
- **Utilizzo di `fork`**
  - quando un processo vuole duplicare se stesso in modo che padre e figlio eseguano parti diverse del codice, ad esempio:
    - *network servers (demoni)*:
      il padre è sempre in attesa di richieste dalla rete;
      quando una richiesta arriva, viene assegnata ad un figlio,
      mentre il padre può tornare immediatamente ad attendere una nuova richiesta
  - quando un processo vuole eseguire un programma diverso (utilizzo della chiamata **`exec`**)

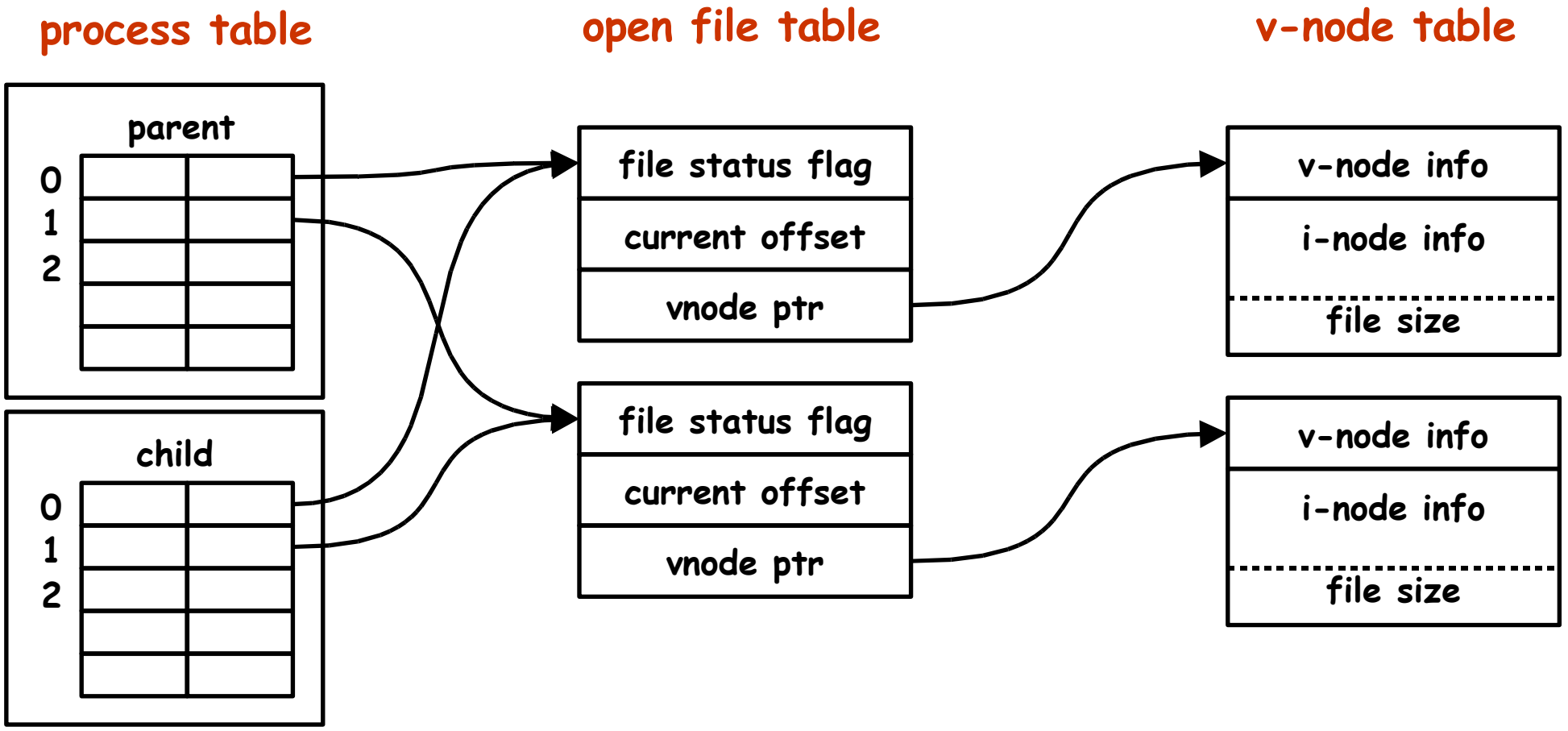
Relazione tra `fork` e file aperti (I)

