

OPERATING SYSTEMS: COMMUNICATION AND SYNCHRONIZATION AMONG PROCESSES



Concurrent server development

To remember...

Before classes

Class

After class

Prepare the prerequisites.

Study the material associated with the **bibliography**:
slides alone are not enough.
Please ask questions (especially after study).

Exercising skills:

- ▶ Perform all **exercises**.
- ▶ Carrying out the **practice notebooks** and **the practical exercises** progressively.

Recommended reading

Base



1. Carretero 2020:
 1. Cap. 6
2. Carretero 2007:
 1. Cap. 6.1 and 6.2

Suggested



1. Tanenbaum 2006:
 1. (es) Chap. 5
 2. (en) Chap. 5
2. Stallings 2005:
 1. 5.1, 5.2 and 5.3
3. Silberschatz 2006:
 1. 6.1, 6.2, 6.5 and 6.6

Contents

- Introduction (definitions):
 - ▣ Concurrent processes.
 - ▣ Concurrency, communication and synchronization
 - ▣ Critical section and Race conditions
 - ▣ Mutual exclusion and critical section.
- Synchronization mechanisms (I):
 - ▣ Initial basic primitives
 - ▣ Semaphores.
- Classic concurrency problems (I):
 - ▣ Producer-consumer
 - ▣ Reader-writers
- Synchronization mechanisms of threads (II)
 - ▣ Semaphores
 - System calls for semaphores.
 - Classic concurrency problems.
 - ▣ Mutex and condition variables
 - System calls for mutex.
 - Classic concurrency problems.

□ Concurrent server development

- ▣ Request servers.
- ▣ Process-based solution.
- ▣ On-demand thread-based solution.
- ▣ Thread pool-based solution.

Contents

- Introduction (definitions):
 - ▣ Concurrent processes.
 - ▣ Concurrency, communication and synchronization
 - ▣ Critical section and Race conditions
 - ▣ Mutual exclusion and critical section.
- Synchronization mechanisms (I):
 - ▣ Initial basic primitives
 - ▣ Semaphores.
- Classic concurrency problems (I):
 - ▣ Producer-consumer
 - ▣ Reader-writers
- Synchronization mechanisms of threads (II)
 - ▣ Semaphores
 - System calls for semaphores.
 - Classic concurrency problems.
 - ▣ Mutex and condition variables
 - System calls for mutex.
 - Classic concurrency problems.

□ Concurrent server development

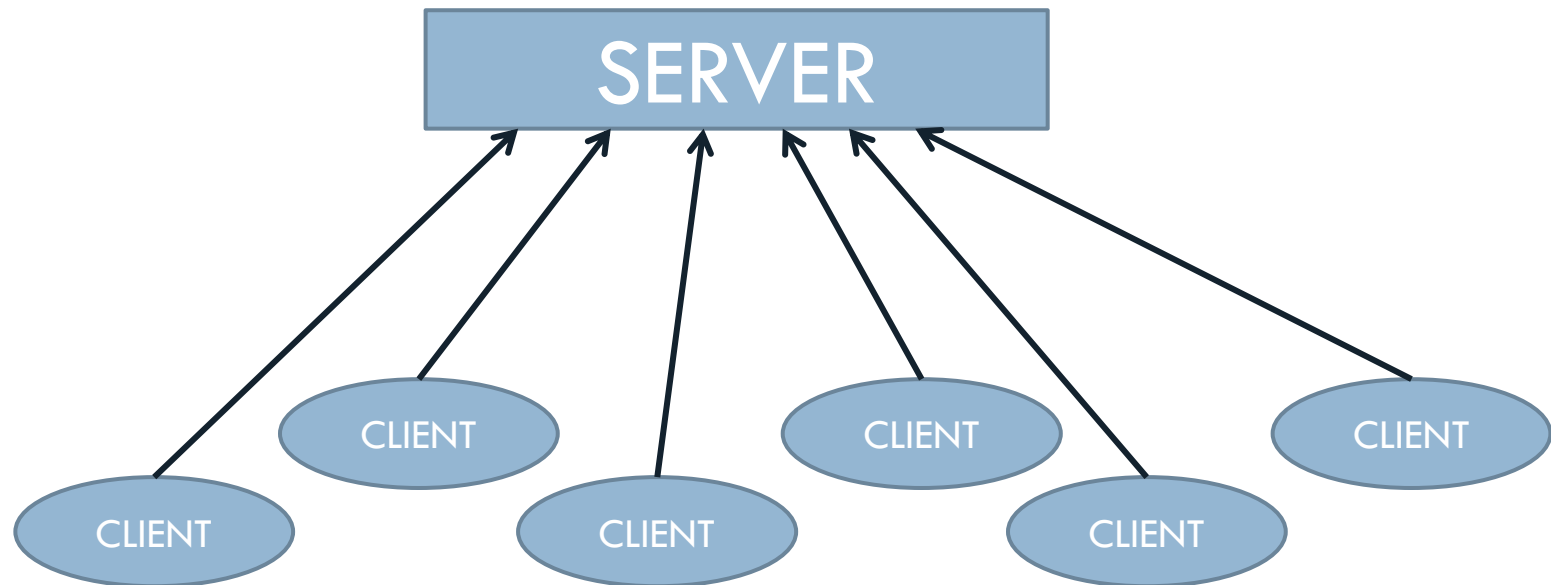
- ▣ **Request servers.**
- ▣ Process-based solution.
- ▣ On-demand thread-based solution.
- ▣ Thread pool-based solution.

Request server

- A server receives requests that it must process.
- In many contexts, request servers are developed:
 - Web Server.
 - Database server.
 - Application server.
 - File server.
 - Messaging applications
 - ...

Request server

- A server receives requests that it must process.



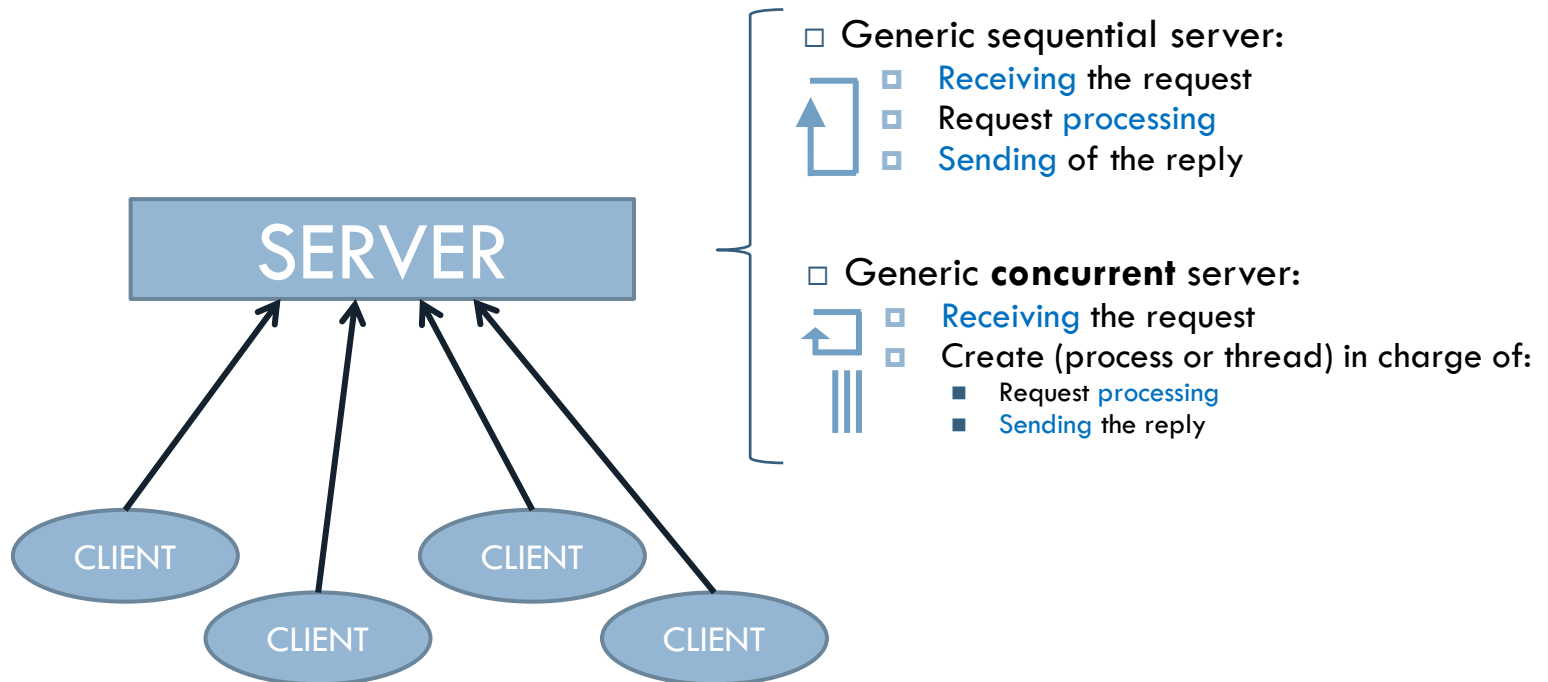
Request server

- A server receives requests that it must process.
- Structure of a generic server:
 - **Receiving** the request:
 - Each request requires a certain time in input/output operations to be received.
 - Request **processing**:
 - A certain CPU processing time.
 - **Sending** of the reply:
 - A certain input/output time for replying.



Concurrent request server?

- A server receives requests that it must process.



