

programmazione:

PROCESSI

```

for (i = 0; i < n; i++) {
    pid_t pid = fork();
    if (pid == -1) {
        fprintf(stderr, "Can't fork, error %d\n", errno);
        exit(EXIT_FAILURE);
    } else if (pid == 0) {
        do_work();
        _exit(EXIT_SUCCESS);
    } else {
        wait();
    }
}

```

THREADS

Threads che eseguono la funzione Thread-fun

```

for (i = 0; i < n; i++) {
    pthread_t thread; sempre NULL
    ret = pthread_create(&thread, NULL, thread_fun, NULL); funzione da eseguire  
prendere e restituire *void
    if (ret != 0) {
        fprintf(stderr, "Can't create a new thread, error %d\n", ret);
        exit(EXIT_FAILURE);
    }
    ret = pthread_join(thread, NULL); se !=NULL salvo risultato  
estende la fine
    if (ret != 0) {
        fprintf(stderr, "Cannot join on thread, error %d\n", ret);
        exit(EXIT_FAILURE);
    }
}

```

Threads con passaggio di parametri

```

shared_array = (unsigned long int*)calloc(n, sizeof(unsigned long int));
pthread_t* threads = (pthread_t*)malloc(n * sizeof(pthread_t));
int* thread_ids = (int*)malloc(n * sizeof(int));
int i;
for (i = 0; i < n; i++) {
    thread_ids[i] = i;
    if (pthread_create(&threads[i], NULL, thread_work, &thread_ids[i]) != 0) {
        fprintf(stderr, "Can't create a new thread, error %d\n", errno);
        exit(EXIT_FAILURE);
    }
}
unsigned long int computed_value = 0;
for (i = 0; i < n; i++) {
    pthread_join(threads[i], NULL);
    computed_value += shared_array[i];
}
free(shared_array);
free(threads);
free(thread_ids);

```

void* thread_work(void *arg) {
 int thread_idx = *((int*)arg);
 int i;
 for (i = 0; i < m; i++)
 shared_array[thread_idx] += v;
 return NULL;
}

creazione
join
pulsione

Thread con valore di ritorno
→ cambiate alle funzioni di prima

```
void* thread_work(void *arg) {
    unsigned long * portion = (unsigned long *) calloc(1, sizeof(unsigned long));
    for (int i = 0; i < m; i++)
        *portion += v;
    pthread_exit(portion);
}
```

parte di join cambiate a quelle di prima

```
unsigned long computed_value = 0, *thread_result;
for (i = 0; i < n; i++) {
    pthread_join(threads[i], (void**) &thread_result);
    computed_value += *thread_result;
    free(thread_result);
}
```

c'è anche la funzione **pthread_detach(pthread_T Thread)**,
notifica al sistema che non ci saranno join su Thread
(ritorna 0 se ha successo altrimenti causa errore)

SEMAFORI

primitive:

sem_T sem;

0 se è tra threads, 1 tra processi

sem_init(& sem, pshared, value)

sem_wait(& sem) → -1

↓ valore di inizio

sem_post(& sem) → +1

sem_destroy(& sem)

produttore - consumitore:

necessari per tutte le sufficienze solo per 1-1

sem_t empty_sem, full_sem;
sem_t read_sem;
sem_t write_sem;
int transaction[BUFFER_SIZE];
int deposit = INITIAL_DEPOSIT;
int read_index, write_index;

void* performanceTransactions(void* x) { //producer thread
 if (sem_val(&empty_sem) == handle_error("Producer: sem_wait error empty sem")); //make sure there is space
 if (sem_val(&write_sem)) handle_error("Producer: sem_wait error write sem"); //get exclusive write access
 transaction[write_index] = performRandomTransaction(); //produzione
 write_index = (write_index + 1) % BUFFER_SIZE;
 if (sem_pos(&write_sem)) handle_error("Producer: sem_post error write sem"); //release exclusive write access
 if (sem_pos(&full_sem)) handle_error("Producer: sem_post error fill sem"); //notify that new element available
 pthread_exit(NULL);

void* processTransactions(void* x) { //consumer thread
 if (sem_val(&full_sem)) { // make sure there is data to consume
 handle_error("Consumer: sem_wait error fill sem");
 } if (sem_val(&read_sem)) { // get exclusive read access
 handle_error("Consumer: sem_wait error read sem");
 } deposit += transactions[read_index];
 read_index = (read_index + 1) % BUFFER_SIZE; // release exclusive read access
 if (sem_pos(&read_sem)) handle_error("Consumer: sem_post error read sem");
 if (sem_pos(&empty_sem)) { // notify that a free cell in the buffer just became available
 handle_error("Consumer: sem_post error empty sem");
 } pthread_exit(NULL);

```
int main(int argc, char* argv[]) {  
    read_index = 0;  
    write_index = 0;  
    if (sem_init(&full_sem, 0, 0)) handle_error("init error fill sem");  
    if (sem_init(&empty_sem, 0, BUFFER_SIZE)) handle_error("init error empty sem");  
    if (sem_init(&read_sem, 0, 1)) handle_error("init error read sem");  
    if (sem_init(&write_sem, 0, 1)) handle_error("init error write sem");  
    int ret;  
    pthread_t producer[NUM_PRODUCERS], consumer[NUM_CONSUMERS];  
    for (int i=0; i<NUM_PRODUCERS; ++i) {  
        ret = pthread_create(&producer[i], NULL, performTransactions, arg);  
        if (ret != 0) handle_error_en(ret, "Error in pthread create (producer)");  
    } for (int j=0; j<NUM_CONSUMERS; ++j) {  
        ret = pthread_create(&consumer[j], NULL, processTransactions, arg);  
        if (ret != 0) handle_error_en(ret, "Error in pthread create (consumer)");  
    } for (i=0; i<NUM_PRODUCERS; ++i) {  
        ret = pthread_join(producer[i], NULL);  
        if (ret != 0) handle_error_en(ret, "Error in pthread join (producer)");  
    } for (j=0; j<NUM_CONSUMERS; ++j) {  
        ret = pthread_join(consumer[j], NULL);  
        if (ret != 0) handle_error_en(ret, "Error in pthread join (consumer)");  
    } if (sem_destroy(&full_sem)) handle_error("Fill sem destroy error");  
    if (sem_destroy(&empty_sem)) handle_error("Empty sem destroy error");  
    if (sem_destroy(&read_sem)) handle_error("read sem destroy error");  
    if (sem_destroy(&write_sem)) handle_error("write sem destroy error");  
    exit(EXIT_SUCCESS);
```

SEMAFORI NAMED

primitive: $\rightarrow \text{sem_T}^*$

si usa $\text{sem_open}(\text{const char} * \text{name}, \text{int} \text{oflag})$ oppure, es. 0600
 $\text{sem_open}(\text{const char} * \text{name}, \text{int} \text{oflag}, \text{mode_T mode}, \text{unsigned int value})$
 qui processo terminato il lavoro esegue $\text{sem_close}(\text{sem_T sem})$
 1 processo deve eseguire $\text{sem_link}(\text{const char} * \text{name})$ valore di link
 se c'è 0_CREAT

server crea named semaphore e il client lo usa con diversi threads

SERVER

CLIENT

```
#define SEMAPHORE_NAME "/simple_scheduler" //in server.c
sem_t* named_semaphore;

void cleanup() { //in server.c
    if (sem_close(named_semaphore)) handle_error("sem_close error");
    if (sem_unlink(SEMAPHORE_NAME)) handle_error("sem_unlink error");
    exit(0);
}

int main(int argc, char* argv[]) { //in server.c
    int ret;
    sem_unlink(SEMAPHORE_NAME);
    named_semaphore = (sem_open(SEMAPHORE_NAME, 0_CREAT | 0_EXCL, 0600, NUM_RESOURCES));
    if (named_semaphore == SEM_FAILED) handle_error("Could not open the named semaphore");
    setQuitHandler(&cleanup); //cattura CTRL+C e manda una cleanup
    int current_value;
    ret = sem_getvalue(named_semaphore, &current_value);
    if (ret) {
        handle_error("Could not access the named semaphore");
        exit(1);
    }
    return 0;
}
```

```
#define SEMAPHORE_NAME "/simple_scheduler" //in client.c
```

```
void* client(void *arg_ptr) { //in client.c
    sem_t* my_named_semaphore = (sem_open(SEMAPHORE_NAME, 0));
    if (my_named_semaphore == SEM_FAILED) handle_error("Could not open the named semaphore from thread");
    if (sem_wlock(my_named_semaphore)) handle_error("Could not lock the semaphore from thread");
    //SEZIONE CRITICA
    if (sem_post(my_named_semaphore)) handle_error("Could not unlock the semaphore from thread");
    if (sem_close(my_named_semaphore)) handle_error("Could not close the semaphore from thread");
    return NULL;
}

int main(int argc, char* argv[]) { //in client.c
    int thread_ID = 0;
    for (int i = 0; i < THREAD_BURST; ++i) {
        pthread_t thread_handle;
        int ret = pthread_create(&thread_handle, NULL, client, NULL);
        if (ret) handle_error_en(ret, "Cannot create thread");
        ++thread_ID;
        ret = pthread_detach(thread_handle);
        if (ret) {
            handle_error_en(ret, "Cannot detach thread");
            exit(1);
        }
    }
    pthread_exit(NULL);
}
```

\downarrow
 $\text{int sem_get_value}(\text{sem_T} * \text{sem}, \text{int} * \text{val})$

\downarrow
 intero restituito al valore del sem

SHARED MEMORY

primitive:

\rightarrow es. "/name"

$\text{int shm_open}(\text{const char} * \text{name}, \text{int} \text{oflag}, \text{mode_T mode})$

descrittore de shm_open

creazione: 0_CREAT|0_EXCL|0_RDWR
 apertura: 0_RDONLY o 0_RDWR

0666 o 0660

$\text{int truncate}(\text{int fd}, \text{off_T length})$ dimensione la memoria
 condizionata a un riferimento fd a una lunghezza length

posizione memoria
 con 0 è automatico

PROT_READ, PROT_WRITE,
 PROT_READ|PROT_WRITE

$\text{void} * \text{mmap}(\text{void} * \text{addr}, \text{size_T length}, \text{int prot}, \text{int flags}, \text{int fd},$
 $\text{off_T offset})$ mappa la shared memory nella memoria mappata
 al processo per noi 0

\downarrow
 otturato da mmap

MAP_SHARED rende le modifiche visibili agli altri processi

$\text{int munmap}(\text{void} * \text{addr}, \text{size_T length})$ cancella il mapping tra
 processo e memoria mappata

→ stereo di shm-open

int **shm_unlink**(const char *name) rimuove una memoria condivisa e se tutti hanno fatto l'unmap la distrugge
→ da shm-open

int **close**(int fd) chiude il descrittore della memoria condivisa

mem_set per dare valori

Tramite fork()

requester cerca i dati, worker li elabora, requester li stampa
int *data è globale

```

int main(int argc, char **argv){
    /* Create and open the needed resources
     */
    sem unlink(SEM_NAME_REQ);
    sem unlink(SEM_NAME_WRK);
    global
    sem request = sem open(SEM_NAME_REQ, O_CREAT | O_EXCL, 0600, 0);
    if (sem_request == SEM_FAILED) handle_error("sem_open failed");
    sem_worker = sem open(SEM_NAME_WRK, O_CREAT | O_EXCL, 0600, 0);
    if (sem_worker == SEM_FAILED) handle_error("sem_open failed");

    shm unlink(SHM_NAME);
    global
    fd = shm open(SHM_NAME, O_CREAT | O_EXCL | O_RDWR, 0600);
    if (fd < 0)
        handle_error("main: error in shm_open");

    if(ftruncate(fd, SIZE) == -1)
        handle_error ("main: ftruncate");

    int ret;
    pid_t pid = fork();
    if (pid == -1) {
        handle_error("main: fork");
    } else if (pid == 0) {
        work();
        _exit(EXIT_SUCCESS);
    }

request();
    int status;
    ret = wait(&status);
    if (ret == -1)
        handle_error("main: wait");
    if (WEXITSTATUS(status)) handle_error_en(WEXITSTATUS(status), "request() crashed");

    /* Close and release resources
     */
    ret = sem close(sem_worker);
    if (ret) handle_error("main: sem_close worker");
    ret = sem close(sem_request);
    if (ret) handle_error("main: sem_close request");

    // then unlink
    ret = sem unlink(SEM_NAME_REQ);
    if (ret) handle_error("main: sem_unlink request");
    ret = sem unlink(SEM_NAME_WRK);
    if (ret) handle_error("main: sem_unlink worker");

    ret = close(fd);
    if (ret == -1)
        handle_error("main: cannot close the shared memory");

    ret = shm unlink(SHM_NAME);
    if (ret) handle_error("main: shm_unlink");

    _exit(EXIT_SUCCESS);
}

```

```

int work() {
    /* map the shared memory in the data array
     */
    if ((data = (int *) mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0)) == MAP_FAILED)
        handle_error ("main: mmap");
    printf("worker: mapped address: %p\n", data);

    /* Wait that the request() process generated data
     */
    printf("worker: waiting initial data\n");
    if (sem wait(sem worker) != 0)
        handle_error("worker: cannot lock the worker semaphore");

    printf("worker: acquire initial data\n");
    printf("worker: update data\n");
    int i;
    for (i = 0; i < NUM; ++i) {
        data[i] = data[i] * data[i];
    }

    printf("worker: release updated data\n");
    /* Signal the requester that elaboration terminated
     */
    if (sem post(sem request) != 0)
        handle_error("worker: cannot lock the request semaphore");

    /* Release resources
     */
    if (munmap(data, SIZE) == -1)
        handle_error("main: munmap");
    return EXIT_SUCCESS;
}