- Una caratteristica della chiamata **`fork`** è che tutti i descrittori che sono aperti nel processo parent sono duplicati nel processo child
- Prima della **`fork`**:



Relazione tra `fork` e file aperti (II)

- Dopo la **`fork`**:



Relazione tra `fork` e file aperti (III)

- E' importante notare che padre e figlio condividono lo stesso *file offset*
- Consideriamo il seguente caso:
  - un processo esegue **`fork`** e poi attende che il processo figlio termini (system call **`wait`**)
  - supponiamo che lo **`stdout`** sia rediretto ad un file, e che entrambi i processi scrivano su **`stdout`**
  - se padre e figlio non condividessero lo stesso offset, avremmo un problema:
    - il figlio scrive su **`stdout`** e aggiorna il proprio current offset
    - il padre sovrascrive **`stdout`** e aggiorna il proprio current offset

Come gestire i descrittori di file

- **Caso 1: il processo padre aspetta che il processo figlio termini**
  - in questo caso, i file descriptor vengono lasciati immutati
  - eventuali modifiche ai file fatte dal processo child verranno riflesse nella file table entry e nella v-node entry del processo figlio
- **Caso 2: i processi padre e figlio sono indipendenti**
  - in questo caso, ognuno chiuderà i descrittori non necessari e proseguirà opportunamente