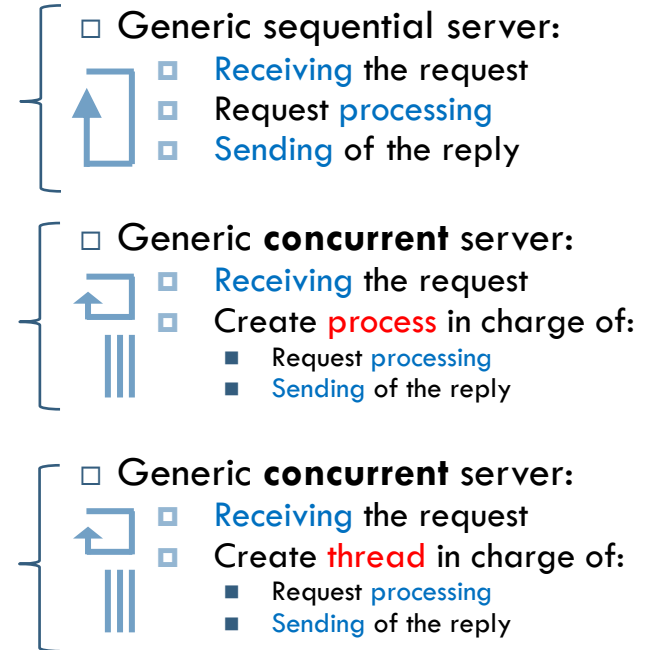
Test environment: test application

- Sequential

- Concurrent with processes

- Concurrent with threads on demand

- Concurrent with threads in a thread pool (pre-created)



Test environment: test application

□ The following will be used to evaluate the solutions:

□ Program that:

■ Attends NPET requests:

- Receiving request.
- Sending of the reply.

■ Measures the time it takes to deal requests.

□ A library that will simulate:

- Receiving requests.
- The processing and sending of responses.

```
#include "request.h"
#define NPET 20

int main()
{
    request_t p;

    t1=measure_time();
    for (i=0; i<NPET; i++) {
        receive_request(&p);
        reply_request(&p);
    }
    t2=measure_time();

    printf("Time: %d", t2-t1);
    return 0;
}
```

```
#ifndef REQUEST_H
#define REQUEST_H

typedef struct request{
    /* ... */
} request_t;

void receive_request ( request_t * p );
void reply_request   ( request_t * p );

#endif
```

Base library

requests.h

12

Alejandro Calderón Mateos 

```
#ifndef REQUEST_H
#define REQUEST_H

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/time.h>

struct request{
    long id;
    /* Other required fields */
    int type;
    /* ... */
};

typedef struct request request_t;

void receive_request ( request_t * p );
void reply_request   ( request_t * p );

#endif
```

Receiving requests

requests.c

13

Alejandro Calderón Mateos 

```
static long petid = 0;

void receive_request (request_t * p)
{
    int delay;

    fprintf(stderr, "Receiving requests\n");
    p->id = petid++;

    /* I/O timing simulation*/
    delay = rand() % 5;
    sleep(delay);

    fprintf(stderr, "Request %d received after %d seconds\n",
            p->id, delay);
}
```

Receiving requests

requests.c

14

Alejandro Calderón Mateos 

```
static long petid = 0;

void receive_request (request_t * p)
{
    int delay;

    fprintf(stderr, "Receiving requests\n");
    p->id = petid++;

    /* I/O timing simulation*/
    delay = rand() % 5;
    sleep(delay);

    fprintf(stderr, "Request %d received after %d seconds\n",
            p->id, delay);
}
```

Here would go some blocking call to receive the request (e.g. from the network).

Processing and sending of requests

requests.c

15

Alejandro Calderón Mateos 

```
void reply_request (request_t * p)
{
    int delay, i;
    char * mz;

    fprintf(stderr, "Sending request %d\n", p->id);

    /* Simulation of processing time */
    mz = malloc(1000000);
    for (i=0; i<1000000; i++) { mz[i] = 0; }
    free(mz) ;

    /* I/O timing simulation*/
    delay = rand() % 20;
    sleep(delay);

    fprintf(stderr, "Request %d sent after %d seconds\n",
            p->id, delay);
}
```

Processing and sending of requests

requests.c

16

Alejandro Calderón Mateos 

```
void reply_request (request_t * p)
{
    int delay, i;
    char * mz;

    fprintf(stderr, "Sending request %d\n", p->id);

    /* Simulation of processing time */
    mz = malloc(1000000);
    for (i=0; i<1000000; i++) { mz[i] = 0; }
    free(mz) ;

    /* I/O timing simulation*/
    delay = rand() % 20;
    sleep(delay);

    fprintf(stderr, "Request %d sent after %d seconds\n",
            p->id, delay);
}
```

The request would
be processed here

Here would go a blocking call to
reply to the request

Initial solution with metering

```
#include "request.h"

const int MAX_REQUESTS = 5;

int main ( int argc, char *argv[] )
{
    struct timeval ts;
    long t1, t2;
    request_t p;

    gettimeofday(&ts, NULL) ;
    t1 = (long)ts.tv_sec * 1000 +
        (long)ts.tv_usec / 1000 ;
    for (int i=0; i<MAX_REQUESTS; i++) {
        receive_request(&p);
        reply_request(&p);
    }
    gettimeofday(&ts, NULL) ;
    t2 = (long)ts.tv_sec * 1000 +
        (long)ts.tv_usec / 1000 ;

    printf("Time: %lf\n", (t2-t1)/1000.0);
    return 0;
}
```

```
#include "request.h"

const int MAX_REQUESTS = 5;

int main ( int argc, char *argv[] )
{
    struct timeval ts;
    long t1, t2;
    request_t p;

    gettimeofday(&ts, NULL) ;
    t1 = (long)ts.tv_sec * 1000 + (long)ts.tv_usec / 1000 ;
    for (int i=0; i<MAX_REQUESTS; i++) {
        receive_request(&p);
        reply_request(&p);
    }
    gettimeofday(&ts, NULL) ;
    t2 = (long)ts.tv_sec * 1000 + (long)ts.tv_usec / 1000 ;

    printf("Time: %lf\n", (t2-t1)/1000.0);
    return 0;
}
```

Execution of the initial solution

```
$ time ./ej1
Receiving requests
Request 0 received after 3 seconds
Sending request 0
Request 0 sent after 6 seconds
Receiving requests
Request 1 received after 2 seconds
Sending request 1
Request 1 sent after 15 seconds
Receiving requests
Request 2 received after 3 seconds
Sending request 2
Request 2 sent after 15 seconds
Receiving requests
Request 3 received after 1 seconds
Sending request 3
Request 3 sent after 12 seconds
Receiving requests
Request 4 received after 4 seconds
Sending request 4
Request 4 sent after 1 seconds
```

Time: 62.110000

```
real    1m2.053s
user    0m0.047s
sys     0m0.000s
```

Problems

- Arrival of requests:
 - If **two requests** came **at the same time**...
 - If **one request comes** while **another is being processed**...
- Use of resources.
 - How will the CPU utilization be?

Comparison

Sequential	Process per req.	Thread per request	Pool of threads
62.11 sec.			

Contents

- Introduction (definitions):
 - ▣ Concurrent processes.
 - ▣ Concurrency, communication and synchronization
 - ▣ Critical section and Race conditions
 - ▣ Mutual exclusion and critical section.
- Synchronization mechanisms (I):
 - ▣ Initial basic primitives
 - ▣ Semaphores.
- Classic concurrency problems (I):
 - ▣ Producer-consumer
 - ▣ Reader-writers
- Synchronization mechanisms of threads (II)
 - ▣ Semaphores
 - System calls for semaphores.
 - Classic concurrency problems.
 - ▣ Mutex and condition variables
 - System calls for mutex.
 - Classic concurrency problems.

□ Concurrent server development

- ▣ Request servers.
- ▣ **Process-based solution.**
- ▣ On-demand thread-based solution.
- ▣ Thread pool-based solution.

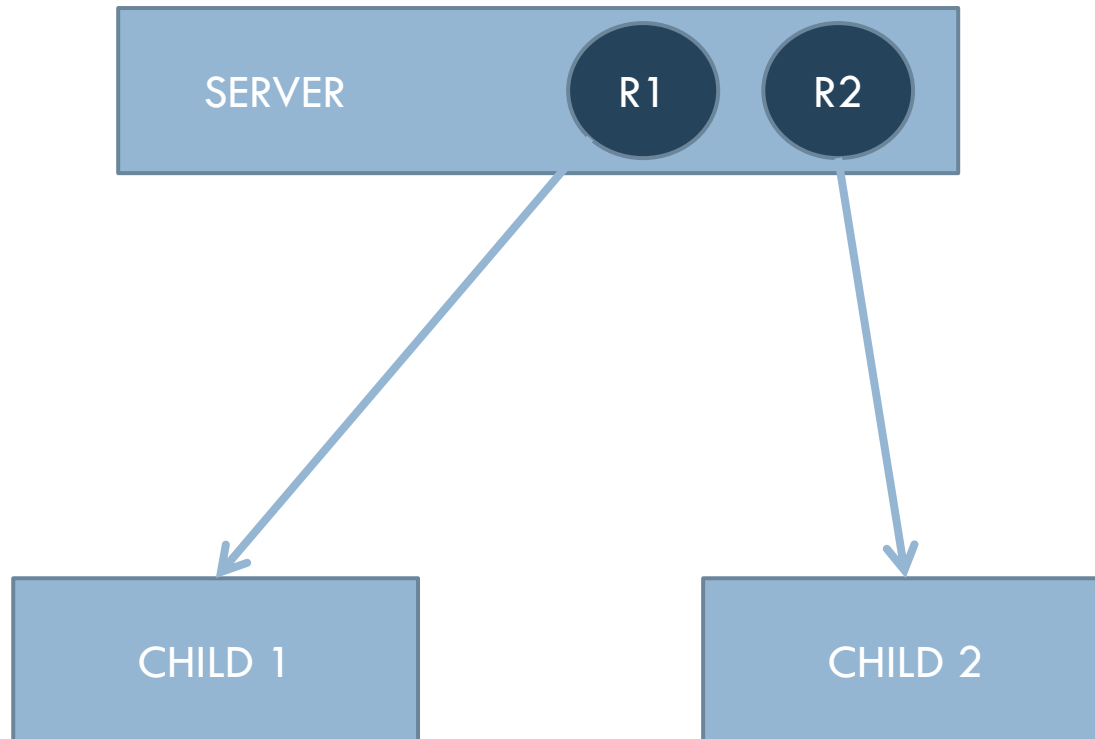
First idea

- Each time a request arrives, a child process is created (with fork system call):
 - The **child process processes** the **request**.
 - The **parent process waits** for the **next request**.