```

```

int request() {
    /*map the shared memory in the data array
     */
    if ((data = (int *) mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0)) == MAP_FAILED)
        handle_error ("main: mmap");
    printf("request: mapped address: %p\n", data);

    int i;
    for (i = 0; i < NUM; ++i) {
        data[i] = i;
    }
    printf("request: data generated\n");

    /* Signal the worker that it can start the elaboration
     * and wait it has terminated
     */
    if (sem post(sem worker) != 0)
        handle_error("request: cannot unlock the worker semaphore");

    if (sem wait(sem request) != 0)
        handle_error("request: cannot unlock the request semaphore");

    printf("request: acquire updated data\n");
    printf("request: updated data:\n");
    for (i = 0; i < NUM; ++i) {
        printf("%d\n", data[i]);
    }

    /* Release resources
     */
    if (munmap(data, SIZE) == -1)
        handle_error("main: munmap");
    return EXIT_SUCCESS;
}

```

regole con produttore - consumatore:

- P crea con open, truncate, mmap (INIZIO main)
 - e elimina con munmap, close, unlink (FINE main)
- C apre con open, mmap (INIZIO main)
 - e chiude con munmap, close (FINE SOTTO PROCESSO e FINE main)

stesso meccanismo
per i semafori

con semafori:

Buffer Full

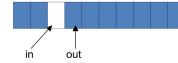


Buffer Empty

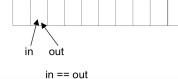


sensore semafori:

Buffer Full



Buffer Empty



FILE

copy da file in file con file descriptor:

```
int main(int argc, char* argv[]) {
    int block_size, src_fd, dest_fd;
    block_size = DEFAULT_BLOCK_SIZE;
    src_fd = open(argv[1], O_RDONLY);
    if (src_fd < 0) handle_error("Could not open source file");
    dest_fd = open(argv[2], O_WRONLY | O_CREAT | O_EXCL, 0644);
    if (dest_fd < 0) {
        if(errno == EEXIST) {
            fprintf(stderr, "WARNING: file %s already exists, I will overwrite it!\n", argv[2]);
            dest_fd = open(argv[2], O_WRONLY | O_CREAT, 0644);
        } else
            handle_error("Could not create destination file");
    }
    performCopyBetweenDescriptors(src_fd, dest_fd, block_size);
    int ret = close(src_fd);
    if (ret < 0) handle_error("Could not close source file");
    ret = close(dest_fd);
    if (ret < 0) handle_error("Could not close destination file");
    exit(EXIT_SUCCESS);
}
```

```
#define DEFAULT_BLOCK_SIZE 128;

static inline void performCopyBetweenDescriptors(int src_fd, int dest_fd, int block_size) {
    char* buf = malloc(block_size);
    while (1) {
        int read_bytes = 0;
        int bytes_left = block_size;
        while (bytes_left > 0) { // ciclo per leggere considerando EINTR
            int ret = read(src_fd, buf + read_bytes, bytes_left);
            if (ret == 0) break;
            if (ret == -1) {
                if(errno == EINTR) // read() interrupted by a signal
                    continue;
                handle_error("Cannot read from source file");
            }
            bytes_left -= ret;
            read_bytes += ret;
        }
        if (read_bytes == 0) break;
        int written_bytes = 0;
        bytes_left = read_bytes;
        while (bytes_left > 0) {
            int ret = write(dest_fd, buf + written_bytes, bytes_left);
            if (ret == -1) {
                if(errno == EINTR) // write() interrupted by a signal
                    continue;
                handle_error("Cannot write to destination file");
            }
            bytes_left -= ret;
            written_bytes += ret;
        }
    }
    free(buf);
}
```

PIPE (padre-figlio)

int pipe(int fd[2]) → fd[0] lettura ↗ si chiudono con close(int fd)
 fd[1] scrittura ↗ m° massimo di byte da leggere

dove salvare i dati ↗ int read(int fd, void *buf, int count)

read ritorna 0 se tutti i descrittori di scrittura sono stati chiusi

de dove prendere i dati ↗ m° massimo di byte da scrivere

int write(int fd, const void *buf, int count)

write causa SIGPIPE se tutti i descrittori di lettura sono stati chiusi

```
int main(int argc, char* argv[]) {
    int ret, i;
    pid_t pid;
    ret = pipe(pipefd);
    if(ret) handle_error("error creating the pipe");
    pid = fork();
    if (pid == -1) handle_error("error creating reader");
    if (pid == 0) {
        reader();
        _exit(0);
    }
    pid = fork();
    if (pid == -1) handle_error("error creating writer");
    if (pid == 0) {
        writer();
        _exit(0);
    }
    ret = close(pipefd[0]);
    if(ret) handle_error("error closing pipe");
    ret = close(pipefd[1]);
    if(ret) handle_error("error closing pipe");
    for (i = 0; i < 2; i++) { // shutdown phase
        int status;
        ret = wait(&status);
        if(ret == -1) handle_error("error waiting for a child to terminate");
        if (WEXITSTATUS(status)) {
            handle_error("ERROR: child process died unexpectedly");
        }
    }
    return 0;
}
```

```
void reader() {
    int data[LUNGDATA];
    int ret = close(pipefd[1]);
    if(ret) handle_error("error closing pipe");
    for (int i = 0; i < NMESSAGGI; i++) {
        read_from_pipe(pipefd[0], data, sizeof(data));
    }
    ret = close(pipefd[0]);
    if(ret) handle_error("READER: error closing pipe");
}

int read_from_pipe(int fd, void *data, size_t data_len) {
    int read_bytes = 0, ret;
    while (read_bytes < data_len) {
        ret = read(fd, data + read_bytes, data_len - read_bytes);
        if (ret == -1 && errno == EINTR) continue;
        if (ret == 0) handle_error("unexpected close of the pipe");
        read_bytes += ret;
    }
    return read_bytes;
}

void writer(int writer_id, sem_t* write_mutex) {
    int data[LUNGDATA];
    int ret = close(pipefd[0]);
    if(ret) handle_error("error closing pipe");
    for (int i = 0 ; i < NMESSAGGI; i++) {
        create_msg(data, MSG_ELEMS, i);
        write_to_pipe(pipefd[1], data, sizeof(data));
    }
    ret = close(pipefd[1]);
    if(ret) handle_error("error closing pipe");
}

int write_to_pipe(int fd, const void *data, size_t data_len) {
    int written_bytes = 0, ret;
    while (written_bytes < data_len) {
        ret = write(fd, data + written_bytes, data_len - written_bytes);
        if (ret == -1 && errno == EINTR) continue;
        if (ret == -1) handle_error("error writing to pipe");
        written_bytes += ret;
    }
    return written_bytes;
}
```

FIFO

named pipe è un file multidimensionale
per processi non relativi
name fifo

creazione int mkfifo(const char * path, mode_t mode)

apertura int open(const char * path, int oflag)

chiusura int close(int fd)

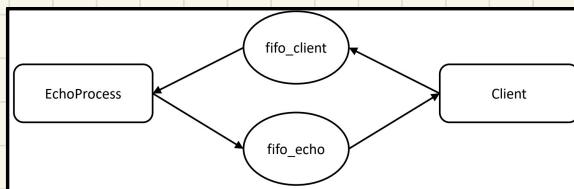
rimozione int unlink(const char * path)

settere le dimensione fentle(int fd, F_SETPIPE, size)

es. 0666

O_RDONLY, O_WRONLY...

es. 10 * sizeof(int)



server

```

int main(int argc, char* argv[]) { //in echo.c
    int ret, echo_fifo, client_fifo;
    char buf[1024];
    // Create the two FIFOs
    unlink(ECHO_FIFO_NAME);
    unlink(CLNT_FIFO_NAME);
    ret = mkfifo(ECHO_FIFO_NAME, 0666);
    if(ret) handle_error("Cannot create Echo FIFO");
    ret = mkfifo(CLNT_FIFO_NAME, 0666);
    if(ret) handle_error("Cannot create Client FIFO");
    echo_fifo = open(ECHO_FIFO_NAME, O_WRONLY); // Open the FIFO
    if(echo_fifo == -1) handle_error("Cannot open Echo FIFO for writing");
    client_fifo = open(CLNT_FIFO_NAME, O_RDONLY); // Open the FIFO
    if(client_fifo == -1) handle_error("Cannot open Client FIFO for reading");
    memset(buf, 0, 1024);
    int bytes_read = readOneByOne(client_fifo, buf, '\n');
    writeMsg(echo_fifo, buf, bytes_read);
    cleanFIFOs(echo_fifo, client_fifo);
    exit(EXIT_SUCCESS);
}

```

```

void writeMsg(int fd, char* buf, int size) { //scrive
    int ret;
    int bytes_sent = 0;
    while (bytes_sent < size) {
        ret = write(fd, buf + bytes_sent, size - bytes_sent);
        if (ret == -1 && errno == EINTR) continue;
        if (ret == -1) handle_error("Cannot write to FIFO");
        bytes_sent += ret;
    }
}

```

```

static void cleanFIFOs(int echo_fifo, int client_fifo) {
    int ret = close(echo_fifo); // close the descriptor
    if(ret) handle_error("Cannot close Echo FIFO");
    ret = close(client_fifo); // close the descriptor
    if(ret) handle_error("Cannot close Client FIFO");
    ret = unlink(ECHO_FIFO_NAME); // destroy the FIFO
    if(ret) handle_error("Cannot unlink Echo FIFO");
    ret = unlink(CLNT_FIFO_NAME); // destroy the FIFO
    if(ret) handle_error("Cannot unlink Client FIFO");
}

```

client

```

int main(int argc, char* argv[]) { //in client.c
    int ret;
    int echo_fifo, client_fifo;
    char buf[1024];
    // Open the two FIFOs
    echo_fifo = open(ECHO_FIFO_NAME, O_RDONLY);
    if(echo_fifo == -1) handle_error("Cannot open Echo FIFO for reading");
    client_fifo = open(CLNT_FIFO_NAME, O_WRONLY);
    if(client_fifo == -1) handle_error("Cannot open Client FIFO for writing");
    memset(buf, 0, 1024);
    int msg_len = strlen(buf);
    writeMsg(client_fifo, buf, msg_len);
    bytes_read = readOneByOne(echo_fifo, buf, '\n');
    buf[bytes_read] = '\0';
    printf("Server response: %s", buf);
    // close the descriptors
    ret = close(echo_fifo);
    if(ret) handle_error("Cannot close Echo FIFO");
    ret = close(client_fifo);
    if(ret) handle_error("Cannot close Client FIFO");
    exit(EXIT_SUCCESS);
}

```

```

int readOneByOne(int fd, char* buf, char separator) { //legge fino al separator (\n)
    int ret;
    int bytes_read = 0;
    do {
        ret = read(fd, buf + bytes_read, 1);
        if (ret == -1 && errno == EINTR) continue;
        if (ret == -1) handle_error("Cannot read from FIFO");
        if (ret == 0){
            printf("%s\n", buf);
            fflush(stdout);
            handle_error_en(bytes_read, "Process has closed the FIFO unexpectedly! Exiting...");
        }
    } while(buf[bytes_read++] != separator);
    return bytes_read;
}

```

SOCKET TCP client - server

creazione `int socket (AF_INET, SOCK_STREAM, 0)`

`ret` ↗ `mess. da inviare` ↗ `no max di byte da scrivere`

size_t send (int fd, const void *buf, size_t n, int flags)

`ret` ↗ `dove memorizzare il mess.` ↗ `se 0 send=write`

size_t recv (int fd, const void *buf, size_t n, int flags)

`ret` ↗ `se 0 recv = read`

chiusura `int close (int fd)`

SERVER

```
#define SERVER_ADDRESS "127.0.0.1"
#define SERVER_COMMAND "TIME"
#define SERVER_PORT 2015

void connection_handler(int socket_desc) { //in server.c
    int ret;
    char allowed_command = SERVER_COMMAND;
    size_t allowed_command_len = strlen(allowed_command);
    char send_buf[256];
    // receive command from client
    char recv_buf[256];
    size_t recv_buf_len = sizeof(recv_buf);
    int recv_bytes;
    // non consideriamo messaggi ricevuti parzialmente
    while ((recv_bytes = recv(socket_desc, recv_buf, recv_buf_len, 0)) < 0) {
        if (errno == EINTR) continue;
        handle_error("Cannot read from socket");
    }
    // parse command received and write reply in send_buf
    if (recv_bytes == allowed_command_len && !memcmp(recv_buf, allowed_command, allowed_command_len)) {
        time_t curr_time;
        time(&curr_time);
        sprintf(send_buf, "%s", ctime(&curr_time));
    } else {
        sprintf(send_buf, "INVALID REQUEST");
    }
    size_t server_message_len = strlen(send_buf);
    while ((ret = send(socket_desc, send_buf, server_message_len, 0)) < 0) {
        if (errno == EINTR) continue;
        handle_error("Cannot write to the socket");
    }
    // close socket
    ret = close(socket_desc);
    if (ret < 0) handle_error("Cannot close socket for incoming connection");
}
```

```
int main(int argc, char* argv[]) { //in server.c
    int ret, socket_desc, client_desc;
    struct sockaddr_in server_addr = {0}, client_addr = {0};
    int sockaddr_len = sizeof(struct sockaddr_in);
    // initialize socket for listening
    socket_desc = socket(AF_INET, SOCK_STREAM, 0);
    if (socket_desc < 0) handle_error("Could not create socket");
    server_addr.sin_addr.s_addr = INADDR_ANY;
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(SERVER_PORT);
    int reuseaddr_opt = 1;
    ret = setsockopt(socket_desc, SOL_SOCKET, SO_REUSEADDR, &reuseaddr_opt, sizeof(reuseaddr_opt));
    if (ret < 0) handle_error("Cannot set SO_REUSEADDR option");
    // bind address to socket
    ret = bind(socket_desc, (struct sockaddr*) &server_addr, sockaddr_len);
    if (ret < 0) handle_error("Cannot bind address to socket");
    // start listening
    ret = listen(socket_desc, MAX_CONN_QUEUE);
    if (ret < 0) handle_error("Cannot listen on socket");
    // loop to handle incoming connections serially
    while (1) {
        client_desc = accept(socket_desc, (struct sockaddr*) &client_addr, (socklen_t*) &sockaddr_len);
        if (client_desc < 0) handle_error("Cannot open socket for incoming connection");
        connection_handler(client_desc);
    }
    exit(EXIT_SUCCESS);
}
```

