OPERATING SYSTEMS: PROCESS COMMUNICATION AND SYNCHRONIZATION

Developing concurrent servers

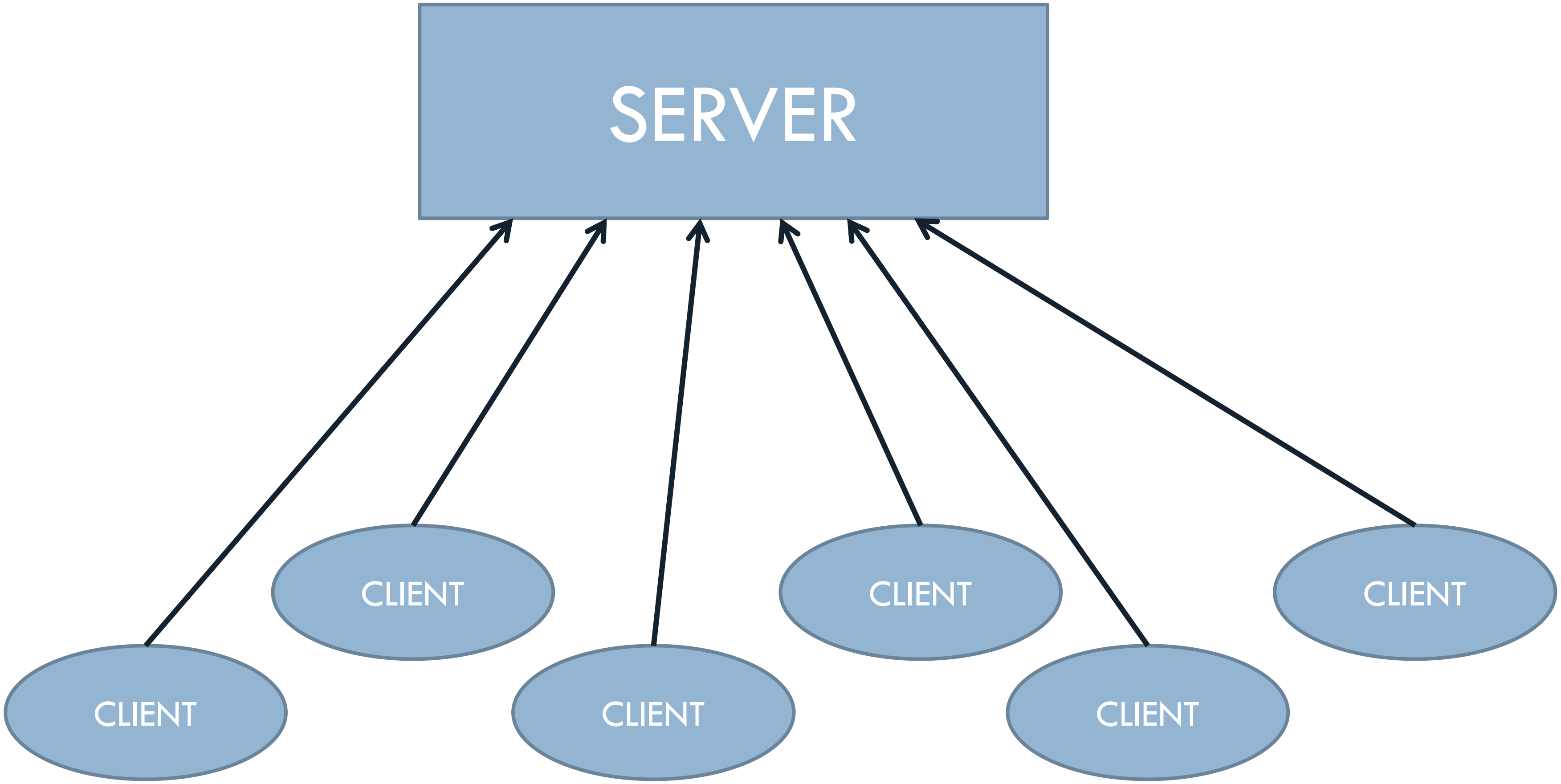
Contents

- **Request servers.**
- Process-based solution.
- Threads on demand solution.
- Thread pool solution.