Process-based server

23

Alejandro Calderón Mateos 



Implementation (1 / 2)

```
#include <sys/types.h>
#include <sys/wait.h>
#include "request.h"

const int MAX_REQUESTS = 5;
void * receiver ( void ) ;

int main ( int argc, char *argv[] )
{
    struct timeval ts;
    long t1, t2;

    gettimeofday(&ts, NULL) ;
    t1 = (long)ts.tv_sec * 1000 + (long)ts.tv_usec / 1000 ;
    receiver() ;
    gettimeofday(&ts, NULL) ;
    t2 = (long)ts.tv_sec * 1000 + (long)ts.tv_usec / 1000 ;

    printf("Time: %lf\n", (t2-t1)/1000.0);
    return 0;
}
```

```
void * receiver ( void )
{
    request_t p;
    int pid, nchilds=0;

    for (int i=0; i<MAX_REQUESTS; i++)
    {
        receive_request(&p);

        pid = fork();
        if (pid<0) { perror("Error en la creación del hijo");}
        if (pid==0) { reply_request(&p); exit(0); } /* HIJO */
        if (pid!=0) { nchilds++; } /* PADRE */
    }

    fprintf(stderr, "Wait for %d nchilds\n", nchilds);
    while (nchilds > 0)
    {
        pid = waitpid(-1, NULL, WNOHANG);
        if (pid > 0) { nchilds--; }
    } ;

    return NULL ;
}
```

```
#include <sys/types.h>
#include <sys/wait.h>
#include "request.h"
```

```
const int MAX_REQUESTS = 5;
void * receiver ( void ) ;
```

```
int main ( int argc, char *argv[] )
{
```

```
    struct timeval ts;
    long t1, t2;
```

```
    gettimeofday(&ts, NULL) ;
    t1 = (long)ts.tv_sec * 1000 + (long)ts.tv_usec / 1000 ;
    receiver() ;
    gettimeofday(&ts, NULL) ;
    t2 = (long)ts.tv_sec * 1000 + (long)ts.tv_usec / 1000 ;
```

```
    printf("Time: %lf\n", (t2-t1)/1000.0);
    return 0;
```

```
}
```


Implementation (2/2)

```
#include <sys/types.h>
#include <sys/wait.h>
#include "request.h"

const int MAX_REQUESTS = 5;
void * receiver ( void ) ;

int main ( int argc, char *argv[] )
{
    struct timeval ts;
    long t1, t2;

    gettimeofday(&ts, NULL) ;
    t1 = (long)ts.tv_sec * 1000 + (long)ts.tv_usec / 1000 ;
    receiver() ;
    gettimeofday(&ts, NULL) ;
    t2 = (long)ts.tv_sec * 1000 + (long)ts.tv_usec / 1000 ;

    printf("Time: %lf\n", (t2-t1)/1000.0);
    return 0;
}

void * receiver ( void )
{
    request_t p;
    int pid, nchilds=0;

    for (int i=0; i<MAX_REQUESTS; i++)
    {
        receive_request(&p);

        pid = fork();
        if (pid<0) { perror("Error en la creación del hijo"); }
        if (pid==0) { reply_request(&p); exit(0); } /* HIJO */
        if (pid!=0) { nchilds++; } /* PADRE */
    }

    fprintf(stderr, "Wait for %d nchilds\n", nchilds);
    while (nchilds > 0)
    {
        pid = waitpid(-1, NULL, WNOHANG);
        if (pid > 0) { nchilds--; }
    } ;

    return NULL ;
}
```

```
void * receiver ( void )
{
    request_t p;
    int pid, nchilds=0;

    for (int i=0; i<MAX_REQUESTS; i++)
    {
        receive_request(&p);

        pid = fork();
        if (pid<0) { perror("Error in fork"); }
        if (pid==0) { reply_request(&p); exit(0); }
        if (pid!=0) { nchilds++; }
    }

    fprintf(stderr, "Wait for %d nchilds\n", nchilds);
    while (nchilds > 0)
    {
        pid = waitpid(-1, NULL, WNOHANG);
        if (pid > 0) { nchilds--; }
    } ;

    return NULL ;
}
```

Execution

```
$ time ./ej2
Receiving requests
Request 0 received after 3 seconds
Receiving requests
Sending request 0
Request 1 received after 1 seconds
Receiving requests
Sending request 1
Request 2 received after 2 seconds
Receiving requests
Sending request 2
Request 3 received after 0 seconds
Receiving requests
Sending request 3
Request 4 received after 3 seconds
Wait for 5 nchilds
Request 0 sent after 6 seconds
Sending request 4
Request 3 sent after 13 seconds
Request 1 sent after 17 seconds
Request 2 sent after 15 seconds
Request 4 sent after 15 seconds
Time: 24.086000

real    0m24.012s
user    0m9.569s
sys     0m5.459s
```

Comparison

Sequential	Process per req.	Thread per request	Pool of threads
62.11 sec.	24.08 sec.		

Problems

- A process must be started (fork) for each incoming request.
- A process must be terminated (exit) for each request that terminates.
- Excessive consumption of system resources
- No admission control.
 - ▣ Quality of service problems.

Solutions with threads

- Thread on demand (per request).
 - ▣ Each time a request is received, a thread is created.
- Pool of threads.
 - ▣ You have a fixed number of threads created.
 - ▣ Each time a request is received, a free thread already created is searched for to handle the request.
 - Communication through a request queue.

Contents

- Introduction (definitions):
 - ▣ Concurrent processes.
 - ▣ Concurrency, communication and synchronization
 - ▣ Critical section and Race conditions
 - ▣ Mutual exclusion and critical section.
- Synchronization mechanisms (I):
 - ▣ Initial basic primitives
 - ▣ Semaphores.
- Classic concurrency problems (I):
 - ▣ Producer-consumer
 - ▣ Reader-writers
- Synchronization mechanisms of threads (II)
 - ▣ Semaphores
 - System calls for semaphores.
 - Classic concurrency problems.
 - ▣ Mutex and condition variables
 - System calls for mutex.
 - Classic concurrency problems.

□ Concurrent server development

- ▣ Request servers.
- ▣ Process-based solution.
- ▣ **On-demand thread-based solution.**
- ▣ Thread pool-based solution.

On-demand (per request) threads

- There is a special thread in charge of receiving the requests.
- Each time a request arrives a thread is created, and **a copy of the request** is passed to the newly created thread.
 - ▣ It must be a copy of the request because the original request could be modified.

Implementation (1 / 3 main)

32

Alejandro Calderón Mateos 

```
#include <pthread.h>
#include <semaphore.h>
#include "request.h"

sem_t snchlds;

int main ( int argc, char *argv[] )
{
    struct timeval ts;
    long t1, t2;
    pthread_t thr;

    gettimeofday(&ts, NULL) ;
    t1 = (long)ts.tv_sec*1000+(long)ts.tv_usec/1000 ;
    sem_init(&snchlds, 0, 0);
    pthread_create(&thr, NULL, receiver, NULL);
    pthread_join(thr, NULL);
    sem_destroy(&snchlds);
    gettimeofday(&ts, NULL) ;
    t2 = (long)ts.tv_sec*1000+(long)ts.tv_usec/1000 ;

    printf("Time: %lf\n", (t2-t1)/1000.0);
    return 0;
}

void * service (void * p)
{
    request_t pet;

    copy_request(&pet, (request_t*)p);
    fprintf(stderr, "Starting service\n");
    reply_request(&pet);
    sem_post(&snchlds);
    fprintf(stderr, "Completing service\n");
    pthread_exit(0); return NULL;
}

void * receiver (void * param)
{
    const int MAX_REQUESTS = 5; int nservice = 0; int i;
    request_t p; pthread_t th_child;

    for (i=0; i<MAX_REQUESTS; i++) {
        receive_request(&p); nservice++;
        pthread_create(&th_child, NULL, service, &p);
    }

    for (i=0; i<nservice; i++) {
        fprintf(stderr, "Doing wait\n");
        sem_wait(&snchlds);
        fprintf(stderr, "After wait\n");
    }

    pthread_exit(0); return NULL;
}
```

```
#include <pthread.h>
#include <semaphore.h>
#include "request.h"
```

```
sem_t snchlds;
```

```
int main ( int argc, char *argv[] )
{
```

```
    struct timeval ts;
    long t1, t2;
    pthread_t thr;
```

```
    gettimeofday(&ts, NULL) ;
    t1 = (long)ts.tv_sec*1000+(long)ts.tv_usec/1000 ;
    sem_init(&snchlds, 0, 0);
    pthread_create(&thr, NULL, receiver, NULL);
    pthread_join(thr, NULL);
    sem_destroy(&snchlds);
    gettimeofday(&ts, NULL) ;
    t2 = (long)ts.tv_sec*1000+(long)ts.tv_usec/1000 ;
```

```
    printf("Time: %lf\n", (t2-t1)/1000.0);
    return 0;
```

```
}
```