CLIENT

```
int main(int argc, char* argv[]) { //in client.c
    int ret, socket_desc;
    struct sockaddr_in server_addr = {0};
    socket_desc = socket(AF_INET, SOCK_STREAM, 0); // create a socket
    if (socket_desc < 0) handle_error("Could not create socket");
    // set up parameters for the connection
    server_addr.sin_addr.s_addr = inet_addr(SERVER_ADDRESS);
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(SERVER_PORT);
    // initiate a connection to the socket
    ret = connect(socket_desc, (struct sockaddr*) &server_addr, sizeof(struct sockaddr_in));
    if (ret < 0) handle_error("Could not create connection");
    // send command to server, non consideriamo messaggi inviati parzialmente
    char command = SERVER_COMMAND;
    size_t command_len = strlen(command);
    while ((ret = send(socket_desc, command, command_len, 0)) < 0) {
        if (errno == EINTR) continue;
        handle_error("Cannot write to socket");
    }
    // read message from the server, non consideriamo messaggi ricevuti parzialmente
    char recv_buf[256];
    size_t recv_buf_len = sizeof(recv_buf);
    while ((ret = recv(socket_desc, recv_buf, recv_buf_len - 1, 0)) < 0) {
        if (errno == EINTR) continue;
        handle_error("Cannot read from socket");
    }
    // close the socket
    ret = close(socket_desc);
    if (ret < 0) handle_error("Cannot close socket");
    exit(EXIT_SUCCESS);
}
```

per interrompere
la connessione

→ per rendere l'indirizzo
IP rintracciabile (no × esame)

per connection set-up:

sockaddr_in.sin_port → ai numeri in int16
mo di porta

- uint16_t htons (uint16_t hostshort)

converte un ushort de host byte order a network byte order

- uint16_t ntohs (uint16_t netshort)

converte un ushort de network byte order a host byte order

- in_addr_t inet_addr (const char * cp)

sockaddr_in.sin_addr.s_addr

IP ADDRESS

zero IPv4 in forme x.y.z.w

AF_INET

&(sockaddr_in.sin_addr) INET_ADDRSTRLEN

- const char * inet_ntop(int af, const void * src, char * dst, socklen_t n)

indirizzo di rete src
nella forma di stringa

restare statico lungo INET_ADDRSTRLEN

STRUTTURE DATI:

```
struct in_addr {  
    in_addr_t s_addr;           /32 bit IPv4 address  
}
```

```
struct sockaddr_in {  
    uint8_t      sin_len;     ① /lunghezza struttura  
    sa_family_t   sin_family;  ② /AF_INET (IPv4)  
    in_port_t     sin_port;    ③ /port num  
    struct in_addr sin_addr;  ④ /IPv4 address  
    char         sin_zero[8];  ⑤ /unused  
}  
/*inizializzare a zero sockaddr prima di usarla*/
```

*

SERVER struct sockaddr_in server_addr={0}, client_addr={0};

CLIENT struct sockaddr_in server_addr={0};

①

non si inizializza

②

= AF_INET

③

= htons(SERVER_PORT)

④

SERVER =INADDR_ANY

CLIENT =inet_addr(SERVER_ADDRESS)

⑤

non si inizializza

Funzioni:

- SERVER:**
- int **socket**(int family, int type, int protocol)
 - int **bind**(int fd, const struct sockaddr *addr, socklen_t len)
 - int **listen**(int sockfd, int backlog)
 - int **accept**(int fd, struct sockaddr *addr, socklen_t *len)
 - int **close**(int fd)
- CLIENT:**
- int **socket**() prima di **connect()** e **close()** dopo **de socket()**
 - int **connect**(int fd, const struct sockaddr *addr, socklen_t len)

se TCP AF_INET
 se UDP SOCK_DGRAM

del server

sizeof(struct sockaddr_in)

lunghezza max delle code per le connessioni MAXCONNQUEUE

come prime due (socklen_t*).

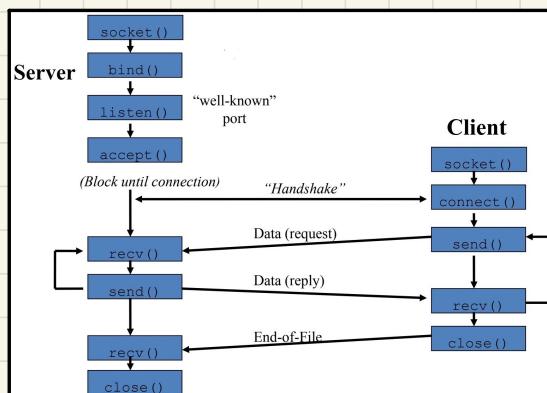


del client

del server

del client

del server



socket send e recv con controllo messaggi parzialmente ricevuti o inviati:

```

if (fgets(buf, sizeof(buf), stdin) != (char*)buf) { //lettura da stdin
    fprintf(stderr, "Error while reading from stdin, exiting...\n");
    exit(EXIT_FAILURE);
}

msg_len = strlen(buf);
bytes_sent=0;

while ( bytes_sent < msg_len ) { //controlla messaggi parzialmente inviati
    ret = send(socket_desc, buf + bytes_sent, msg_len - bytes_sent, 0);
    if (ret == -1 && errno == EINTR) continue;
    if (ret == -1) handle_error("Cannot write to the socket");
    bytes_sent += ret;
}

//controlla se messaggio ricevuto corrisponde al comando di terminazione
if (msg_len == quit_command_len && !memcmp(buf, quit_command, quit_command_len)) break;
recv_bytes = 0;

do { //controlla messaggi parzialmente ricevuti e legge 1 byte alla volta fino a trovare '\n'
    ret = recv(socket_desc, buf + recv_bytes, 1, 0);
    if (ret == -1 && errno == EINTR) continue;
    if (ret == -1) handle_error("Cannot read from the socket");
    if (ret == 0) break;
} while ( buf[recv_bytes++] != '\n' );

```

parti unore da sostituire ai while precedenti

SOCKET UDP client - server

↑
net
↑
size_t

sendto (int fd, const void *buf, size_t n, int flags,
const struct sockaddr *dest_addr, socklen_t l)

- fd, buf, n, flags come in send()
- dest_addr, l come in connect()

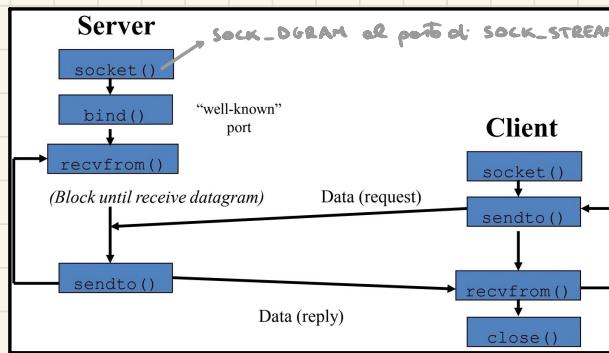
net
↑

recvfrom (int fd, void *buf, size_t n, int flags,
struct sockaddr *src_addr, socklen_t *addrlen)

- fd, buf, n, flags come in recv()
- src_addr e addrlen variabili per distinguere i client (anche null)

```
bytes_sent=0;
while ( bytes_sent < msg_len ) {
    ret = sendto(socket_desc, buf, msg_len, 0, (struct sockaddr*) &server_addr, sizeof(struct sockaddr_in));
    if (ret == -1 && errno == EINTR) continue;
    if (ret == -1) handle_error("Cannot write to the socket");
    bytes_sent = ret;
}
recv_bytes = 0;
do {
    ret = recvfrom(socket_desc, buf, buf_len, 0, NULL, NULL);
    if (ret == -1 && errno == EINTR) continue;
    if (ret == -1) handle_error("Cannot read from the socket");
    if (ret == 0) break;
    recv_bytes = ret;
} while ( recv_bytes<=0 );
```

→ & client_addr
→ & sockaddr_buf



Temi principali:

- processi, thread, concorrenza
- il sistema operativo
- reti di calcolatori
- inter-process communication
- sistemi distribuiti
- sicurezza informatica

processo: entità dinamica che viene caricata in memoria generata a partire da un programma, più precisamente:
è una sequenza di attività (Task) controllate dallo scheduler
in un processore gestito dal S.O.

RAM

composto da:

- programma codice
- insieme di dati
- n° di attributi che descrivono il processo durante l'esecuzione

durante l'esecuzione contiene diverse info. (process elements):

- identificatore
- task
- priorità
- program counter
- puntatori di memoria
- dati di contenuto
- info. di I/O
- accounting information

contenuti nel process control block

- creato e gestito dal S.O.
- permette supporti per processi multipli

ogni processo ha 2 caratteristiche:

- Scheduling / esecuzione: segue un path di esecuzione (che può essere intrecciato con altri processi)
- Resource ownership

• trattate in modo
indipendente
dal S.O.

padre e figlio
hanno forme →

Task
↓
↓

Date
Text

Thread:

l'unità di Scheduling si riferisce ad un Thread / process leggero
l'unità di resource ownership si riferisce a un processo o task

ogni Thread ha:

- uno stato di esecuzione (running, ready, ...)
- un contesto quando non è running
- uno stack di esecuzione
- spazio riservato al Thread per variabili locali
- accesso alle memorie e risorse del suo processo (che condivide tutti i Thread dello stesso processo)

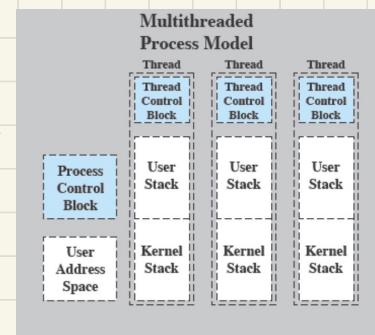
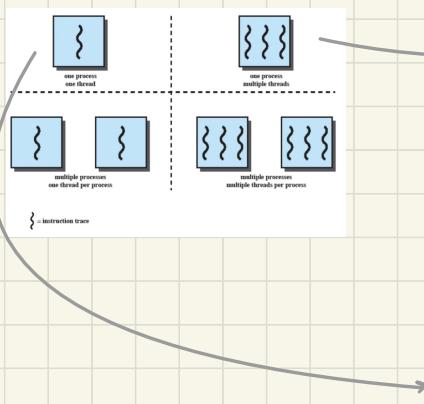
un Thread può essere visto come un indipendente program counter operativo all'interno di un processo

→ registro con indirizzo delle prossime istruzioni

benefici:

- la creazione di un Thread richiede meno tempo rispetto a un processo
- passare da un Thread ad un altro richiede meno tempo di passare da un processo ad un altro
- i Thread possono comunicare fra di loro senza invocare il Kernel

multi Thread:



Tipi di esecuzione:

- running
- ready
- blocked

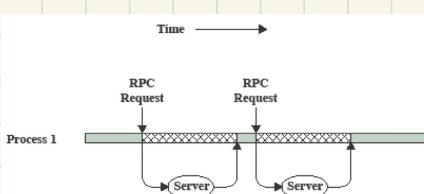
per cambiare stato:

- spawn (mava Thread ready)
- block
- unblock
- finish

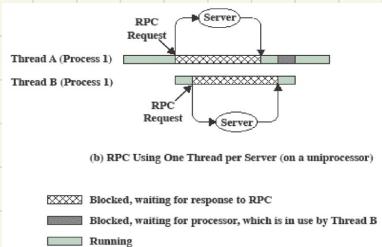
es.

remote procedure call (2 diversi host)

con 1 Thread



con 2 thread



categorie di implementazione Thread:

ULT user level Thread gestiti dall'utente (Tramite librerie)

KLT Kernel level Thread gestiti dal S.O. (processi leggeri)

il Kernel non è a conoscenza dei Thread

gestiti come processi decide il Kernel lo scheduling

vantaggi ULT:

- negliamo le politiche di alternanza
- il passaggio tra thread non richiede il Kernel

svantaggi ULT:

- una chiamata di blocco blocca tutti i ULT
- il S.O. non sa dell'esistenza dei thread quindi non concede altri core

vantaggi KLT:

- si possono eseguire diversi cose a diversi thread
- con un thread bloccato puoi pensare ad altri

disvantaggi KLT:

- si deve uscire il Kernel per passare da uno all'altro

esiste anche un approccio combiato

Thread e processi:

Threads:Processes	Description	Example Systems
1:1	Each thread of execution is a unique process with its own address space and resources.	Traditional UNIX implementations
M:1	A process defines an address space and dynamic resource ownership. Multiple threads may be created and executed within that process.	Windows NT, Solaris, Linux, OS/2, OS/390, MACH
1:M	A thread may migrate from one process environment to another. This allows a thread to be easily moved among distinct systems.	Ra (Clouds), Emerald
M:N	Combines attributes of M:1 and 1:M cases.	TRIX

manager: il manager thread assegna il lavoro agli altri thread
gestisce gli input e fornisce i risultati per le task

pipeline: la task fa parte di una serie di operazioni guidate in serie concorrentemente da thread diversi

