



Sistemi  
Operativi

Bruschi  
Monga Re

Concorrenza  
Semafori

# Sistemi Operativi<sup>1</sup>

Mattia Monga

Dip. di Informatica  
Università degli Studi di Milano, Italia  
[mattia.monga@unimi.it](mailto:mattia.monga@unimi.it)

a.a. 2013/14

<sup>1</sup> © 2008–14 M. Monga. Creative Commons Attribution — Condividi allo stesso modo 4.0 Internazionale. <http://creativecommons.org/licenses/by-sa/4.0/deed.it>. Immagini tratte da [2] e da Wikipedia.



Sistemi  
Operativi

Bruschi  
Monga Re

Concorrenza  
Semafori

# Lezione XXIV: Concorrenza



- **Concorrenza:** *run together & compete*
- Un processo non è piú un programma in esecuzione che può essere considerato in isolamento
- Non determinismo: il sistema nel suo complesso ( $P_1 + P_2 + \text{Scheduler}$ ) rimane deterministico, ma se si ignora lo scheduler le esecuzioni di  $P_1$  e  $P_2$  possono combinarsi in molto modi, con output del tutto differenti
- **Sincronizzazione:** si usano meccanismi (Peterson, TSL, semafori, monitor, message passing, ...) per imporre la combinazione voluta di  $P_1$  e  $P_2$

# Processi (senza mem. condivisa)



Sistemi  
Operativi

Bruschi  
Monga Re

Concorrenza  
Semafiori

```
1  int shared[2] = {0, 0};
2  /* int clone(int (*fn)(void *),
3   * void *child_stack,
4   * int flags,
5   * void *arg);
6   * crea una copia del chiamante (con le caratteristiche
7   * specificate da flags) e lo esegue partendo da fn */
8  if (clone(run, /* il nuovo
9           * processo esegue run(shared), vedi quarto
10          * parametro */
11         malloc(4096)+4096, /* lo stack del nuovo processo
12          * (cresce verso il basso!) */
13         SIGCHLD, // in questo caso la clone è analoga alla fork
14         shared) < 0){
15     perror("Errore nella creazione");exit(1);
16 }
17 if (clone(run, malloc(4096)+4096, SIGCHLD, shared) < 0){
18     perror("Errore nella creazione");exit(1);
19 }
20
```

# Processi (senza mem. condivisa)



Sistemi  
Operativi

Bruschi  
Monga Re

Concorrenza  
Semafori

```
1 int run(void* s)
2 {
3     int* shared = (int*)s; // alias per comodità
4     while (shared[0] < 10) {
5         sleep(1);
6         printf("Processo figlio (%d). s = %d\n",
7             getpid(), shared[0]);
8         if (!(shared[0] < 10)){
9             printf("Corsa critica!!!!\n");
10            abort();
11        }
12        shared[0] += 1;
13    }
14    return 0;
15 }
```

# Thread (con mem. condivisa)



Sistemi  
Operativi

Bruschi  
Monga Re

Concorrenza  
Semafiori

```
1  int shared[2] = {0, 0};
2  /* int clone(int (*fn)(void *),
3   * void *child_stack,
4   * int flags,
5   * void *arg);
6   * crea una copia del chiamante (con le caratteristiche
7   * specificate da flags) e lo esegue partendo da fn */
8  if (clone(run, /* il nuovo
9           * processo esegue run(shared), vedi quarto
10          * parametro */
11         malloc(4096)+4096, /* lo stack del nuovo processo
12          * (cresce verso il basso!) */
13         CLONE_VM | SIGCHLD, // (virtual) memory condivisa
14         shared) < 0){
15     perror("Errore nella creazione");exit(1);
16 }
17
18 if (clone(run, malloc(4096)+4096, CLONE_VM | SIGCHLD, shared) < 0){
19     perror("Errore nella creazione");exit(1);
20 }
```

# Thread (mutua esclusione con Peterson)



Sistemi  
Operativi

Bruschi  
Monga Re

Concorrenza  
Semafori

```
1
2 void enter_section(int process, int* turn, int* interested)
3 {
4     int other = 1 - process;
5     interested[process] = 1;
6     *turn = process;
7     while (*turn == process && interested[other]){
8         printf("Busy waiting di %d\n", process);
9     }
10 }
11
12 void leave_section(int process, int* interested)
13 {
14     interested[process] = 0;
15 }
```