Implementation (2/3 receiver)

33

Alejandro Calderón Mateos 

```
#include <pthread.h>
#include <semaphore.h>
#include "request.h"

sem_t snchlds;

int main ( int argc, char *argv[] )
{
    struct timeval ts;
    long t1, t2;
    pthread_t thr;

    gettimeofday(&ts, NULL) ;
    t1 = (long)ts.tv_sec*1000+(long)ts.tv_usec/1000 ;
    sem_init(&snchlds, 0, 0);
    pthread_create(&thr, NULL, receiver, NULL);
    pthread_join(thr, NULL);
    sem_destroy(&snchlds);
    gettimeofday(&ts, NULL) ;
    t2 = (long)ts.tv_sec*1000+(long)ts.tv_usec/1000 ;
    printf("Time: %lf\n", (t2-t1)/1000.0);
    return 0;
}

void * service (void * p)
{
    request_t pet;

    copy_request(&pet, (request_t*)p);
    fprintf(stderr, "Starting service\n");
    reply_request(&pet);
    sem_post(&snchlds);
    fprintf(stderr, "Completing service\n");
    pthread_exit(0); return NULL;
}

void * receiver (void * param)
{
    const int MAX_REQUESTS = 5; int nservice = 0; int i;
    request_t p; pthread_t th_child;

    for (i=0; i<MAX_REQUESTS; i++) {
        receive_request(&p); nservice++;
        pthread_create(&th_child, NULL, service, &p);
    }

    for (i=0; i<nservice; i++) {
        fprintf(stderr, "Doing wait\n");
        sem_wait(&snchlds);
        fprintf(stderr, "After wait\n");
    }

    pthread_exit(0); return NULL;
}
```

```
const int MAX_REQUESTS = 5;
```

```
void * receiver ( void * param )
{
```

```
    int nservice = 0;
```

```
    request_t p;
```

```
    pthread_t th_child;
```

```
    for (int i=0; i<MAX_REQUESTS; i++) {
```

```
        receive_request(&p);
```

```
        nservice++;
```

```
        pthread_create(&th_child, NULL, service, &p);
```

```
    }
```

```
    for (int i=0; i<nservice; i++) {
```

```
        fprintf(stderr, "Doing wait\n");
```

```
        sem_wait(&snchlds);
```

```
        fprintf(stderr, "After wait\n");
```

```
    }
```

```
    pthread_exit(0);
```

```
    return NULL;
```

```
}
```

Implementation (3/3 service)

34

Alejandro Calderón Mateos 

```
#include <pthread.h>
#include <semaphore.h>
#include "request.h"

sem_t snchlds;

int main ( int argc, char *argv[] )
{
    struct timeval ts;
    long t1, t2;
    pthread_t thr;

    gettimeofday(&ts, NULL) ;
    t1 = (long)ts.tv_sec*1000+(long)ts.tv_usec/1000 ;
    sem_init(&snchlds, 0, 0);
    pthread_create(&thr, NULL, receiver, NULL);
    pthread_join(thr, NULL);
    sem_destroy(&snchlds);
    gettimeofday(&ts, NULL) ;
    t2 = (long)ts.tv_sec*1000+(long)ts.tv_usec/1000 ;
    printf("Time: %lf\n", (t2-t1)/1000.0);
    return 0;
}
```

```
void * service (void * p)
{
    request_t pet;

    copy_request(&pet, (request_t*)p);
    fprintf(stderr, "Starting service\n");
    reply_request(&pet);
    sem_post(&snchlds);
    fprintf(stderr, "Completing service\n");
    pthread_exit(0); return NULL;
}
```

```
void * receiver (void * param)
{
    const int MAX_REQUESTS = 5; int nservice = 0; int i;
    request_t p; pthread_t th_child;

    for (i=0;i<MAX_REQUESTS;i++) {
        receive_request(&p); nservice++;
        pthread_create(&th_child, NULL, service, &p);
    }

    for (i=0;i<nservice;i++) {
        fprintf(stderr, "Doing wait\n");
        sem_wait(&snchlds);
        fprintf(stderr, "After wait\n");
    }

    pthread_exit(0); return NULL;
}
```

```
void * service ( void * p )
{
    request_t pet;

    memmove(&pet, (request_t *)p, sizeof(pet));
    fprintf(stderr, "Starting service\n");
    reply_request(&pet);
    sem_post(&snchlds);

    fprintf(stderr, "Completing service\n");
    pthread_exit(0);
    return NULL;
}
```

Thoughts

□ Can a race condition occur?

```
void * receiver ( void * param )
{
    const int MAX_REQUESTS = 5;
    int nservice = 0;
    request_t p;
    pthread_t th_child;

    for (int i=0; i<MAX_REQUESTS; i++) {
        receive_request(&p);
        nservice++;
        pthread_create(&th_child, NULL, service, &p);
    }

    for (int i=0; i<nservice; i++) {
        fprintf(stderr, "Doing wait\n");
        sem_wait(&snchilds);
        fprintf(stderr, "After wait\n");
    }

    pthread_exit(0);
    return NULL;
}
```

```
void * service ( void * p )
{
    request_t pet;

    memmove(&pet, (request_t *)p, sizeof(pet));
    fprintf(stderr, "Starting service\n");
    reply_request(&pet);
    sem_post(&snchilds);

    fprintf(stderr, "Completing service\n");
    pthread_exit(0);
    return NULL;
}
```

Execution

```
$ time ./ej3
Receiving requests
Request 0 received after 3 seconds
Receiving requests
Starting service
Sending request 0
Request 1 received after 1 seconds
Receiving requests
Starting service
Sending request 1
Request 2 received after 0 seconds
Receiving requests
Starting service
Sending request 3
Request 3 received after 3 seconds
Receiving requests
Starting service
Sending request 4
Request 4 received after 2 seconds
Doing wait
...
After wait
Doing wait
Request 1 sent after 15 seconds
Completing service
After wait
Doing wait
Request 0 sent after 17 seconds
Completing service
After wait
Time: 20.012000

real    0m20.012s
user    0m0.033s
sys     0m0.000s
```

Comparison

Sequential	Process per req.	Thread per request	Pool of threads
62.11 sec.	24.08 sec.	20.01 sec.	

Problem

- Thread creation and termination has a lower cost than process creation and termination, but it is still a cost.

- There is no admission control:
 - ▣ What happens if too many requests arrive or the requests received are not completed?

Thoughts

- Can a race condition occur?



Thoughts

40

Alejandro Calderón Mateos 

□ Can a race condition occur?

```
void * receiver (void * param)
```

```
request_t p; 
```

```
receive_request(&p);
```

```
nservice++;
```

```
pthread_create(&child, NULL, service, &p);
```

```
receive_request(&p);
```

```
nservice++;
```

```
pthread_create(&child, NULL, service, &p);
```

```
...
```




Thoughts

41

Alejandro Calderón Mateos 

□ Can a race condition occur?

```
void * receiver (void * param)
```

```
request_t p;    
receive_request(&p);  
nservice++;  
pthread_create(&child, NULL, service, &p);  
  
receive_request(&p);  
nservice++;  
pthread_create(&child, NULL, service, &p);  
...
```

Thoughts

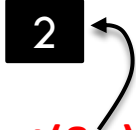
42

Alejandro Calderón Mateos 

□ Can a race condition occur?

```
void * receiver (void * param)
```

```
request_t p; 2  
receive_request(&p);  
nservice++;  
pthread_create(&child, NULL, service, &p);  
  
receive_request(&p);  
nservice++;  
pthread_create(&child, NULL, service, &p);  
...
```



Thoughts

43

Alejandro Calderón Mateos 

□ Can a race condition occur?

```
void * receiver (void * param)
```

```
request_t p; 2
```

```
receive_request(&p);  
nservice++;  
pthread_create(&child, NULL, service, &p);
```

```
receive_request(&p);  
nservice++;  
pthread_create(&child, NULL, service, &p);
```