ATTENZIONE:

- Tutti i Thread hanno accesso alla stessa memoria condivisa
- i Thread hanno le loro sezioni di dati protetti
- i programmatore sono responsabili di preservare la memoria condivisa (se si fa bene la programmazione è thread-safe)

PThreads: più veloce di fare le fork
libreria VLT che useremo

pthread_create() creazione

return()

pthread_exit() non fa le clean up

quando un Thread ha finito (non serve più) esistere

pthread_cancel() da un altro Thread

exit()

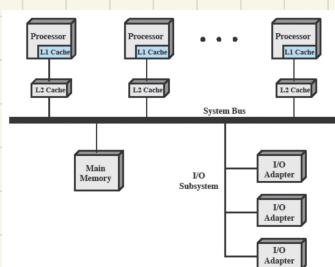
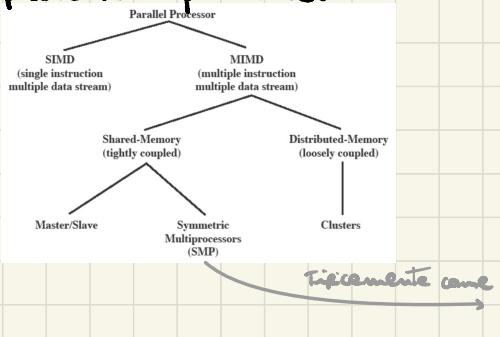
pthread_detach() per far dividere i Thread dal principale

usando **pthread_exit()** viene usato nel Thread principale
o **pthread_detach()** usato agli altri dopo essi continueranno
dopo la chiusura del principale

computer systems:

- SIMD single instruction multiple data
- MISD multiple instructions single data
- MIMD multiple instruction multiple data

processori paralleli:



concorrente:

applicazioni
multiple

dividere il
Tempo tra più
app. attive

applicazioni
strutturate

strutture
del S.O.

è un insieme
di processi e
Threads

istruzione critica: non può essere interrotta

sezione critica: porzione di codice in cui si accede a risorse condivise

mutua esclusione: se un programma è in una sezione critica deve essere l'unico che opera su quei dati in quel momento

condizione di rettifica: quando due processi modifichino lo stesso dato e il risultato dipende da chi finisce dopo

deadlock: situazione di fallo

livelock: quando dei processi scommettono steti in continuazione senza risultato

starvation: quando un processo è ready e non viene mai eseguito

Degree of Awareness	Relationship	Influence that One Process Has on the Other	Potential Control Problems
Processes unaware of each other	Competition	<ul style="list-style-type: none">Results of one process independent of the action of othersTiming of process may be affected	<ul style="list-style-type: none">Mutual exclusionDeadlock (renewable resource)Starvation
Processes indirectly aware of each other (e.g., shared object)	Cooperation by sharing	<ul style="list-style-type: none">Results of one process may depend on information obtained from othersTiming of process may be affected	<ul style="list-style-type: none">Mutual exclusionDeadlock (renewable resource)StarvationData coherence
Processes directly aware of each other (have communication primitives available to them)	Cooperation by communication	<ul style="list-style-type: none">Results of one process may depend on information obtained from othersTiming of process may be affected	<ul style="list-style-type: none">Deadlock (consumable resource)Starvation

meccanismi di concorrenza:

semaphore: contante da cui capiamo se la risorsa è disponibile o no

semaphore binario: semaforo con valori 0, 1

mutex: come il semaforo ma solo il processo che lo blocca lo può sbloccare

variabile di condizione: espressione booleana che blocca un processo / thread finché non è soddisfatta

monitor: variabile il cui accesso è consentito a un solo processo al tempo stesso, potrebbe avere una coda di processi in attesa

flags: è una parola usata come un meccanismo di sincronizzazione qui l'T corrisponde ad un event, il Thread rimane bloccato fino a quando i bit degli eventi richiesti non sono attivati

messaggio: per la comunicazione

spinlock: non blocca il processo ma va in busy waiting

semfor:

- inizializzato con un intero non negativo
- l'operazione **semWait** ne decremente il valore (se negativo process in attesa)
- l'operazione **semSignal** ne incrementa il valore

intruzioni atomiche!

```
struct semaphore {  
    int count;  
    queueType queue;  
};  
void semWait(semaphore s)  
{  
    s.count--;  
    if (s.count < 0) {  
        /* place this process in s.queue */;  
        /* block this process */;  
    }  
}  
void semSignal(semaphore s)  
{  
    s.count++;  
    if (s.count <= 0) {  
        /* remove a process P from s.queue */;  
        /* place process P on ready list */;  
    }  
}
```

```
struct binary_semaphore {  
    sum {zero, one} value;  
    queueType queue;  
};  
void semWaitB(binary_semaphore s)  
{  
    if (s.value == one)  
        s.value = zero;  
    else {  
        /* place this process in s.queue */;  
        /* block this process */;  
    }  
}  
void semSignalB(semaphore s)  
{  
    if (s.queue is empty())  
        s.value = one;  
    else {  
        /* remove a process P from s.queue */;  
        /* place process P on ready list */;  
    }  
}
```

bivio →

prima che un processo decrementa il semforo non si può sapere se si blocherà o meno
non c'è modo di sapere, se un sistema a 1 processore, quale processo continuerà dopo una semSignal se ce ne sono 2 running concorrentemente

semfori forti: la queue è FIFO

semfori deboli: l'ordine delle queue non è prefissato

semfori in C:

- `int sem_init (sem_t *sem, int pshared, unsigned value);`
- `int sem_wait (sem_t *sem)` wait
- `int sem_post (sem_t *sem)` signal
- `int sem_destroy (sem_t *sem)` destruction

sem → semforo

pshared → 0 se è il semforo è tra threads, 1 se tra processi

value → valore iniziale (n° di risorse)

monitors:

come i semafori ma più facili da controllare

- implementati in molti linguaggi di programmazione

- implementati come una libreria

- composti da una o più procedure, una sequenza di inizializzazioni e dati locali

i local data sono accessibili solo dalle procedure del monitor e non da altre

il processo entra nel monitor invocando una delle sue procedure

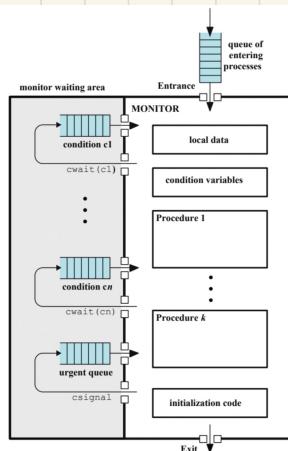
solo un processo alla volta può essere eseguito nel monitor

sincronizzazione:

si ottiene con l'uso di variabili di condizione contenute nel monitor e accessibili solo tramite il monitor

si opera su tali variabili tramite:

- **cwait(c)** → sospende l'esecuzione del processo chiamante secondo la condizione c
- **csignal(c)** → riprende l'esecuzione di un processo bloccato da una cwait con la stessa condizione



message passing:

quando due processi interagiscono devono avere 2 requisiti:

- **sincronizzazione** → per impostare la mutua esclusione (il message passing li garantisce)
- **communication** → per scambiare informazioni

la funzione è composta solitamente da 2 primitive:

- **send** (destinazione, messaggio)
- **receive** (proveniente, messaggio)

- se non c'è un messaggio in attesa il processo si blocca fino alla ricezione di un messaggio o si continua l'esecuzione abbandonando il tentativo di esecuzione

caratteristiche di progettazione di sistemi di messaggi per comunicazione e sincronizzazione tra processi:

Synchronization	Format	Queuing Discipline
Send	Content	FIFO
blocking	Length	Priority
nonblocking	fixed	
Receive	variable	
blocking		
nonblocking		
test for arrival		
Addressing		
Direct		
send		
receive		
explicit		
implicit		
Indirect		
static		
dynamic		
ownership		

blocking send, blocking receive:

- il mittente ed il destinatario sono bloccati fino alla consegna del messaggio
- chiamato anche appuntamento (rendezvous)
- permette una stretta sincronizzazione tra i processi

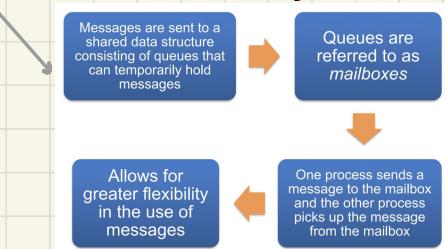
nonblocking send, blocking receive: comunicazione più utile!

- il mittente continua l'esecuzione ma il destinatario è bloccato fino all'arrivo del messaggio richiesto
- invia uno o più messaggi a più destinazioni più velocemente possibile

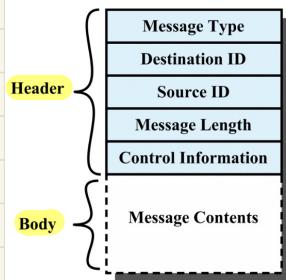
addressing: indirizzamento del destinatario

diretto → parla direttamente al destinatario

indiretto → parla ad un'entità che poi dà il destinatario quando lui lo chiede



composizione del messaggio:



problema produttore / consumatore:

si ha che:

- i o più produttori generano dati e li aggiungono al buffer
- i o più consumatori prendono elementi del buffer e alle volte
- si accede al buffer e alle volte (se produttori che consumatori)

problema:

i produttori non devono aggiungere dati ad un buffer pieno
i consumatori non devono prendere dati ad un buffer vuoto

```
/* program producerconsumer */  
semaphore n = 0, s = 1;  
  
void producer()  
{  
    while (true) {  
        produce();  
        semWait(s);  
        append();  
        semSignal(s);  
        semSignal(n);  
    }  
}
```

```
void consumer()  
{  
    while (true) {  
        semWait(n);  
        semWait(s);  
        take();  
        semSignal(s);  
        consume();  
    }  
}
```

soluzione semafori
(buffer infinito)

```

/* program boundedbuffer */
const int sizeofbuffer = /* buffer size */;
semaphore s = 1, n = 0, e = sizeofbuffer;

void producer()
{
    while (true) {
        produce();
        semWait(e);
        semWait(s);
        append();
        semSignal(s);
        semSignal(n);
    }
}

```

```

void consumer()
{
    while (true) {
        semWait(n);
        semWait(s);
        take();
        semSignal(s);
        semSignal(e);
        consume();
    }
}

```

notre (per altro buffer)
relazione
bounded-buffer

bounded-buffer
con i monitor:

```

void producer()
{
    char x;
    while (true) {
        produce(x);
        append(x);
    }
}

void consumer()
{
    char x;
    while (true) {
        take(x);
        consume(x);
    }
}

void take (char x)
{
    if (count == 0) cwait(notempty);
    x = buffer[nextout];
    nextout = (nextout + 1) % N;
    count--;
    csignal(notfull);
}

void append (char x)
{
    if (count == N) cwait(notfull);
    buffer[nextin] = x;
    nextin = (nextin + 1) % N;
    count++;
    /* one more item in buffer */
    csignal(notempty);
}

```

problema lettore/scrittori:

- qualcuno numero di lettori può leggere il file
- solo 1 scrittore alla volta può scrivere il file
- se uno scrittore scrive nessuno può leggere

```

/* program readersandwriters */
int readcount = 0;
semaphore x = 1, wsem = 1;

```

```

while (true) {
    semWait (wsem);
    WRITEUNIT();
    semSignal (wsem);
}

```

writer()

- acquire exclusive lock using binary semaphore wsem

```

while (true) {
    semWait (x);
    readcount++;
    if (readcount == 1) semWait (wsem);
    semSignal (x);
    READUNIT();
    semWait (x);
    readcount--;
    if (readcount == 0) semSignal (wsem);
    semSignal (x);
}

```

reader()

- binary semaphore x to safely update variable readcount
- when there is only one reader ($x=1$) lock wsem (no writes!)
- once the read is performed, unlock wsem if no reader is active (i.e., readcount==0) => writers may starve, extensions needed!

deadlock:

1 o più processi sono bloccati aspettando un'azione da un altro bloccato (permanente e no soluzioni efficienti)

risorse rivelabili: possono essere usate da un processo alla volta e non vengono "consumate" dall'uso

es. processori, canali I/O, semafori

risorse consumabili: vengono create e distrutte es. interrupt, segnali, messaggi.

condizioni per il deadlock:

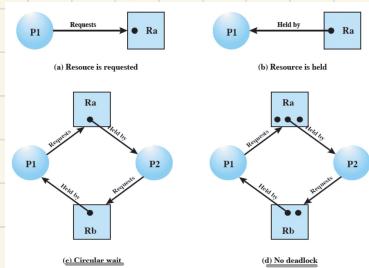
Mutual Exclusion	Hold-and-Wait	No Pre-emption	Circular Wait
• only one process may use a resource at a time	• a process may hold allocated resources while awaiting assignment of others	• no resource can be forcibly removed from a process holding it	• a closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain

necessarie ma non sufficienti

3 approcci di soluzione:

- prevenzione
- evitare
- trovarli

esempio di un grafo di allocazione di risorse



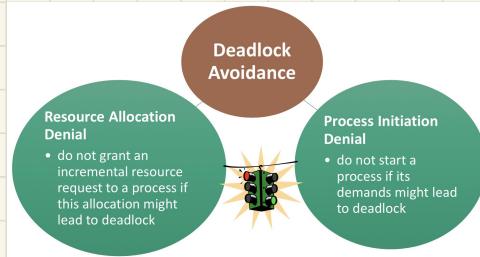
prevenzione:
indiretta preventire una delle 3 condizioni necessarie
diretta prevenire le wait circulari