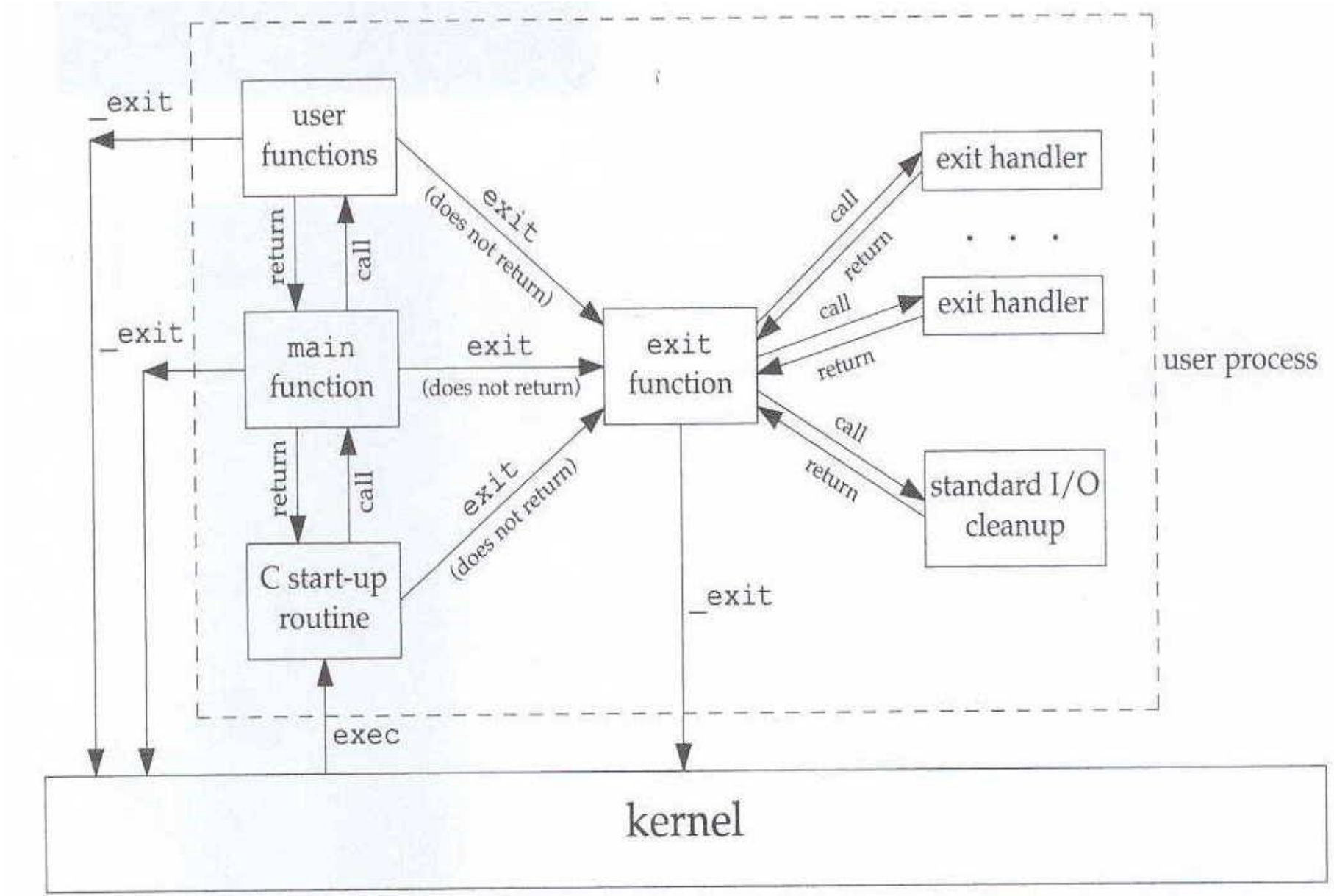
Esercizi

- Scrivere un programma C che apre un file, effettua una **`fork`** e scrive messaggi diversi sul file a seconda che sia padre o figlio. Come si alternano i messaggi nel file?
- Nel programma prima descritto spostare la open dopo la **`fork`** e verificare se il contenuto del file è cambiato rispetto al programma precendete e spiegarne il perchè.

Terminazione di processi (I)

- **Esistono tre modi per terminare in modo NORMALE:**
  - eseguire un return da main (è equivalente a chiamare **`exit`**)
  - chiamare la funzione **`exit`**:
    - **`void exit(int status);`**
      - invoca di tutti gli exit handlers che sono stati registrati
      - chiude di tutti gli I/O stream standard
      - è specificata in ANSI C
  - chiamare la system call **`_exit`**:
    - **`void _exit(int status);`**
      - ritorna al kernel immediatamente
      - è chiamata come ultima operazione da **`exit`**
      - è specificata nello standard POSIX.1

Terminazione di processi (II)



Terminazione di processi (III)

- **Esistono due modi per terminare in modo ANORMALE:**
  - Quando un processo riceve certi segnali
    - generati dal processo stesso
    - generati da altri processi
    - generati dal kernel
  - Chiamando **`abort:`**
    - **`void abort();`**
    - La chiamata ad **`abort`** costituisce un caso speciale del primo caso dei tre sopra elencati, in quanto genera il segnale **`SIGABRT`**

NOTA: per informazioni sui segnali usa **`man 7 signal`**

Azioni del kernel alla terminazione del processo

- **Sia nel caso di terminazione normale che in quella anormale le azioni attuate dal kernel sono le stesse:**
  - rimozione della memoria utilizzata dal processo
  - chiusura dei descrittori aperti

Valori di ritorno alla terminazione del processo

- **Exit status:**
  - Valore che viene passato ad **`exit`** (e **`_exit)`** e che notifica il padre su come è terminato il processo (successo, con errore)
- **Termination status:**
  - Valore che viene generato dal kernel nel caso di una terminazione normale/anormale
  - Nel caso si parli di terminazione anormale, si specifica la ragione per questa terminazione anormale
  - L'*exit status* e' convertito dal kernel in *termination status* quando alla fine viene chiamata **`_exit`**
- **Come ottenere questi valori?**
  - Tramite le funzioni **`wait`** e **`waitpid`** (descritte in seguito)