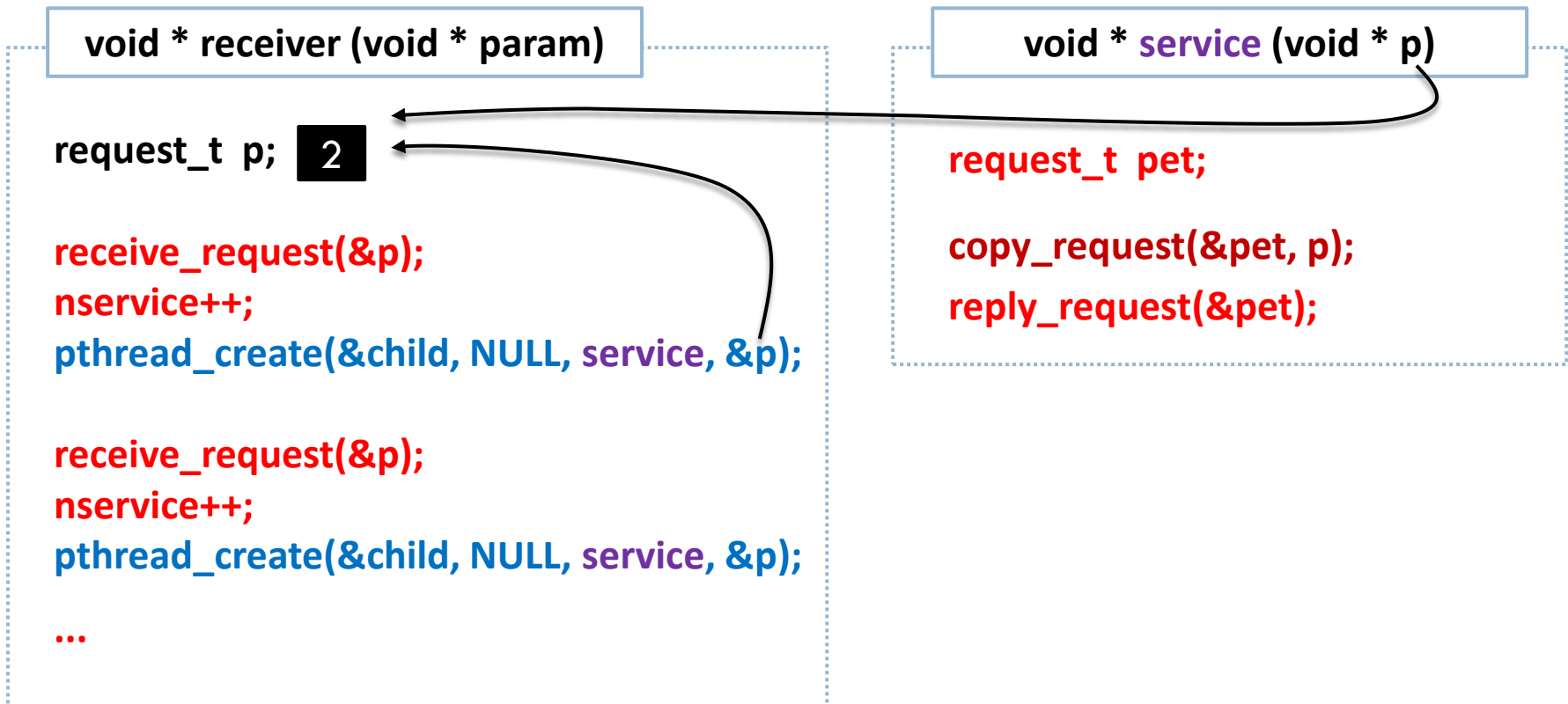
```
...
```

Thoughts

44

Alejandro Calderón Mateos 

□ Can a race condition occur?

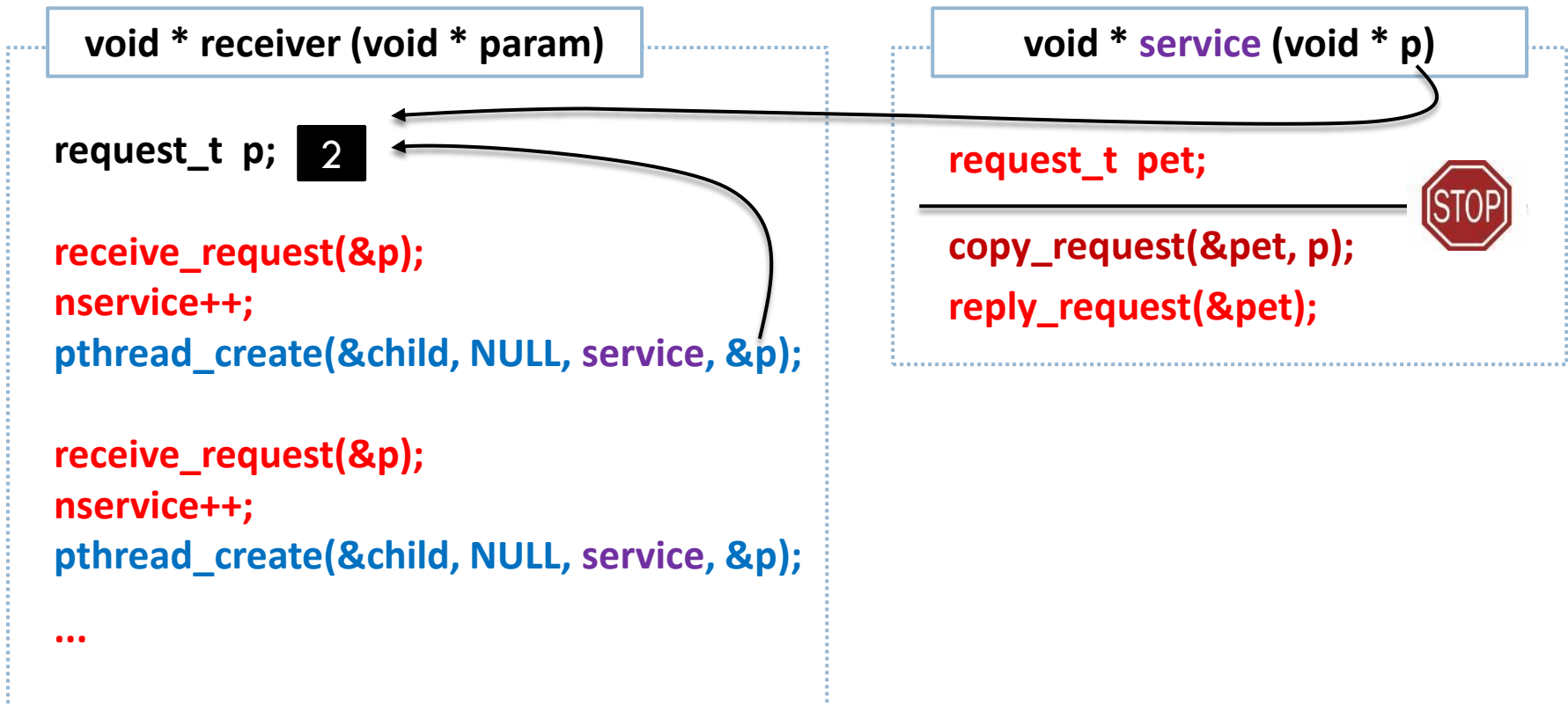


Thoughts

45

Alejandro Calderón Mateos 

□ Can a race condition occur?

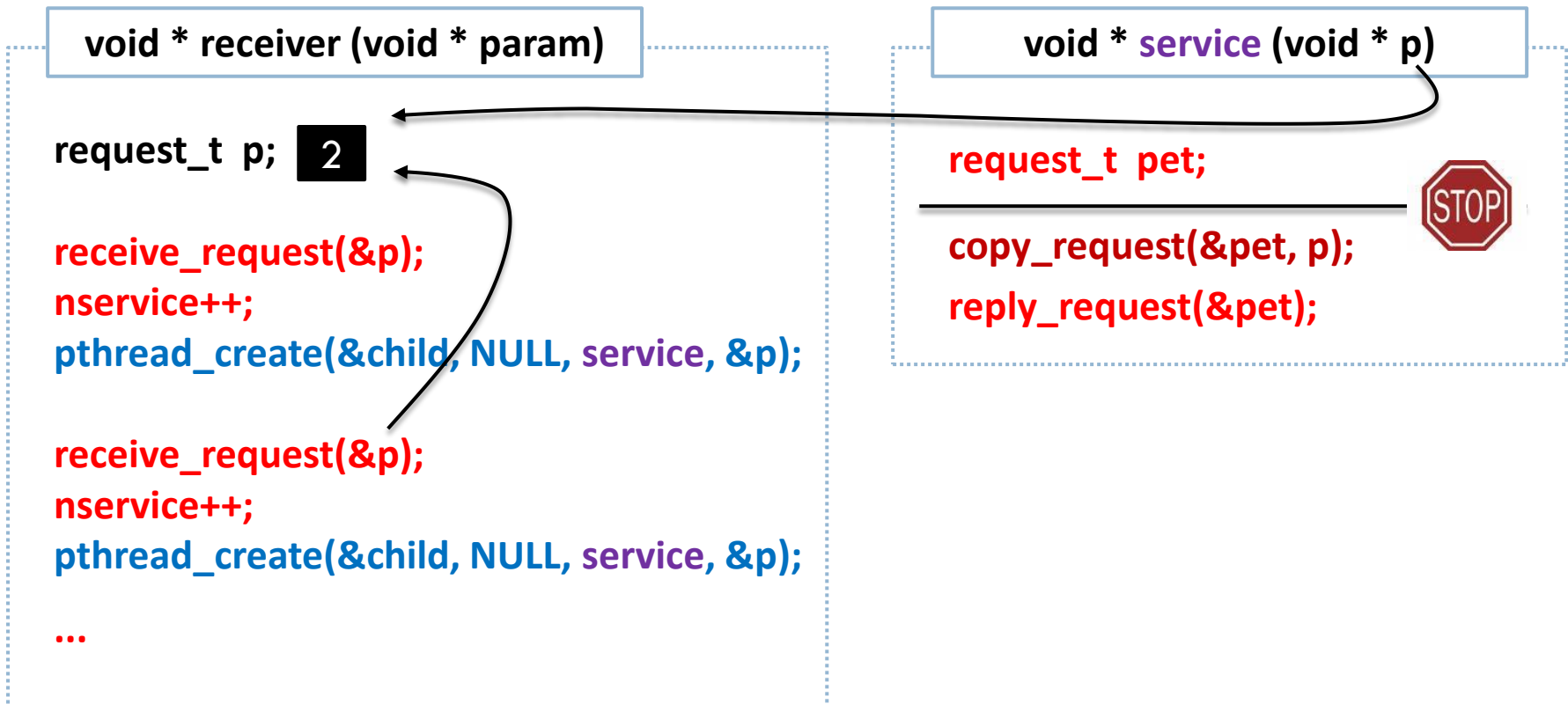


Thoughts

46

Alejandro Calderón Mateos 

□ Can a race condition occur?