Request servers

- Request servers are developed in many contexts:
  - Web server.
  - Database server.
  - Applications server.
  - File exchange servers.
  - Instant messaging applications.
  - …

Server



Problem: Request server

- A server receives requests that must be processed.
- Generic server structure:
  - Request **reception**.
    - Each request requires a certain amount of time in input/output operations to be received.
  - Request **processing**.
    - A certain amount of time for CPU processing.
  - Reply **sending**.
    - A certain amount of input/output operations to perform reply.

A library for testing

- To be able to evaluate solutions today we will use a simple library as a baseline example.

- Ideas:
  - Simulate requests reception.
  - Simulate request processing.
  - Simulate reply sending.

Base library


```c
#ifndef REQUEST_H
#define REQUEST_H

   struct request {
     long id;
     /* Rest of fields */
     int kind;
     char url[80];
     /* ... */
   };

   typedef struct request request_t;

   void get_request (request_t * r);
   void reply_request  (request_t * r);

#endif
```

Receiving requests


```
static long reqid = 0;

void receive_request(request_t * r)  {
  int delay;
  fprintf(stderr, “Receiving request\n");
  r->id = reqid++;

  /* Simulate I/O time*/
  delay = rand() % 5;
  sleep(delay);

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

Receiving requests


```
static long reqid = 0;

void receive_request(request_t * r)  {
  int delay;
  fprintf(stderr, "Receiving request\n");
  r->id = reqid++;

  /* Simulate I/O time*/
  delay = rand() % 5;
  sleep(delay);

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

In real code here some blocking call would be used to **receive** the request (e.g. from network)

Sending requests


```
void reply_request (request_t * r)  {
  int delay, i;
  double x;
  fprintf(stderr, “Sending reply %d\n", r->id);

  /* Simulate processing time */
  for (i=0;i<1000000;i++) { x = 2.0 * i; }

  /* Simulate I/O time */
  delay = rand() % 20;
  sleep(delay);

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

Sending requests


```
void reply_request (request_t * r)  {
  int delay, i;
  double x;
  fprintf(stderr, "Sending reply %d\n", r->id);

  /* Simulate processing time */
  for (i=0;i<1000000;i++) { x = 2.0 * i; }

  /* Simulate I/O time */
  delay = rand() % 20;
  sleep(delay);

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

Request processing would go here

Some blocking call would be used here to **reply** the request.

A first try

- Run forever the following sequence:
  - Receive a request.
  - Reply the request.

```
#include “request.h“

 int main()  {
   request_t r;

   for (;;) {
     receive_request(&r);
     reply_request(&r);
   }

  return 0;
}
```

Problems

- Request arrival.
  - If **two requests** arrive **at the same time** …
  - If **a request** arrives while **one is being processed** …

- Resource utilization.
  - How will CPU utilization be?

Measuring initial solution


```
#include “request.h"
#include <stdio.h>
#include <time.h>

int main() {
    int i;
   const int MAX_REQUESTS = 5;

   time_t t1,t2;
   double diff;

   request_t r;
```

```
  t1 = time(NULL);
  for (i=0;i<MAX_REQUESTS;i++) {
      receive_request(&r);
      reply_request(&r);
   }
   t2 = time(NULL);

   diff = difftime(t2,t1);
   printf(“Time: %lf\n",diff);

   return 0;
}
```

Execution


```
$ time app/seq
Receiving request
Request 0 received after 3 seconds
Sending reply 0
Request 0 sent after 6 seconds (computed 999999.000000)
Receiving request
Request 1 received after 2 seconds
Sending reply 1
Request 1 sent after 15 seconds (computed 999999.000000)
Receiving request
Request 2 received after 3 seconds
Sending reply 2
Request 2 sent after 15 seconds (computed 999999.000000)
Receiving request
Request 3 received after 1 seconds
Sending reply 3
Request 3 sent after 12 seconds (computed 999999.000000)
Receiving request
Request 4 received after 4 seconds
Sending reply 4
Request 4 sent after 1 seconds (computed 999999.000000)
Time: 62.000000

real  1m2.009s
user  0m0.008s
sys   0m0.000s
```

Evaluation

Normal	Processes	Thread per request	*Thread Pool*
62 seg.			