- mutua esclusione
- hold and wait → richiede che un processo chiede tutte le risorse di cui ha bisogno subito e blocca il processo finché non ha tutte disponibili simultaneamente (inefficiente: potrebbe aspettare tanto e usare risorse per troppo tempo)
- no preemption → i processi richiedono e rilasciano le risorse (non le "tempo"), si deve avere priorità tra i processi (pratico solo se si può salvare lo stato e restituire)
- wait circulari → definire un ordine lineare di tipi di risorse, se un processo ha una risorsa R può avere solo altre risorse che vengono dopo R

entro i deadlock:

si esce dinamicamente, si controlla se le risorse richieste nel concorso porterebbe ad un deadlock

- richiede la conoscenza delle future richieste di un processo
- consente più concorrenza delle prevenzione



rigitto dell'arroto di un processo:

il S.O. permette l'esecuzione di un nuovo processo se la somma delle risorse già prese e quelle richieste del processo è <= al totale delle risorse

pendicolo del test:

```
boolean safe (state S) {  
    int currentavail[m];  
    process rest[<number of processes>];  
    currentavail = available;  
    rest = [<number of processes>];  
    possible = true;  
    while (possible) {  
        <find a process Pk in rest such that  
        claim [k,*] = alloc [k,*] < currentavail>;  
        if (round) /* simulate execution of Pk */  
            currentavail = currentavail + alloc [k,*];  
        rest = rest - {Pk};  
    }  
    else possible = false;  
}  
return (rest == null);  
}
```

(c) test for safety algorithm (banker's algorithm)

vantaggi:

- non è necessario anticipare e ripartire i processi (come nel rilevamento dei deadlock)
- non è restrittivo come la prevenzione

restrizioni:

- il numero massimo di risorse richieste da ogni processo deve essere stabilito prima
- i processi devono essere indipendenti (no interdipendenze)
- ci deve essere un numero preciso di risorse da allocare, aggiornate costantemente

la prevenzione limita l'accesso alle risorse, il rilevamento le de oppone possibile

rilevare i deadlock:

si può fare un controllo ad ogni richiesta o meno frequentemente
e secondo delle probabilità di avere deadlock

controllo ad ogni richiesta:

ventagg: → se c'è un deadlock lo scopriamo subito, l'algoritmo è facile

svantagg: → c'è un consumo importante di Tempo di CPU

soluzioni al deadlock: (ordine crescente di completezza)

- abortire tutti i processi nel deadlock (più comune)
- riportare tutti i processi interessati ad un checkpoint definito prima e farli ripartire
- abortire tutti i processi nel deadlock una alla volta fino a quando non c'è più il deadlock
- rimuovere le risorse una alla volta finché non c'è più il deadlock (c'è bisogno di un meccanismo di rollback)

riassunto approcci al deadlock:

Approach	Resource Allocation Policy	Different Schemes	Major Advantages	Major Disadvantages
Prevention	Conservative; undercommits resources	Requesting all resources at once	<ul style="list-style-type: none">• Works well for processes that perform a single burst of activity• No preemption necessary	<ul style="list-style-type: none">• Inefficient• Delays process initiation• Future resource requirements must be known by processes
		Preemption	<ul style="list-style-type: none">• Convenient when applied to resources whose state can be saved and restored easily	<ul style="list-style-type: none">• Preempts more often than necessary
		Resource ordering	<ul style="list-style-type: none">• Feasible to enforce via compile-time checks• Needs no run-time computation since problem is solved in system design	<ul style="list-style-type: none">• Disallows incremental resource requests
Avoidance	Midway between that of detection and prevention	Manipulate to find at least one safe path	<ul style="list-style-type: none">• No preemption necessary	<ul style="list-style-type: none">• Future resource requirements must be known by OS• Processes can be blocked for long periods
Detection	Very liberal; requested resources are granted where possible	Invoke periodically to test for deadlock	<ul style="list-style-type: none">• Never delays process initiation• Facilitates online handling	<ul style="list-style-type: none">• Inherent preemption losses

approcci software alla mutua esclusione:

scenari: i processi comunicano attraverso la memoria centrale su una macchina a singolo processore

conseguenze: mutua esclusione al livello di accesso alla memoria

- le operazioni di read/write per lo stesso indirizzo sono serializzate secondo un arbitrato della memoria
- no supporto in S.O., hardware, o linguaggio di programmazione

algoritmo di Dekker:

Tentativo:

- 1° c'è il monopolio anche se non si è in C.S. (quasi sterzatura)
- 2° se sono entrambi falsi allo stesso momento possono entrare in C.S. in 2
- 3° se entrambi i flag sono true nessuno entra in C.S.

corretto:

```
int me = 0, other = 1; // P0 (flip for P1)
```

```
while (true) {  
    /*NCS*/  
    flag[me] = true; → meglio entrare  
    while (flag[other]) { → se tu non entri  
        if (turn == other) { → e se è il tuo turno  
            flag[me] = false; → non meglio più entrare  
            while (turn == other) /* busy wait */ ; → se è il tuo turno aspetta  
            flag[me] = true; → meglio entrare  
        }  
    }  
    /* CS */  
    turn = other; → ora tu puoi entrare  
    flag[me] = false; → non meglio più entrare  
}
```

algoritmo di Dijkstra: ESAME

generalizzare Dekker per avere a che fare con N processi

garantite:

ME mutua esclusione

ND no deadlock (

NS no starvation) implica

non è garantito

altri problemi: servono read/write atomiche, condizione di memoria per K

```

/* global storage */
boolean interested[N] = {false, ..., false}
boolean passed[N]      = {false, ..., false}
int k < any>           // k ∈ {0, 1, ..., N-1}

/* local info */
int i = <entity ID> // i ∈ {0, 1, ..., N-1}

```

algoritmo:

```

while (true) {
    /*NCS*/
    1. interested[i] = true
    2. while (k != i) {
        3.     passed[i] = false
        4.     if (!interested[k]) then k = i
    }
    5. passed[i] = true
    6. for j in 1 ... N except i do
        7.     if (passed[j]) then goto 2
    8. <critical section>
    9. passed[i] = false; interested[i] = false
}

```

→ di chi è il turno

→ interessato ad entrare

→ K sceglie tra i processi interessati

→ il processo i ha passato la fase 1

→ si rinvia se più di 1 processo ha passato la fase 1

algoritmo del portiere: ESAME!

l'idea è come del portiere: ordiniamo e numeriamo

code of process i, $i \in \{1, \dots, n\}$

```

while (1){ //client thread
    /*NCS*/
    choosing = true; //doorway
    for j in 1 .. N except i {
        send(Pj,num);
        receive(Pj,v);
        num = max(num,v);
    }
    num = num+1;
    choosing = false;
    for j in 1 .. N except i { //backery
        do{
            send(Pj,choosing);
            receive(Pj,v);
        }while (v == true);
        do{
            send(Pj,v);
            receive(Pj,v);
        }while (v != 0 && (v,j) < (num,i));
    }
    /*CS*/
    num = 0;
}

```

```

//global variable
//inizialization:
int num = 0;
boolean choosing = false;
// and process ip/ports

```

```

while (1){ //server thread
    receive(Pj,message);
    if (message is a number)
        send(Pj,num);
    else
        send(Pj,choosing);
}

```

Assumptions:

- Finite response time
- Reliable communication channels

comunicazione tra processi IPC:

- un processo può accedere solo al suo address space
- ogni processo ha il suo address space
- il Kernel può accedere a tutti

procesi ↗
independenti non possono modificare o essere modificati da altri processi
cooperanti possono modificare o essere modificati da altri processi

motivi per la cooperazione:

condizionamento di info, speed-up delle computazioni, modularità, convenienza

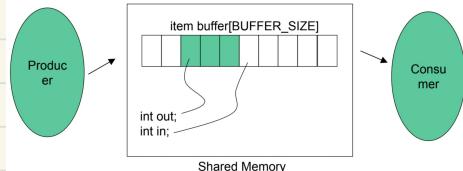
la IPC è il meccanismo per i processi di comunicare e sincronizzare le loro azioni, ci sono 2 modelli principali:

- 1) memoria condivisa i processi si scambiano dati in una memoria condivisa
- 2) passaggio di messaggi i processi si scambiano messaggi tramite Kernel

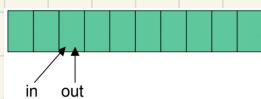
1)

una regione di memoria condivisa viene inserita tra 2 o più processi (Kernel non interviene)

es. produttore - consumatore con bounded buffer



buffer pieno

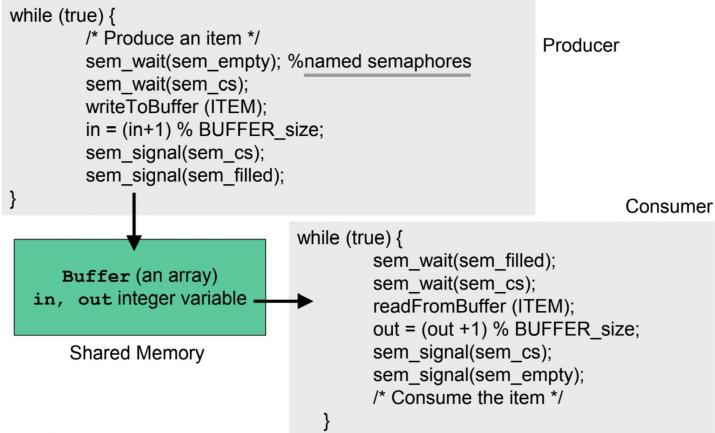


in == ant; rem - empty . val == 0;
rem - filled . val == BUFFER_SIZE

buffer vuoto



in == ant; rem - empty . val == BUFFER_SIZE;
rem - filled . val == 0



operazioni:

- `shm_open()` crea e/o apre la memoria condivisa
- `truncate()` o `ftruncate()` dimensione
- `mmap()` mappa nella memoria ora si può fare read/write
- `close()`
- `shm_unlink()`

message passing:

2 primitive:

- `send (destinazione, messaggio)` o `send (messaggio)`
- `receive (destinatario, messaggio)` o `receive (messaggio)`

formato messaggio:

Tipo del messaggio

ID destinazione

ID destinatario

lunghezza messaggio

info. di controllo

corpo del messaggio) body

) header

la queue è solitamente FIFO ma può avere priorità

pipes e named pipes (FIFOs) in Unix/Linux

un **pipe** consente la comunicazione con 1 verso tra un genitore e 1 figlio, è facile da usare e quando termina il processo viene rimossa automaticamente
la system call è `pipe(1)`

unnamed → solo tra padre e figlio

named → può accadere che unico con nome il nome

named:

- chiamato FIFO
- ha un nome
- non viene rimossa automaticamente quando termina il processo

- non c'è bisogno per padre-figlio
- bidirezionali
- possono essere creati e usati da tutti e 2 i processi

pipe in UNIX:

`int pipe(int fd[2])` per la creazione (caso di fallimento) ^(restituzione -1 in)
fd[0] lettura, fd[1] scrittura {descrittori di file per read() / write()}

se si tenta la lettura di una pipe vuota si ricevono blocchi finché non lo è più, lo stesso per le scritture di una pipe piena

le dim. massime di una pipe sono 26 pagine di memoria Linux
65,536 byte con page size di 4096

l'end of file si legge quando tutti gli scrittori che condividevano fd[1] lo hanno chiuso → la read() restituisce 0 come riferisce allo stesso modo se tutte le copie di fd[0] sono state chiuse la write su fd[1] restituisce il segnale SIGPIPE

entro deadlock:

per farlo tutti i processi devono chiudere i descrittori di pipe che non gli servono (con la `close()`)

ogni processo che crea le coppie (fd[0], fd[1]) deve fare le `close()` della sua copia di fd[1] prima di leggere da fd[0]

fifo in UNIX:

`int mkfifo(char *name, int mode)` → permette

la rimozione di una fifo dal file system avviene con `unlink()`

normalmente quando si apre una fifo in scrittura / lettura il processo si blocca finché non viene aperto anche in lettura / scrittura

per enterlo in modo si può aggiungere il flag O-NONBLOCK
(se perdi si vede senso un lettore → SIGPIPE)

1. Processi e Thread

PROCESSI:

Un processo è un'entità dinamica caricata sulla memoria RAM generata da un programma; più precisamente, è una sequenza di attività controllate da un programma che viene svolto su un processore tipicamente sotto la gestione o supervisione del rispettivo SO. Un programma inoltre comprende dati e attributi, il tutto viene salvato nel Process Control Block, creato e gestito dal SO che può così supportare più processi. Il SO inoltre permette all'utente di creare processi, a ciascuno garantisce tutte le risorse necessarie e permette a più processi di comunicare fra di loro. Per la gestione dei processi si utilizzano le seguenti Unix System Calls: fork(), wait(), exit(). Ogni processo (figlio) viene creato da un altro processo (padre) tramite una fork(), inizialmente sono quasi identici (anche stesso Process Control Block ma con diverso id processo) ma la loro evoluzione è diversa. Per fare sì che il processo padre attenda la terminazione del figlio si usa wait(), mentre per uscire da un processo si usa exit() specificando anche lo status di terminazione.

THREADS:

Un processo ha due caratteristiche: le risorse richieste (spaziale) e il cammino di esecuzione (temporale). Della parte temporale si occupano i thread. Un Sistema Operativo può supportare un processo con un thread, più processi con un thread ciascuno, un processo con più thread, più processi con più thread ciascuno. Il Multithreading è l'abilità di un Sistema Operativo di supportare più thread contemporaneamente, in questo caso ogni processo deve avere un indirizzo di memoria virtuale in cui mantenere la sua immagine e accesso protetto al processore, ad altri processi, ai file e alle risorse I/O.