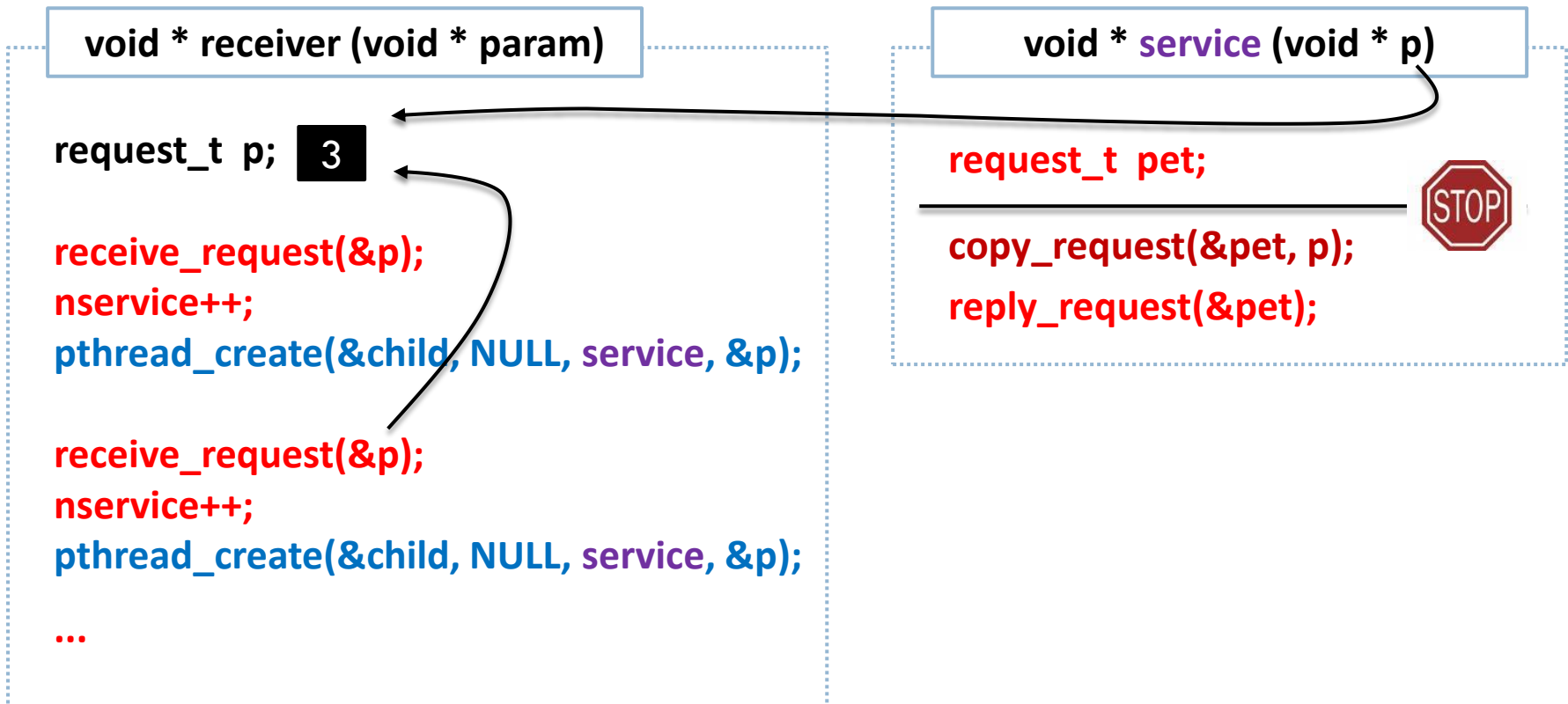


Thoughts

47

Alejandro Calderón Mateos 

□ Can a race condition occur?

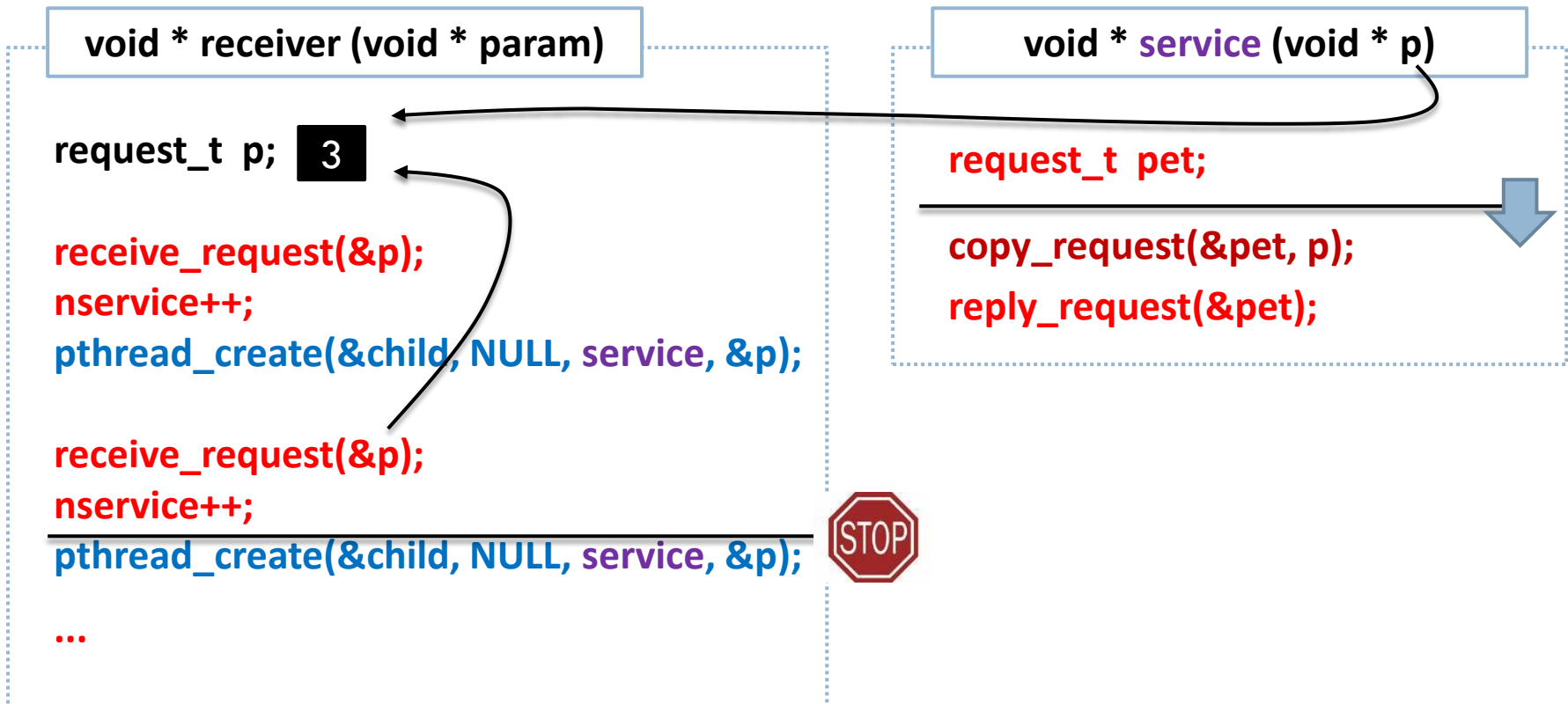


Thoughts

48

Alejandro Calderón Mateos 

□ Can a race condition occur?



Possible solution (mutex + condition)

□ Parent thread

```
lock(mutex); /* access to the resource */  
while (request is not copied)  
    wait(condition, mutex);  
mark request as no copied;  
unlock(mutex);
```

□ Child thread

```
lock(mutex);  
copy request  
mark request as copied;  
signal(condition);  
unlock(mutex);
```

Contents

- Introduction (definitions):
 - ▣ Concurrent processes.
 - ▣ Concurrency, communication and synchronization
 - ▣ Critical section and Race conditions
 - ▣ Mutual exclusion and critical section.
- Synchronization mechanisms (I):
 - ▣ Initial basic primitives
 - ▣ Semaphores.
- Classic concurrency problems (I):
 - ▣ Producer-consumer
 - ▣ Reader-writers
- Synchronization mechanisms of threads (II)
 - ▣ Semaphores
 - System calls for semaphores.
 - Classic concurrency problems.
 - ▣ Mutex and condition variables
 - System calls for mutex.
 - Classic concurrency problems.

□ Concurrent server development

- ▣ Request servers.
- ▣ Process-based solution.
- ▣ On-demand thread-based solution.
- ▣ **Thread pool-based solution.**

Threads pool

- A thread pool is a set of threads that you have created at the beginning to run a service:
 - ▣ Each time a request arrives, it is placed in a queue of pending requests.
 - ▣ All threads wait until there is a request in the queue and remove it for processing.