# Thread (mutua esclusione con Peterson)



Sistemi  
Operativi

Bruschi  
Monga Re

Concorrenza  
Semafiori

```
1 int run(const int p, void* s)
2 {
3     int* shared = (int*)s; // alias per comodità
4     while (enter_section(p, &shared[1], &shared[2]), shared[0] < 10) {
5         sleep(1);
6         printf("Processo figlio (%d). s = %d\n",
7             getpid(), shared[0]);
8         if (!(shared[0] < 10)){
9             printf("Corsa critica!!!!\n");
10            abort();
11        }
12        shared[0] += 1;
13        leave_section(p, &shared[2]);
14    }
15    leave_section(p, &shared[2]); // il test nel while è dopo enter_section
16
17    return 0;
18 }
```





```
1 $ time ./threads-peterson > /tmp/output
2 real 0m11.091s
3 user 0m0.000s
4 sys 0m0.089s
5 $ grep -c "Busy waiting" /tmp/output
6 92314477
```

# Thread (mutua esclusione con TSL)



Sistemi  
Operativi

Bruschi  
Monga Re

Concorrenza  
Semafori

```
1
2 void enter_section(int *s); /* in enter.asm */
3 void leave_section(int *s){ *s = 0; }
4
5 int run(const int p, void* s){
6     int* shared = (int*)s; // alias per comodità
7     while (enter_section(&shared[1]), shared[0] < 10) {
8         sleep(1);
9         printf("Processo figlio (%d). s = %d\n",
10             getpid(), shared[0]);
11        fflush(stdout);
12        if (!(shared[0] < 10)){
13            printf("Corsa critica!!!!\n");
14            abort();
15        }
16        shared[0] += 1;
17        leave_section(&shared[1]);
18        sched_yield();
19    }
20    leave_section(&shared[1]); // il test nel while è dopo enter_section
```



# Thread (mutua esclusione con TSL)

Sistemi  
Operativi

Bruschi  
Monga Re

Concorrenza  
Semafori

```
1 section .text
2 global enter_section
3
4 enter_section:
5     enter 0, 0 ; 0 bytes of local stack space
6     mov ebx,[ebp+8] ; first parameter to function
7
8
9     ;; bts bitbase bitoffset
10    ;; selects the bitoffset bit in bitbase,
11    ;; stores the value in the CF flag, and sets the bit to 1
12 spin: lock bts dword [ebx], 0
13     jc spin
14
15     leave ; mov esp,ebp / pop ebp
16     ret
```



Una variabile intera **condivisa** controllata da system call che interagiscono con lo scheduler:

- down** decrementa, **bloccando** il chiamante se il valore corrente è 0; `sem_wait`
- up** incrementa, rendendo **ready** altri processi precedentemente bloccati se il valore corrente è maggiore di 0; `sem_post`

# Mutua esclusione con semafori



Sistemi  
Operativi

Bruschi  
Monga Re

Concorrenza  
Semafori

```
1 void down(sem_t *s){
2     if (sem_wait(s) < 0){
3         perror("Errore semaforo (down)");
4         exit(1);
5     }
6 }
```

```
1 void up(sem_t *s){
2     if (sem_post(s) < 0){
3         perror("Errore semaforo (up)");
4         exit(1);
5     }
6 }
```

# Mutua esclusione con semafori



Sistemi  
Operativi

Bruschi  
Monga Re

Concorrenza  
Semafori

```
1 void down(sem_t *s){
2     if (sem_wait(s) < 0){
3         perror("Errore semaforo (down)");
4         exit(1);
5     }
6 }
```

```
1 void up(sem_t *s){
2     if (sem_post(s) < 0){
3         perror("Errore semaforo (up)");
4         exit(1);
5     }
6 }
```

# Mutua esclusione con semafori



Sistemi  
Operativi

Bruschi  
Monga Re

Concorrenza  
Semafori

```
1 int shared = 0;
2 pthread_t p1, p2;
3 sem_t ss;
4
5 void* run(void* s){
6     while (down(&ss),
7           shared < 10) {
8         sleep(1);
9         printf("Processo thread (%p). s = %d\n",
10              pthread_self(), shared);
11         if (!(shared < 10)){
12             printf("Corsa critica!!!!\n");
13             abort();
14         }
15         shared += 1;
16         up(&ss);
17         pthread_yield();
18     }
19     up(&ss);
20     return NULL;
```