Ogni thread ha uno stato di esecuzione (running, ready e blocked), un context data salvato quando non è running, uno stack di esecuzione, spazio statico per variabili locali, accesso alla memoria e alle risorse del processo di cui fa parte (Tutti i thread hanno accesso alla stessa memoria globale del processo oltre ad averne una privata).

- Un processo con singolo thread ha un Process control block, un user address space, un user stack e un kernel stack.
- Un processo con più thread ha invece un Process control block, un user address space e, per ciascun thread, un Thread Control Block, un user stack e un kernel stack.

I thread possono essere gestiti su due livelli: User Level Thread (ULT) o Kernel Level Thread (KLT).

- Nel primo caso il kernel non si accorge dell'esistenza dei thread che vengono gestiti interamente da una libreria, è limitante perché il processo avrà assegnato un solo core e non potrà quindi eseguire più thread contemporaneamente, ma si risparmia tempo non passando per il kernel e permette l'esecuzione su ogni sistema.

- Nel secondo caso il kernel manterrà il contesto dell'informazione fra il processo e i thread e lo scheduling sarà fatto a livello di kernel, permette di avere in esecuzione contemporanea più thread su più core anche se causa una perdita di tempo dovuta al passaggio per il kernel. Esistono anche degli approcci combinati.

Nei sistemi UNIX si utilizza la libreria PThread che permette di organizzare il programma in diversi task indipendenti che possono essere eseguiti contemporaneamente. Si può programmare con l'utilizzo di Thread attraverso due tipi di schemi: manager/worker (un thread ne crea altri e aspetta i risultati) o pipeline (divide il task in più thread e li lancia tutti insieme).

I benefici dei thread sono la rapidità di creazione e terminazione e nel passare da un thread all'altro, inoltre si possono sincronizzare tra loro e la comunicazione fra essi non necessita l'invocazione del kernel.

2. Concorrenza (mutua esclusione e sincronizzazione)

La concorrenza può presentarsi in tre diversi contesti: applicazioni multiple (condivisione del tempo fra varie applicazioni), applicazioni strutturate (gestione applicazioni con più processi), struttura dei sistemi operativi (loro stessi sono composti da processi e thread).

TERMINOLOGIA:

- **Operazione atomica**: una sequenza di istruzioni che non può essere interrotta.
- **Sezione critica**: sezione di codice che richiede accesso a risorse condivise che non devono essere toccate da altri thread che lavorano contemporaneamente.
- **Mutua esclusione**: condizione che quando un processo in una sezione critica accede a risorse condivise, nessun altro processo sia in una sezione critica che accede alle stesse risorse condivise.
- **Race Condition** (Corsa alla risorsa): situazione in cui due o più thread o processi leggono o scrivono allo stesso tempo in una memoria condivisa, per cui il risultato dipende dal timing. Il dato finale sarà quello scritto dall'ultimo processo, che sovrascriverà il dato già scritto da altri processi.
- **Deadlock**: un processo va in deadlock se attende una situazione che non si verificherà mai, ad esempio due processi che attendono ciascuno la terminazione dell'altro.
- **Livelock**: due o più processi cambiano continuamente il loro stato in risposta ai cambiamenti di stato dell'altro, senza fare però lavoro utile.
- **Starvation**: situazione in cui un processo è pronto a ripartire ma non viene scelto dallo scheduler e rimane inattivo. Magari inoltre blocca una risorsa e non permette agli altri di usarla.

Le difficoltà della concorrenza riguardano: condivisione di memorie globali, gestione da parte del sistema operativo delle risorse, difficoltà nel trovare errori di programmazione poiché le esecuzioni sono non deterministiche e irripetibili. Il sistema operativo deve conoscere tutti i processi in esecuzione, il loro stato, allocare e deallocare risorse per ognuno, proteggere dati e risorse da interferenze di altri processi, cercare di garantire che processi e output siano indipendenti dalla velocità di esecuzione. Processi concorrenti sono in conflitto quando competono per l'utilizzo della stessa risorsa, in questo caso bisogna affrontare tre problemi: la necessità di mutua esclusione, il deadlock e la starvation. La mutua esclusione deve permettere l'accesso alla sezione critica se e solo se nessun altro processo è nella sezione critica, inoltre deve bloccare gli altri che tentano di accederci. La sezione critica deve avere tempo finito. Si può rendere a livello hardware: disabilito interrupt del sistema operativo così che nessuno fermi l'esecuzione della sezione critica, ma l'efficienza peggiora e questo metodo non funziona con sistemi multi-processore. Servono istruzioni come la Compare&Swap: si confronta un valore in memoria con un valore di test, se sono uguali si scambia il valore in memoria con uno nuovo (il tutto atomicamente). Altrimenti si può usare l'istruzione Exchange che scambia il valore contenuto in un registro con un valore in memoria.

```
int compare_and_swap (int* reg, int oldval, int newval) {  
    ATOMIC();  
    int old_reg_val = *reg;  
    if (old_reg_val == oldval)  
        *reg = newval;  
    EN_ATOMIC();  
    return old_reg_val;  
}
```

```
void exchange (int *register, int *memory) {  
    int temp;  
    temp = *memory;  
    *memory = *register;  
    *register = temp;  
}
```

```
/*program mutual exclusion */  
const int n /* number of processes */;  
int bolt;  
void P (int i) {  
    while (true) {  
        while (compare_and_swap (&bolt, 0, 1) == 1)  
            /* do nothing */;  
        /* critical section */;  
        bolt = 0;  
        /* remainder */;  
    }  
}  
void main() {  
    bolt = 0;  
    parbegin (P(1), P(2), ..., P(n));  
}
```

```
/*program mutual exclusion */  
const int n /* number of processes */;  
int bolt;  
void P (int i) {  
    while (true) {  
        int keyi = 1;  
        do exchange (&keyi, &bolt) while (keyi != 0);  
        /* critical section */;  
        bolt = 0;  
        /* remainder */;  
    }  
}  
void main() {  
    bolt = 0;  
    parbegin (P(1), P(2), ..., P(n));  
}
```

Fra gli svantaggi abbiamo il busy-waiting poiché il processo che attende continua a consumare tempo del processore, starvation quando un processo lascia la sezione critica e più di un processo sta attendendo di entrarci, c'è possibilità che si verifichi il deadlock dovuto ad esempio a priorità fra processi, non è disponibile in tutte le architetture. Possiamo gestire la concorrenza anche grazie a meccanismi software: semafori, semafori binari, mutex, condition variabile, monitor, event flag, messaggi, spinlocks.

I semafori vengono sempre inizializzati ad un valore non negativo, le uniche operazioni consentite per modificarne il valore sono semwait (decremento) e semsignal (incremento), se eseguendo semwait il semaforo non è più >0 il processo si blocca ma non si può sapere se un processo si bloccherà o no fino a che non decremento il semaforo, né quale verrà bloccato fra due processi concorrenti. Le operazioni di semwait e semsignal devono essere necessariamente implementate come atomiche poiché la manipolazione di un semaforo è a sua volta un problema di mutua esclusione. Possono essere implementate in hardware o firmware con l'utilizzo di algoritmi come quello di Dekker, per semwait e semsignal posso utilizzare la struttura di programma con mutua esclusione attraverso compare&swap.

```
struct semaphore {
    int count;
    queueType queue;
};

void semWait (semaphore s) {
    s.count--;
    if (s.count < 0) {
        /* place this process in s.queue */
        /* block this process */
    }
}

void semSignal (semaphore s) {
    s.count++;
    if (s.count <= 0) {
        /* remove process P from s.queue */
        /* place process P on ready list */
    }
}
```

Il Message Passing è utile nella gestione di sistemi distribuiti che quindi non hanno memoria in comune. Permette sincronizzazione e scambio di informazioni fra i processi. Le primitive fondamentali sono send(destination, message) e receive(source, message). La comunicazione fra due processi implica la sincronizzazione. Quando chiamo una receive ho due possibilità, se un messaggio è in attesa di essere consegnato allora proseguo l'esecuzione, se nessun messaggio è stato inviato allora attenderò di riceverne uno o alternativamente abbandonerò il tentativo di receive e proseguiro. Quando chiamo una send posso attendere una ricevuta di consegna o proseguiro con l'esecuzione. Con la struttura BlockingSend- BlockingReceive sia il sender che il receiver sono bloccati fino a che il messaggio non è consegnato. La struttura NonblockingSend- BlockingReceive è la combinazione più utilizzata e permette di inviare più messaggi a diverse destinazioni rapidamente. Per la consegna di messaggi posso usare indirizzamento diretto o indiretto. Nel caso di indirizzamento diretto viene specificato il destinatario nella send, mentre nella receive è facoltativo il mittente. Nel caso di indirizzamento indiretto il messaggio è mandato ad una mailbox e il destinatario lo prende da lì. La mailbox può essere una semplice coda che mantiene i messaggi fino a che il receiver non li ha presi. Ogni messaggio deve avere una struttura definita contenente tipo del messaggio, source e destination id, lunghezza del messaggio, informazioni di controllo e contenuto del messaggio.

```
/*program mutual exclusion */
const int n = /* number of processes */;
void P (int i) {
    message msg;
    while (true) {
        receive (box, msg);
        /* critical section */;
        send (box, msg);
        /* remainder */;
    }
}
void main() {
    create_mailbox (box);
    send (box, null);
    parbegin (P(1), P(2), ..., P(n));
}
```

Problema produttore / consumatore

Problema lettori / scrittori

3. Deadlock e Starvation

Le condizioni per il deadlock sono: **mutua esclusione** (se non ho mutua esclusione non potrò avere deadlock perché più processi accederanno alla risorsa senza attendere), **hold-and-wait** (un processo mantiene bloccate delle risorse mentre attende che gliene vengano assegnate delle altre), **no preemption** (non è possibile rimuovere forzatamente delle risorse ad un processo). Queste condizioni sono necessarie ma non sufficienti, vi è una quarta condizione, la **circular wait**, quando c'è una catena di processi in cui ognuno mantiene una risorsa e ha bisogno di un'altra risorsa tenuta a sua volta da un altro processo (non necessariamente avrà deadlock). Possiamo avere 3 approcci risolutivi:

- **Prevenire:** eliminare le condizioni che lo generano. Si può attuare prevenzione indiretta (evito le tre condizioni necessarie) o diretta (evito circular wait). Per risolvere l'hold-and-wait il processo può prendere le risorse solo quando sono tutte disponibili ma questo genera lunghe attese e non sempre so in anticipo di cosa avrà bisogno. Per risolvere la no preemption il sistema operativo deve essere in grado di poter levare una risorsa ad un processo per garantirla ad un altro, bisogna quindi decidere un concetto di priorità e avere la capacità di salvare lo stato di un processo per farlo poi ripartire. Per evitare la circular wait devo definire una linea d'ordine per le risorse, per cui un processo che ha una risorsa potrà richiedere solo le risorse che sono in ordine "dopo" quelle che già ha, presenta lunghe attese e non sempre so in anticipo di quali risorse avrà bisogno.
- **Evitare:** si fanno scelte dinamiche sullo stato attuale delle risorse, si decide se la richiesta di allocazione di una risorsa fatta da un processo porterà ad un deadlock, per farlo c'è bisogno di sapere anche quali saranno le richieste successive. Due approcci: resource allocation denial (vietare l'accesso a una risorsa se rischio il deadlock) o process initiation denial (non si fa partire un processo se le future richieste porteranno a deadlock, ovvero se le risorse richieste più quelle già allocate superano quelle disponibili). Un safe state uno stato in cui l'allocazione di risorse ulteriori non porta a deadlock. La resource allocation denial è meno restrittiva della prevenzione del deadlock ed è anche migliore della process initiation denial perché evita di bloccare un intero processo. Gli svantaggi di questo approccio sono la necessità di conoscere in anticipo le richieste di risorse di un processo, può considerare solo processi indipendenti e senza sincronizzazione, necessita di un numero fisso di risorse allocabili e nessun processo può terminare se ha risorse allocate.

struct state {
int resource[m];
int available[m];
int claim[n][m];
int alloc[n][m];
}

Claim matrix C		
P1	P2	P3
3	2	2
6	1	3
3	1	4
4	2	2

Allocation matrix A		
P1	P2	P3
1	0	0
6	1	2
2	1	1
0	0	2

R1	R2	R3
9	3	6

R1	R2	R3
0	1	3

```
Algoritmo di allocazione risorse:
if (alloc [i,*] + request [*] > claim [i,*]) <error>;
/le risorse richieste superano quelle disponibili/
else if (request [*] > available [*]) <suspend process>;
/non ho risorse disponibili/
else {
    <define newstate by:
        alloc [i,*] = alloc [i,*] + request [*];
        available [*] = available [*] - request [*];
    >/simulo l'allocazione delle risorse/
    if (safe (newstate)) <carry out allocation>;
    else {
        <restore original state>;
        <suspend process>;
    }
}
```

```
Algoritmo per il test for safety (banker's algorithm)
boolean safe (state S) {
    int currentavail[m];
    process rest[<number of processes>];
    currentavail = available; /risorse disponibili/
    rest = {all processes};
    possible = true;
    while (possible) { /molto oneroso, controlla tutti i processi/
        <find a process Pk in rest such that
            claim [k,*] - alloc [k,*] <= currentavail;
        if (found) { /simula esecuzione di Pk/
            currentavail = currentavail + alloc [k,*];
            rest=rest - {Pk};
        }
        else possible = false;
    }
    return (rest == null); /se false lo stato non è sicuro/
}
```

- **Individuare:** e recuperare il deadlock. Le strategie di individuazione hanno come idea fondamentale la concessione di risorse quando possibile e il controllo periodico volto a trovare situazioni di deadlock. Posso ad esempio controllare se sono in una situazione di deadlock ogni volta che alloco nuove risorse, questo metodo individua subito eventuali problemi ma è molto oneroso sulla CPU. Un classico protocollo prevede il segnare iterativamente i processi che possono

essere sospesi o terminati. Se alla fine sono tutti segnati allora non abbiamo deadlock. Per recuperare un deadlock si possono uccidere i processi in deadlock (soluzione comune), ripristinare il sistema ad uno stato precedente senza deadlock (non mi garantisce che non si ripresenti lo stesso problema), uccidere uno alla volta i processi in deadlock cercando di risolverlo senza doverli uccidere tutti, usare la preemption (sottraggo risorse) per eliminare forzatamente il deadlock.