Implementation: main (1 / 3)

52

Alejandro Calderón Mateos 

```
#include "request.h"
#include <pthread.h>
#include <semaphore.h>

#define MAX_BUFFER 128
request_t buffer[MAX_BUFFER];

int n_elements = 0;
int pos_service = 0;
int fin = 0;
pthread_mutex_t mutex;
pthread_cond_t not_full;
pthread_cond_t no_empty;

void * receiver (void * param) ;
void * service (void * param) ;

int main()
{
    struct timeval ts;
    long t1, t2;
    pthread_t thr;
    pthread_t thr1(MAX_SERVICE);
    const int MAX_SERVICE = 5;

    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&not_full, NULL);
    pthread_cond_init(&no_empty, NULL);

    for (int i=0; i<MAX_SERVICE; i++) {
        pthread_create(&thr1[i], NULL, service, NULL);
    }
    sleep(1);

    gettimeofday(&ts, NULL);
    t1 = (long)ts.tv_sec*1000 + (long)ts.tv_usec/1000;
    pthread_create(&thr, NULL, receiver, NULL);
    pthread_join(thr, NULL);
    for (int i=0; i<MAX_SERVICE; i++) {
        pthread_join(thr1[i], NULL);
    }
    gettimeofday(&ts, NULL);
    t2 = (long)ts.tv_sec*1000 + (long)ts.tv_usec/1000;

    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&not_full);
    pthread_cond_destroy(&no_empty);
    printf("Time: %lf\n", (t2-t1)/1000.0);
    return 0;
}

void * receiver (void * param)
{
    const int MAX_REQUESTS = 5;
    request_t p;
    int pos=0;
    for (int i=0; i<MAX_REQUESTS; i++)
    {
        receive_request(&p);
        pthread_mutex_lock(&mutex);
        while (n_elements == MAX_BUFFER)
            pthread_cond_wait(&not_full, &mutex);
        buffer[pos] = p;
        pos = (pos+1) % MAX_BUFFER;
        n_elements++;
        pthread_cond_signal(&no_empty);
        pthread_mutex_unlock(&mutex);
    }
    fprintf(stderr, "Closing receiver\n");
    pthread_mutex_lock(&mutex);
    sleep(1);
    pthread_cond_broadcast(&no_empty);
    pthread_mutex_unlock(&mutex);
    fprintf(stderr, "Receiver closed\n");
    pthread_exit(0);
    return NULL;
}

void * service (void * param)
{
    request_t p;
    for (;;)
    {
        pthread_mutex_lock(&mutex);
        while (n_elements == 0)
        {
            if (fin==1) {
                fprintf(stderr, "Finalizing service\n");
                pthread_mutex_unlock(&mutex);
                pthread_exit(0);
            }
            pthread_cond_wait(&no_empty, &mutex);
        }
        // while
        printf(stderr, "Serving pos. %d\n", pos_service);
        p = buffer[pos_service];
        pos_service = (pos_service + 1) % MAX_BUFFER;
        n_elements--;
        pthread_cond_signal(&not_full);
        pthread_mutex_unlock(&mutex);
        reply_request(&p);
    }
    pthread_exit(0);
    return NULL;
}
```

```
#include "request.h"
#include <pthread.h>
#include <semaphore.h>
```

```
#define MAX_BUFFER 128
request_t buffer[MAX_BUFFER];
```

```
int n_elements = 0;
int pos_service = 0;
int fin=0;
```

```
pthread_mutex_t mutex;
pthread_cond_t not_full;
pthread_cond_t no_empty;
```

```
void * receiver (void * param) ;
void * service (void * param) ;
```

Implementation: main (2/3)

53

Alejandro Calderón Mateos



```
#include "request.h"
#include <pthread.h>
#include <semaphore.h>

#define MAX_BUFFER 128
request_t Buffer[MAX_BUFFER];

int n_elements = 0;
int pos_service = 0;
int fin = 0;
pthread_mutex_t mutex;
pthread_cond_t not_full;
pthread_cond_t no_empty;

void * receiver (void * param) ;
void * service (void * param) ;

int main()
{
    struct timeval ts;
    long t1, t2;
    pthread_t thr;
    pthread_t ths[MAX_SERVICE];
    const int MAX_SERVICE = 5;

    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&not_full, NULL);
    pthread_cond_init(&no_empty, NULL);

    for (int i=0; i<MAX_SERVICE; i++) {
        pthread_create(&ths[i], NULL, service, NULL);
    }

    gettimeofday(&ts, NULL);
    t1 = (long)ts.tv_sec*1000 + (long)ts.tv_usec/1000;
    pthread_create(&thr, NULL, receiver, NULL);
    pthread_join(thr, NULL);
    for (int i=0; i<MAX_SERVICE; i++) {
        pthread_join(ths[i], NULL);
    }

    gettimeofday(&ts, NULL);
    t2 = (long)ts.tv_sec*1000 + (long)ts.tv_usec/1000;

    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&not_full);
    pthread_cond_destroy(&no_empty);

    printf("Time: %lf\n", (t2-t1)/1000.0);
    return 0;
}

void * receiver (void * param)
{
    const int MAX_REQUESTS = 5;
    request_t p;
    int pos=0;
    for (int i=0; i<MAX_REQUESTS; i++)
    {
        receive_request(&p);
        pthread_mutex_lock(&mutex);
        while (n_elements == MAX_BUFFER)
            pthread_cond_wait(&not_full, &mutex);
        Buffer[pos] = p;
        pos = (pos+1) % MAX_BUFFER;
        n_elements++;
        pthread_cond_signal(&no_empty);
        pthread_mutex_unlock(&mutex);
    }
    fprintf(stderr, "Closing receiver\n");
    pthread_mutex_lock(&mutex);
    fin++;
    pthread_cond_broadcast(&no_empty);
    pthread_mutex_unlock(&mutex);
    fprintf(stderr, "Receiver closed\n");
    pthread_exit(0);
    return NULL;
}

void * service (void * param)
{
    request_t p;
    for (;;)
    {
        pthread_mutex_lock(&mutex);
        while (n_elements == 0)
        {
            if (fin==1) {
                fprintf(stderr, "Finalizing service\n");
                pthread_mutex_unlock(&mutex);
                pthread_exit(0);
            }
            pthread_cond_wait(&no_empty, &mutex);
        } // while
        print(stderr, "Serving pos. %d\n", pos_service);
        p = Buffer[pos_service];
        pos_service = (pos_service + 1) % MAX_BUFFER;
        n_elements--;
        pthread_mutex_unlock(&mutex);
        reply_request(&p);
    }
    pthread_exit(0);
    return NULL;
}
```

```
int main()
{
    struct timeval ts;
    long t1, t2;
    pthread_t thr;
    pthread_t ths[MAX_SERVICE];
    const int MAX_SERVICE = 5;

    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&not_full, NULL);
    pthread_cond_init(&no_empty, NULL);

    for (int i=0; i<MAX_SERVICE; i++) {
        pthread_create(&ths[i], NULL, service, NULL);
    }
    sleep(1);
```

Implementation: main (3/3)

54

Alejandro Calderón Mateos



```
#include "request.h"
#include "pthread.h"
#include "semaphore.h"

#define MAX_BUFFER 128
request_t Buffer[MAX_BUFFER];

int n_elements = 0;
int pos_service = 0;
int fin = 0;
pthread_mutex_t mutex;
pthread_cond_t not_full;
pthread_cond_t no_empty;

void * receiver (void * param) ;
void * service (void * param) ;

int main()
{
    struct timeval ts;
    long t1, t2;
    pthread_t thr;
    pthread_t ths[MAX_SERVICE];
    const int MAX_SERVICE = 5;

    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&not_full, NULL);
    pthread_cond_init(&no_empty, NULL);

    for (int i=0; i<MAX_SERVICE; i++) {
        pthread_create(&ths[i], NULL, service, NULL);
    }
    sleep(1);

    gettimeofday(&ts, NULL);
    t1 = (long)ts.tv_sec*1000 + (long)ts.tv_usec/1000;
    pthread_create(&thr, NULL, receiver, NULL);
    pthread_join(thr, NULL);
    for (int i=0; i<MAX_SERVICE; i++) {
        pthread_join(ths[i], NULL);
    }
    gettimeofday(&ts, NULL);
    t2 = (long)ts.tv_sec*1000 + (long)ts.tv_usec/1000;

    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&not_full);
    pthread_cond_destroy(&no_empty);
    printf("Time: %lf\n", (t2-t1)/1000.0);
    return 0;
}
```

```
void * receiver (void * param)
{
    const int MAX_REQUESTS = 5;
    request_t p;
    int pos=0;
    for (int i=0; i<MAX_REQUESTS; i++)
    {
        receive_request(&p);
        pthread_mutex_lock(&mutex);
        while (n_elements == MAX_BUFFER)
            pthread_cond_wait(&not_full, &mutex);
        Buffer[pos] = p;
        pos = (pos+1) % MAX_BUFFER;
        n_elements++;
        pthread_cond_signal(&no_empty);
        pthread_mutex_unlock(&mutex);
    }
    fprintf(stderr, "Closing receiver\n");
    pthread_mutex_lock(&mutex);
    fin = 1;
    pthread_cond_broadcast(&no_empty);
    pthread_mutex_unlock(&mutex);
    fprintf(stderr, "Receiver closed\n");
    pthread_exit(0);
    return NULL;
}

void * service (void * param)
{
    request_t p;
    for (;;)
    {
        pthread_mutex_lock(&mutex);
        while (n_elements == 0)
        {
            if (fin==1)
            {
                fprintf(stderr, "Finalizing service\n");
                pthread_mutex_unlock(&mutex);
                pthread_exit(0);
            }
            pthread_cond_wait(&no_empty, &mutex);
        }
        // while
        printf(stderr, "Serving pos. %d\n", pos_service);
        p = Buffer[pos_service];
        pos_service = (pos_service + 1) % MAX_BUFFER;
        n_elements--;
        pthread_cond_signal(&not_full);
        pthread_mutex_unlock(&mutex);
        reply_request(&p);
    }
    pthread_exit(0);
    return NULL;
}
```

```
gettimeofday(&ts, NULL) ;
t1 =(long)ts.tv_sec*1000+(long)ts.tv_usec/1000;
```

```
pthread_create(&thr, NULL, receiver, NULL);
pthread_join(thr, NULL);
for (int i=0; i<MAX_SERVICE; i++) {
    pthread_join(ths[i], NULL);
}
```

```
gettimeofday(&ts, NULL) ;
t2 =(long)ts.tv_sec*1000+(long)ts.tv_usec/1000 ;
```

```
pthread_mutex_destroy(&mutex);
pthread_cond_destroy(&not_full);
pthread_cond_destroy(&no_empty);
```

```
printf("Time: %lf\n", (t2-t1)/1000.0);
return 0;
```

```
}
```