Processi *zombie*

- **Cosa succede se il padre termina prima del figlio?**
  - il processo figlio viene "adottato" dal processo **`init`** (PID=1), in quanto il kernel vuole evitare che un processo divenga "orfano" (cioè senza un PPID)
  - quando un processo termina, il kernel esamina la tabella dei processi per vedere se aveva figli; in tal caso, il PPID di ogni figlio viene posto uguale a 1
- **Cosa succede se il figlio termina prima del padre?**
  - generalmente il padre aspetta mediante la funzione **`wait`** che il figlio finisca ed ottiene le varie informazioni sull'*exit status*
  - se il figlio termina senza che il padre lo “aspetti”, il padre non avrebbe più modo di ottenere informazioni sull'*exit status* del figlio
  - per questo motivo, alcune informazioni sul figlio vengono mantenute in memoria e il processo diventa uno *zombie*

Lo stato di *zombie*

- Quando un processo entra nello stato di zombie, il kernel mantiene le informazioni che potrebbero essere richieste dal processo padre tramite **`wait`** e **`waitpid`**
  - process ID
  - termination status
  - accounting information (tempo impiegato dal processo)
- Il processo resterà uno zombie fino a quando il padre non eseguirà una delle system call **`wait`** o **`waitpid`**
- Per quanto riguarda i figli del processo **`init`**:
  - non possono diventare zombie
  - tutte le volte che un figlio di **`init`** termina, **`init`** esegue una chiamata **`wait`** e raccoglie eventuali informazioni: questo è il modo in cui gli zombie vengono eliminati dal sistema

System call `wait` e `waitpid`

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

- La **`wait`** e **`waitpid`** sono utilizzate per ottenere informazioni sulla terminazione dei processi figli
- Quando un processo chiama **`wait`** o **`waitpid`**:
  - può bloccarsi, se tutti i suoi figli sono ancora in esecuzione
  - può ritornare immediatamente con il termination status di un figlio, se un figlio ha terminato ed il suo termination status è in attesa di essere raccolto
  - può ritornare immediatamente con un errore, se il processo non ha alcun figlio
- **Nota:**
  - se eseguiamo una system call **`wait`** quando abbiamo già ricevuto **`SIGCHLD`**, essa termina immediatamente, altrimenti si blocca

System call `wait` e `waitpid`

- **Significato degli argomenti:**
  - **`status`** è un puntatore ad un intero; se diverso da **`NULL`**, il termination status viene messo in questa locazione
  - Il valore di ritorno è il process id del figlio che ha terminato
- **Differenza tra `wait` e `waitpid`:**
  - **`wait`** blocca il chiamante fino a quando un qualsiasi figlio non sia terminato
  - **`waitpid`** ha delle opzioni per evitare di bloccarsi
  - **`waitpid`** può mettersi in attesa di uno specifico processo
- **Il contenuto del termination status dipende dall'implementazione:**
  - bit per la terminazione normale, bit per l'exit status, etc.

Note sulla system call `waitpid`

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

- Argomento **`pid`**:
  - **`pid == -1`** si comporta come **`wait`**
  - **`pid > 0`** attende la terminazione del figlio con process id uguale a **`pid`**
  - **`pid == 0`** attende la terminazione di qualsiasi figlio con process group ID uguale a quello del chiamante
  - **`pid < -1`** attende la terminazione di qualsiasi figlio con process group ID uguale a **`–pid`**
- Opzioni (parametro **`options`**):

  `WNOHANG` non si blocca se il child non ha terminato

Notifica della terminazione di un figlio

- Quando un processo termina (normalmente o no), il padre viene informato dalla ricezione di un segnale **`SIGCHLD`**
- La notifica è asincrona
- Il padre ha la possibilità di:
  - ignorare il segnale (default)
  - predisporre una funzione speciale, detta *signal handler*, che viene invocata automaticamente quando il segnale viene ricevuto

Esercizi

- Scrivere un programma che esegue una **`fork`**, rimane in attesa che il figlio stampi un messaggio a video con successo e stampa un altro messaggio prima di uscire.
- Modificare il programma precedente affinché vengano generati un numero di figli dato sulla linea comandi. Cosa succede agli altri figli?