DEKKER-DIJKSTRA:

La mutua esclusione può essere garantita anche tramite approcci software. Si può utilizzare l'algoritmo di Dekker che funziona per mutua esclusione fra due processi. Dijkstra ha generalizzato l'algoritmo di Dekker per poter funzionare con un numero di processi > 2. Si rischia sempre che due processi escano dal while contemporaneamente e quello che non è in k sia più veloce ed entri in critical section prima che quello che sta in k si setti come passed, comunque quello non in critical section ritornerà al while sopra. Avendo 3 processi può essere che uno sia in k e quindi abbia diritto a proseguire ma per il sistema operativo abbia una priorità minore degli altri due, per cui rischiamo busy-waiting e starvation perché il processo che va in sezione critica potrebbe rimanere fermo (non in CPU) mentre i due nel while utilizzano la CPU.

L'algoritmo di Dijkstra quindi garantisce la mutua esclusione ed evita il deadlock ma non garantisce la no starvation e inoltre necessita di read/write atomiche e di memoria condivisa per k.

```
Algoritmo di Dekker
/*GLOBAL*/ boolean flag[2] = {false, false}
int me = 0, other = 1; //PO (flip for P1)
while (true) {
    /*non critical section*/
    flag[me] = true;
    while (flag[other]) {
        if (turn == other) {
            flag[me] = false;
            while (turn == other) /*busy wait*/;
            flag[me] = true;
        }
    }
    /*critical section*/
    turn = other;
    flag[me] = false;
}
```

```
Algoritmo di Dijkstra
/*GLOBAL*/    boolean interested[N] = {false, ..., false}
                boolean passed[N] = {false, ..., false}
                int k = <any>      //k fra 0 e N-1
/*LOCAL*/     int i = <entity ID>/i fra 0 e N-1
while (true) {
    /*non critical section*/
    interested[i] = true;
    while (k != i) {
        passed[i] = false;
        if (!interested[k]) then k=i;
    }
    passed[i] = true;
    for j in 1...N except i do
        if (passed[j]) then goto "while (k != i)"
    /*critical section*/
    passed[i] = false; interested[i] = false;
}
```

BAKERY ALGORITHM:

Per l'algoritmo del panettiere il concetto principale è che i processi prendono un numeretto e si mettono in fila per entrare nella sezione critica. Si ha che un processo attende nel controllare il numero di fila di un processo che lo sta modificando, maximum serve per non arrivare a numeri di fila infiniti. Le caratteristiche di questo algoritmo sono che i processi comunicano leggendo e scrivendo su variabili condivise (come Dijkstra), read e write non devono più essere operazioni atomiche, ogni variabile condivisa è posseduta da un processo che la può scrivere mentre gli altri la possono leggere, nessun processo può quindi fare scritture concorrenti, i tempi di esecuzione sono scollegati.

```
Ho N processi, ciascuno con identificatore i tra 1 ed N.
Ho due array: choosing contenente booleani e number contenente interi.
while (1) {
    /*non critical section*/
    while (number[i] == 0) {
        choosing[i] = true;
        number[i] = (1 + max {number[j] | (1 <= j <= N) except i}) %MAXIMUM
        choosing[i] = false;
    }
    for j in 1..N except i {
        while (choosing[j] == true);
        while (number[j] != 0 && (number[j],j) < (number[i],i));
    }
    /*critical section*/
    number[i] = 0;
}

Questa riga "number[i] = 1 + max {number[j] | (1 <= j <= N)}" si implementa:
local1 = 0;
for local2 in 1..N {
    local3 = number[local2];
    if (local1 < local3) local1 = local3;
}
number[i] = 1 + local1
```

4. COMUNICAZIONE INTER PROCESSO (IPC)

I vantaggi per cui due processi potrebbero essere cooperativi riguardano la modularità del programma, la convenienza (lavorare su più processi potrebbe essere meglio), la velocità (se un pezzo di programma attende qualcosa il resto può proseguire), lo scambio di informazioni. I meccanismi con cui si permette la comunicazione inter processo sono di due tipi: memoria condivisa o scambio di messaggi.

- o Nel caso di memoria condivisa, una volta assegnata dal kernel, questa sarà di rapido accesso per i processi perché non avranno bisogno di ripassare dal kernel ogni volta. I processi potranno leggere e scrivere in quest'area di memoria, per questo avremo bisogno di un meccanismo che impedisca ai due processi di accedere contemporaneamente (ad esempio un semaforo).
- o Nel caso di message passing, i messaggi sono salvati in una coda del kernel. Un processo scrive in questa coda e l'altro legge ma entrambi devono chiamare il kernel ogni volta e questo causa una notevole perdita di tempo. C'è bisogno di almeno due primitive, la send e la receive che prendono come parametro obbligatorio il messaggio e come parametri facoltativi rispettivamente il destinatario e il mittente. Il formato dei messaggi è stabilito, ciascuno contiene: tipo messaggio, ID destinatario e ID mittente, lunghezza del messaggio, informazioni di controllo e contenuto del messaggio. Il message passing è realizzato tramite Pipes o Named-Pipes (FIFOs).
 - Una Pipe attiva un collegamento bidirezionale fra due processi padre e figlio (processi relazionati), quando il processo termina la pipe è eliminata. Le Pipe permettono a più processi di comunicare come se stessero accedendo a dei file sequenziali, a livello di sistema operativo una pipe è un buffer di dimensione stabilita. Vi sono due descrittori per ogni Pipe, uno per la scrittura uno per la lettura, per evitare situazioni di deadlock si devono chiudere i descrittori appena finito l'utilizzo e ogni processo lettore deve subito chiudere il descrittore in scrittura come ogni processo scrittore deve subito chiudere il descrittore in lettura. Se si tenta di scrivere una pipe piena o leggere una pipe vuota si viene bloccati, ma esiste il modo di rendere read e write non bloccanti.
 - Le Named Pipe (FIFO) permettono la comunicazione anche tra processi non relazionati, non si chiudono autonomamente e sono monodirezionali. Normalmente, l'apertura di una FIFO è bloccante, nel senso che il processo che tenta di aprirla in lettura viene bloccato fino a quando un altro processo non la apre in scrittura come anche viceversa, anche se questo comportamento può essere inibito. Per lettura e scrittura vale lo stesso che per le pipe.

5. SOCKET

Lo scambio di dati fra entità (contenute in sistemi) può essere molto complesso. Per questo c'è bisogno di un protocollo che lo regoli, definito da semantica, sintassi e timing. Il mittente deve informare la rete riguardo l'identità del destinatario o attivare un collegamento diretto, inoltre deve assicurarsi che il destinatario sia pronto a ricevere e salvare i dati. Per la comunicazione si utilizzano vari moduli che collaborano fra di loro, i livelli di rete. Una Socket permette comunicazione fra un processo client e un server, ed è definita dalla coppia di indirizzo IP e numero di porta, può essere una stream socket (TCP), una datagram socket (UDP) a seconda del protocollo di comunicazione utilizzato.

- o I socket TCP sono identificati da una quadrupla: indirizzo IP del mittente, numero di porta del mittente, indirizzo IP del destinatario, numero di porta del destinatario. Una connessione TCP è unidirezionale, quindi ci sono due socket associati a una connessione: uno per il mittente e uno per il destinatario. Il server si occupa di eseguire socket(), bind(), listen(), accept() dopo quest'ultima quando il client esegue socket() e connect() si ha l'handshake e si eseguono una serie di recv() e send() sia da parte del client che da parte del server, fino a quando il client esegue una close(), la quale ricevuta dal server lo porta a eseguire close() a sua volta. Il protocollo TCP è orientato alla

connessione: stabilisce una connessione prima di trasferire dati e garantisce l'affidabilità nella consegna. Tra le sue proprietà ci sono affidabilità (garantisce che i dati siano consegnati senza errori e in ordine), controllo di congestione (adatta il tasso di trasmissione per evitare la congestione della rete), flusso di dati bidirezionale (supporta la trasmissione bidirezionale di dati tra mittente e destinatario), per questi motivi l'header TCP è molto più articolato di quello UDP.

- I socket UDP sono identificati da una coppia: indirizzo IP del destinatario, numero di porta del destinatario. Poiché UDP è senza connessione, non ci sono socket associati in modo specifico a una connessione, ogni pacchetto inviato o ricevuto è associato a un singolo socket. Non c'è handshake quindi il server si occupa solo di fare socket() e bind() prima di iniziare a inviare e ricevere, mentre il client dopo socket() è pronto ad inviare e ricevere e può in ogni momento fare close(). Il protocollo UDP non è orientato alla connessione: non stabilisce una connessione prima della trasmissione dei dati. Tra le sue proprietà ci sono la poca affidabilità (non garantisce l'ordine o la consegna dei dati), nessun controllo di congestione (non adatta il tasso di trasmissione in base alla congestione della rete), trasmissione veloce (adatto per applicazioni che richiedono una comunicazione più veloce e possono tollerare la perdita di alcuni pacchetti) ed inoltre il suo header è molto meno articolato ed ingombrante di quello TCP.

Un network adaptor (scheda di rete) è l'interfaccia tra un host e la rete, è costituito da due parti separate che interagiscono tramite una FIFO. Una parte interagisce con la CPU, la seconda parte interagisce con la rete e implementa i livelli fisico e di collegamento. Tutto il sistema è comandato da una SO che comunica con la CPU attraverso il CSR, un registro in cui entrambi possono settare e leggere dei flag. L'host può comunicare cosa accade nel CSR in due possimi modi:

- Busy waiting, la CPU legge continuamente il CSR finché non trova una modifica, ragionevole per dispositivi che non fanno altro come i router.

- Interrupt, l'adaptor invia un interrupt all'host che va quindi a leggere il CSR per capire cosa fare.

Il trasferimento dati dall'adaptor alla memoria e viceversa può essere:

- Direct Memory Access, non coinvolge la CPU, l'host ha un'area di memoria assegnata in cui vengono immediatamente inviati i frame. La memoria dove allocare i frames è organizzata attraverso una buffer descriptor list, ovvero un vettore di puntatori ad aree di memoria (i buffer) dove è descritta anche la quantità di memoria disponibile in ciascuna area. Per i frame che arrivano dall'adaptor si utilizza la tecnica scatter read / gather write secondo la quale frame distinti sono allocati in buffer distinti, nel caso in cui un frame sia troppo grande può essere spezzato in più buffer.

- Programmed I/O, lo scambio dati passa per la CPU, la memoria deve essere dual port per cui sia processore che adaptor devono poter leggere e scrivere.

Quando un messaggio viene inviato da un utente in una socket il sistema operativo lo copia in un buffer nella memoria in una zona di buffer descriptor. Tale messaggio viene processato da tutti i livelli che aggiungono le header. Quando il messaggio è pronto viene avvertita la SCO dell'adaptor attraverso alcuni flag del CSR, la SCO invia il messaggio sulla linea e notifica alla CPU il termine dell'invio settando CSR e inviando un interrupt. La CPU a questo punto lancia un interrupt handler che resetta i flag del CSR e libera le opportune risorse.

6. SISTEMI DISTRIBUITI

Un sistema distribuito è un insieme di entità (processi e thread ad esempio) separate spazialmente, non più nello stesso computer, che devono poter comunicare e coordinarsi. I sistemi distribuiti sono già molto presenti (LAN, DBMS, ATM, internet, peer to peer) in tutti gli ambiti e spesso necessitano di velocità quasi real time. I sistemi distribuiti sono preferibili rispetto a quelli centralizzati poiché sono:

- Economici: tanti sistemi sono più economici di uno più grande di pari potenza
- Veloci: posso creare sistemi di una potenza altrimenti non ottenibile da un sistema singolo

o Distribuiti: posso consultarlo da più punti fisici

o Scalabili: permettono una crescita incrementale, posso fare piccoli miglioramenti quando voglio

o Affidabili: il guasto di una piccola parte non impatta su tutto il sistema

La difficoltà maggiore sta nel far comunicare i sistemi distribuiti, inoltre viene generato molto traffico dati nella rete e aumento i rischi legati alla sicurezza. L'obiettivo primario è la condivisione di dati e risorse che devono essere sincronizzate e coordinate, quindi si deve gestire la concorrenza sia temporale che spaziale. Inoltre internet non è completamente affidabile e si deve poter gestire eventuali fallimenti di una macchina indipendentemente dalle altre. I principali problemi sono:

o Eterogeneità: macchine, reti, sistemi operativi e linguaggi di programmazione sono differenti

o Apertura: deve essere pubblico

o Sicurezza: integrità intesa come protezione contro l'alterazione dei dati, la disponibilità ovvero evitare che un dato sia bloccato in una macchina affinché rimanga disponibile

o Scalabilità: bisogna evitare perdita di prestazioni dovuta all'overhead e evitare colli di bottiglia nel sistema

o Affidabilità: non ho problemi se fallisce una sola macchina ma devo trovare, tollerare e risolvere eventuali failure (si potrebbe quindi necessitare di ridondanza)

o Concorrenza: spaziale e non solo temporale

o Flessibilità: il sistema deve essere facilmente modificabile anche se lega sistemi operativi e kernel diversi

o Performance: aumentando il numero di macchine aumenta l'overhead di gestione e aumentano i tempi di comunicazione

o Trasparenza: per l'utente l'esperienza non deve cambiare. Non deve potersi accorgere del fatto che il sistema è distribuito e non centralizzato

Bisogna inserire fra il sistema operativo e l'applicazione un Middleware che permetta ai sistemi di comunicare e cooperare. Il middleware è il cuore del sistema distribuito e si collega ad ogni sistema operativo tramite diverse interfacce e ad ogni applicazione grazie a specifiche API. Per comunicare tra sistemi distribuiti è necessario il message passing che avviene fra i middleware. L'inizializzazione di sistemi distribuiti può essere di tipo client-server o peer-to-peer.