Implementation: receiver

55

Alejandro Calderón Mateos



```
#include "request.h"
#include <pthread.h>
#include <semaphore.h>

#define MAX_BUFFER 128
request_t Buffer[MAX_BUFFER];

int n_elements = 0;
int pos_service = 0;
int fin = 0;
pthread_mutex_t mutex;
pthread_cond_t not_full;
pthread_cond_t no_empty;

void * receiver (void * param) ;
void * service (void * param) ;

int main()
{
    struct timeval ts;
    long t1, t2;
    pthread_t thr;
    pthread_t thr(MAX_SERVICE);
    const int MAX_SERVICE = 5;

    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&not_full, NULL);
    pthread_cond_init(&no_empty, NULL);

    for (int i=0; i<MAX_SERVICE; i++) {
        pthread_create(&thr[i], NULL, service, NULL);
    }
    sleep(1);

    gettimeofday(&ts, NULL);
    t1 = (long)ts.tv_sec*1000 + (long)ts.tv_usec/1000;
    pthread_create(&thr, NULL, receiver, NULL);
    pthread_join(thr, NULL);
    for (int i=0; i<MAX_SERVICE; i++) {
        pthread_join(thr[i], NULL);
    }
    gettimeofday(&ts, NULL);
    t2 = (long)ts.tv_sec*1000 + (long)ts.tv_usec/1000;

    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&not_full);
    pthread_cond_destroy(&no_empty);
    printf("Time: %lf\n", (t2-t1)/1000.0);
    return 0;
}

void * receiver (void * param)
{
    const int MAX_REQUESTS = 5;
    request_t p; int pos=0;
    for (int i=0; i<MAX_REQUESTS; i++)
    {
        receive_request(&p);
        pthread_mutex_lock(&mutex);
        while (n_elements == MAX_BUFFER)
            pthread_cond_wait(&not_full, &mutex);
        Buffer[pos] = p;
        pos = (pos+1) % MAX_BUFFER;
        n_elements++;
        pthread_cond_signal(&no_empty);
        pthread_mutex_unlock(&mutex);
    }
    fprintf(stderr, "Closing receiver\n");
    pthread_mutex_lock(&mutex);
    sleep(1);
    pthread_cond_broadcast(&no_empty);
    pthread_mutex_unlock(&mutex);
    fprintf(stderr, "Receiver closed\n");
    pthread_exit(0);
    return NULL;
}

void * service (void * param)
{
    request_t p;
    for (;;)
    {
        pthread_mutex_lock(&mutex);
        while (n_elements == 0)
        {
            if (fin==1) {
                fprintf(stderr, "Finalizing service\n");
                pthread_mutex_unlock(&mutex);
                pthread_exit(0);
            }
            pthread_cond_wait(&no_empty, &mutex);
        }
        // while
        printf(stderr, "Serving pos. %d\n", pos_service);
        p = Buffer[pos_service];
        pos_service = (pos_service + 1) % MAX_BUFFER;
        n_elements--;
        pthread_cond_signal(&not_full);
        pthread_mutex_unlock(&mutex);
        reply_request(&p);
    }
    pthread_exit(0);
    return NULL;
}
```

```
void * receiver (void * param)
{
    const int MAX_REQUESTS = 5;
    request_t p; int pos=0;

    for (int i=0; i<MAX_REQUESTS; i++)
    {
        receive_request(&p);
        pthread_mutex_lock(&mutex);
        while (n_elements == MAX_BUFFER)
            pthread_cond_wait(&not_full, &mutex);
        Buffer[pos] = p;
        pos = (pos+1) % MAX_BUFFER;
        n_elements++;
        pthread_cond_signal(&no_empty);
        pthread_mutex_unlock(&mutex);
    }
    fprintf(stderr, "Closing receiver\n");
    pthread_mutex_lock(&mutex);
    fin=1;
    pthread_cond_broadcast(&no_empty);
    pthread_mutex_unlock(&mutex);
    fprintf(stderr, "Receiver closed\n");
    pthread_exit(0);
    return NULL;
}
```

Implementation: service

56

Alejandro Calderón Mateos 

```
#include "request.h"
#include <pthread.h>
#include <semaphore.h>

#define MAX_BUFFER 128
request_t Buffer[MAX_BUFFER];

int n_elements = 0;
int pos_service = 0;
int fin = 0;
pthread_mutex_t mutex;
pthread_cond_t not_full;
pthread_cond_t no_empty;

void * receiver (void * param) ;
void * service (void * param) ;

int main()
{
    struct timeval ts;
    long t1, t2;
    pthread_t thr;
    pthread_t th(MAX_SERVICE);
    const int MAX_SERVICE = 5;

    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&not_full, NULL);
    pthread_cond_init(&no_empty, NULL);

    for (int i=0; i<MAX_SERVICE; i++) {
        pthread_create(&th[i], NULL, service, NULL);
    }
    sleep(1);

    gettimeofday(&ts, NULL);
    t1 = (long)ts.tv_sec*1000 + (long)ts.tv_usec/1000;
    pthread_create(&thr, NULL, receiver, NULL);
    pthread_join(thr, NULL);
    for (int i=0; i<MAX_SERVICE; i++) {
        pthread_join(th[i], NULL);
    }
    gettimeofday(&ts, NULL);
    t2 = (long)ts.tv_sec*1000 + (long)ts.tv_usec/1000;
    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&not_full);
    pthread_cond_destroy(&no_empty);
    printf("Time: %lf\n", (t2-t1)/1000.0);
    return 0;
}

void * receiver (void * param)
{
    const int MAX_REQUESTS = 5;
    request_t p;
    int pos=0;
    for (int i=0; i<MAX_REQUESTS; i++)
    {
        receive_request(&p);
        pthread_mutex_lock(&mutex);
        while (n_elements == MAX_BUFFER)
            pthread_cond_wait(&not_full, &mutex);
        Buffer[pos] = p;
        pos = (pos+1) % MAX_BUFFER;
        n_elements++;
        pthread_cond_signal(&no_empty);
        pthread_mutex_unlock(&mutex);
    }
    fprintf(stderr, "Closing receiver\n");
    pthread_mutex_lock(&mutex);
    sleep(1);
    pthread_cond_broadcast(&no_empty);
    pthread_mutex_unlock(&mutex);
    fprintf(stderr, "Receiver closed\n");
    pthread_exit(0);
    return NULL;
}

void * service (void * param)
{
    request_t p;
    for (;;)
    {
        pthread_mutex_lock(&mutex);
        while (n_elements == 0)
        {
            if (fin==1)
            {
                fprintf(stderr, "Finalizing service\n");
                pthread_mutex_unlock(&mutex);
                pthread_exit(0);
            }
            pthread_cond_wait(&no_empty, &mutex);
        }
        // while
        printf(stderr, "Serving pos. %d\n", pos_service);
        p = Buffer[pos_service];
        pos_service = (pos_service + 1) % MAX_BUFFER;
        n_elements--;
        pthread_cond_signal(&not_full);
        pthread_mutex_unlock(&mutex);
        reply_request(&p);
    }
    pthread_exit(0);
    return NULL;
}
```

```
void * service (void * param)
{
    request_t p;

    for (;;)
    {
        pthread_mutex_lock(&mutex);
        while (n_elements == 0)
        {
            if (fin==1) {
                fprintf(stderr, "Finalizing service\n");
                pthread_mutex_unlock(&mutex);
                pthread_exit(0);
            }
            pthread_cond_wait(&no_empty, &mutex);
        } // while
        printf(stderr, "Serving pos. %d\n", pos_service);
        p = Buffer[pos_service];
        pos_service = (pos_service + 1) % MAX_BUFFER;
        n_elements--;
        pthread_cond_signal(&not_full);
        pthread_mutex_unlock(&mutex);
        reply_request(&p);
    }
    pthread_exit(0);
    return NULL;
}
```


Comparison

Sequential	Process per req.	Thread per request	Pool of threads
62.11 sec.	24.08 sec.	20.01 sec.	?

OPERATING SYSTEMS: COMMUNICATION AND SYNCHRONIZATION AMONG PROCESSES



Concurrent server development