Nel caso di Client-Server computing, ho un client con basse prestazioni e un server che sopperisce alle prestazioni del client. Sono connessi in Lan o tramite internet, l'applicazione si svolge nel server mentre il client viene usato solo per mostrare il risultato finale. Definendo 4 livelli di compiti: presentation logic, application logic, database logic, DBMS, si possono specificare 4 tipi di applicazione client-server:

o host based processing: ha i quattro livelli sul server

o server based processing: presentation è sul client, gli altri sul server

o cooperative processing: presentation e application sul client mentre application, database e DBMS sul server

o client based processing: presentation, application, database sul client mentre database e DBMS sul server

Una SOA (Service Oriented Architecture) è spesso usata nelle aziende ed organizza le funzioni in una struttura modulare interconnessa. Una serie di servizi e di clienti sono collegati tramite interfacce che permettono la comunicazione, i Service provider rendono noti i loro servizi al Service broker e quando un Service requester ha bisogno di qualcosa chiede al broker che lo rimanda al provider corretto.

Una Remote Procedure Call RPC è una funzione eseguibile da remoto che permette a macchine differenti di interagire usando semplici procedure (c'è bisogno di standardizzazione). Si possono avere connessioni persistenti (la connessione non viene dismessa per usi futuri) o non persistenti (appena si riceve il risultato la connessione viene dismessa). Le RPC possono essere sincrone (chi le

chiama attende il risultato) o asincrone (chi le chiama prosegue nell'elaborazione senza attendere il risultato che arriverà più avanti). Nel caso in cui ci siano degli errori si può scegliere quale politica applicare tra:

- At least once, prevede che si faccia nuovamente la richiesta nel caso in cui non si riceva risposta, sorge un problema se alla fine ricevo due risultati che potrebbero essere diversi.
- At most once, prevede che non si faccia nulla nel caso in cui non si riceva il risultato.
- Exactly one tra le tre la più difficile da implementare.

7. IOT

L'IoT è composto da qualsiasi dispositivo connesso con della capacità di calcolo e capace di generare e ricevere dati, ognuno di essi necessita di sensori, attuatori, microcontrollori, elementi di trasmissione e di identificazione univoca. Generalmente si considerano con ROM e RAM limitata, low-power, con performance e capacità di scambiare dati limitate. I sistemi operativi dedicati devono di conseguenza occupare poca memoria, supportare vari hardware e tipi di comunicazione, essere molto ottimizzati, efficienti energeticamente, con velocità realtime e sicuri.

8. SICUREZZA

La sicurezza è un processo che va gestito correttamente, le possibili minacce riguardano l'hardware, il software, i dati o le comunicazioni, nulla è completamente sicuro ma bisogna lavorare per avere un livello più alto possibile. Cercare sicurezza tenendo nascoste le implementazioni dei propri sistemi non è sempre la scelta giusta, rendendo pubblico il proprio codice si può infatti sperare che qualche utente trovi fallo e le segnali, è importante ricordare che la sicurezza di un sistema è pari alla sicurezza del componente meno sicuro e che non esiste sicurezza senza codice di buona qualità (security by design). La Computer Security è la protezione di un sistema di calcolo per preservare la CIA che comprende i tre obiettivi della sicurezza: Confidentiality, Integrity e Availability. Inoltre due concetti fondamentali sono: Authenticity (verificare se una informazione è genuina e se proviene dal mittente indicato) e Accountability (poter tracciare le responsabilità di una falla nella sicurezza).

TERMINOLOGIA:

- Gli attacchi si dividono in attivi (leggono, modificano, generano e distruggono informazioni) o passivi (leggono informazioni). Gli attivi possono essere di quattro categorie: replay (catturo informazioni e le rimando), masquerade (mi spacco per qualcun altro), modification of messages, denial of service (mando informazioni non richieste per intasare una rete).

- Un malware è un software malevolo creato per arrecare danno o usare le risorse di un target, in alcuni casi si propaga automaticamente.

- Un virus è un software che infetta processi modificandoli e si riproduce.

Un multiple-threat malware infetta in diversi modi quindi può infettare vari tipi di files.

Un blended attack usa vari metodi di infezione e trasmissione per potersi diffondere più velocemente e gravemente.

- Un rootkit è un programma malevolo che fa ottenere permessi di amministratore nel sistema, può essere persistente se sopravvive a un reboot, memory based altrimenti, può essere in user mode e quindi sfruttare le attività dell'utente modificandole o in kernel mode.

- Buffer Overflow è un meccanismo di attacco comune per cui ormai si conoscono varie tecniche di prevenzione. Sfrutta errori nella programmazione per i quali diventa possibile scrivere più dati della capacità disponibile di un buffer, sovrascrivendo locazioni di memoria adiacenti. Questo può portare alla corruzione dei dati di un programma, trasferimento del controllo ed esecuzione di codice dell'attacker. Per sfruttare il buffer overflow bisogna trovare vulnerabilità nel programma, capire come il buffer è salvato in memoria e determinare il potenziale di eventuali corruzioni di memoria per esempio nel modificare il contenuto di un array nello stack si può andare oltre e

sovrascrivere lo spazio allocato fino al return address facendo così in modo che punti alla porzione di codice che si vuole eseguire. Per fermare questo genere di attacchi bisogna programmare più attentamente e controllando i limiti dei buffer, questo si può fare con linguaggi di programmazione che impongono memory safety o attraverso librerie per linguaggi che non lo fanno di default. La protezione dal buffer overflow si può distinguere in due categorie: **compile-time defense** (rende il programma resistente ad attacchi) e **stack protection mechanism** (rivelà e risolve eventuali attacchi in programmi esistenti). Una tecnica consiste nell'utilizzo di canarini: si inserisce una porzione di memoria nello stack fra il return address e le variabili locali e prima di proseguire dove indicato dal return address si controlla che il contenuto del canarino non sia stato modificato. Altrimenti potrei dividere le sezioni di memoria in cui è permesso scrivere da quelle in cui è permesso leggere (WOX) o dividere quelle di codice eseguibile da quelle modificabili (DEP).

- Un **Bot** è un programma che si diffonde come un virus ma senza avere effetti indesiderati sul momento. Prende il controllo di vari dispositivi per creare una botnet fino a che non viene attivata per lanciare attacchi che non rendano tracciabile il creatore del bot. Le caratteristiche necessarie per una botnet sono: capacità di eseguire codice, un facile controllo da remoto, facilità di propagazione. Sfruttano vulnerabilità comuni a molti sistemi, attraverso ping e provando con una serie di indirizzi IP casuali tentano di entrare in reti locali e di propagarsi al loro interno. Con una botnet si possono fare vari attacchi fra cui: DDoS, spamming, sniffing traffic, keylogging, diffusione di malware, installazione di pubblicità, modificare chat e manipolare giochi.

- Un **DoS** (Denial of Service) è un attacco che prevede l'esaurimento delle risorse di un sistema per far sì che non sia disponibile a richieste lecite. Può riguardare banda di rete, risorse di sistema o risorse di applicazioni. Si rischia che il destinatario dell'attacco possa risalire alla sorgente da cui è partito. Usando una botnet l'attacco è più efficace e meno tracciabile, in questo caso si parla di **Distributed Denial of Service** (DDoS). Come soluzione si potrebbe bloccare il traffico se fosse inusualmente alto, ma potrebbe essere lecito e inoltre l'attaccante avrebbe comunque raggiunto il suo obiettivo. Si può risolvere limitando il numero di ping al secondo o inserendo dei captcha. Anche i service providers potrebbero essere attivi per evitare attacchi di questo tipo, bloccando pacchetti con indirizzo IP mittente che non corrisponde con quello reale o tenendo traccia dei percorsi che fanno i pacchetti, ma non è un costo che porterebbe beneficio al service provider stesso per cui è raro che implementi queste funzionalità.

TECNICHE DI PROTEZIONE:

o **Crittografia**: è un elemento molto importante e necessario ma non sufficiente, si basa sui numeri casuali (che nei computer sono in realtà semi-casuali).

- **Crittografia simmetrica**: si genera una chiave specifica per una comunicazione con cui si cifra e decifra il messaggio tramite un algoritmo. Un attacco basato sulla brute force non è realizzabile perché le chiavi sono di una lunghezza tale che servirebbero anni per trovare quella giusta. Per cifrare un insieme di dati si possono dividere in blocchi e cifrare ciascuno separatamente, invece nel caso di uno stream di dati la cifratura deve essere continua. I più famosi metodi di cifratura sono DES, Triple DES, AES.

- **Crittografia asimmetrica**: chi vuole ricevere dei dati genera due chiavi, una pubblica e una privata. Con quella pubblica i dati vengono cifrati e solo io che ho quella privata posso decifrarli. Deve essere semplice creare queste coppie di chiavi ma impossibile risalire alla privata partendo da quella pubblica.

MAC algorithm (Message Authentication Code): si utilizza per autenticare il messaggio. Data una chiave si calcola il MAC del messaggio e lo si aggiunge in coda al messaggio così che dopo la trasmissione il ricevente può ricalcolare il MAC del messaggio ricevuto e controllare che sia uguale al MAC calcolato in partenza. Si utilizzano funzioni di hash che permettono di avere come output un numero di lunghezza fissa indipendentemente dal messaggio in input.

o **Autenticazione**: sono presenti vari metodi:

- **Password utente**: quando inserisco una password viene calcolato e salvato il suo hash in modo tale che sia impossibile risalire alla password. Per evitare che utilizzando la stessa password si abbia lo

stesso hash viene anche inserito un sale che fa una prima codifica della password, questo sale viene poi salvato insieme all'hash finale della password ed è utilizzato per i successivi tentativi di autenticazione. Le minacce per questo tipo di autenticazione sono due: un user può guadagnare l'accesso ad una macchina usando un guest account, password cracker (programma che indovina le password).

- **Token:** Si dividono in memory cards e smart cards, le memory cards possono memorizzare dati per l'identificazione ma non hanno potenza di calcolo, le smart cards hanno anche della potenza di calcolo.

- **Biometrie:** si possono anche utilizzare le proprie biometrie per l'autenticazione poiché sono uniche e le si ha sempre con se, una caratteristica richiesta è che non varino nel tempo. Alcuni esempi sono le caratteristiche facciali, le impronte digitali, la forma della mano, la retina, l'iride, la firma o la voce ma sono utilizzate anche biometrie comportamentali. Il pericolo maggiore è che vengano "rubate" perché non si possono resettare.

o **Controllo di accesso:** assegna il tipo di accesso in base alle circostanze o all'utente che lo richiede. Le politiche di access control sono di tre tipi:

- **DAC discretionary access control** (ho il controllo in base alla mia identità e posso garantire il controllo anche ad altre identità);

- **MAC mandatory access control** (ho il controllo in base alla mia identità);

- **RBAC role-based access control** (ho il controllo in base al mio ruolo nel sistema, posso avere più ruoli).

o **Antivirus:** una soluzione efficace che controlla i file prima di memorizzarli, se malevoli li rimuove a meno che l'utente non dia indicazione diversa. L'obiettivo primario è la prevenzione ma in caso non venga raggiunto gli step sono 3 detection, identification, removal. Ogni antivirus può avere un suo decifratore, nel caso di sistemi più grandi si usa un digital immune system che identifica potenziali virus e li manda ad una macchina interna o esterna che analizza, ne crea una signature da mandare ai sistemi locali per poterlo riconoscere.

o **Firewall:** si occupa di bloccare pacchetti non legittimi provenienti dalla rete. Può lavorare a livello di trasporto bloccando alcune porte, a livello di rete bloccando alcuni indirizzi IP o a livello di applicazione. La gestione può prevedere il blocco di tutti i pacchetti eccetto quelli esplicitamente consentiti o alternativamente il blocco dei soli pacchetti specificati. I firewall sono generalmente numerosi in una rete, se ne inserisce uno all'ingresso dalla rete esterna, uno su ogni macchina e altri eventuali fra gli switch.

o **Intrusion detection:** Se l'intrusione viene scovata in tempo l'intruso può essere espulso senza arrecare danni, con l'host-based IDS si può monitorare il traffico di ogni host per creare delle statistiche e riconoscere immediatamente un livello anormale di traffico dovuto ad un'intrusione. Una honeypot è un sistema esca (singolo o una rete) che simula la rete interna ed è posta generalmente prima del firewall verso l'esterno, il suo scopo è attirare su di sé gli attacchi per raccoglierne informazioni. Si possono inserire honeypots anche all'interno della rete per controllare quali attacchi sono riusciti a superare il firewall.