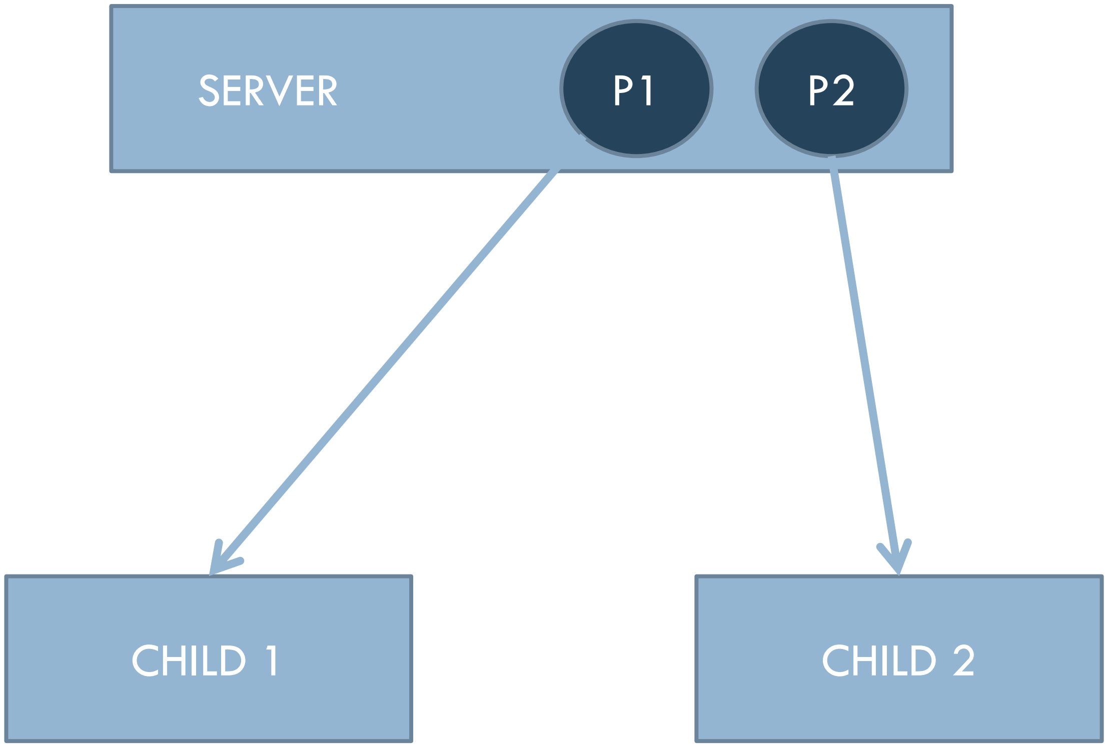
Contents

- Request severs.
- **Process-based solution.**
- Threads on demand solution.
- Thread pool solution.

Main idea

- Each time a request arrives a new child process is created:
  - **Child process** performs **request processing**.
  - **Parent process** waits for **next request**.

Process-based server



Implementation (1/3)


```
#include “request.h"
#include <stdio.h>
#include <time.h>
#include <sys/wait.h>

int main() {
  const int MAX_REQUESTS = 5;
  int i;
  time_t t1,t2;
  request_t r;
  int pid, nchildren=0;

  t1 = time(NULL);
```

Implementation (2/3)


```
for (i=0;i<MAX_REQUESTS; ++i)  {
    receive_request(&r);

    do {
      fprintf(stderr, "Checking children\n");
      pid = waitpid(-1, NULL, WNOHANG);
      if (pid>0) { nchildren--; }
   } while (pid > 0);

   pid = fork();
   if (pid<0)     { perror("Error creating child"); }
   if (pid==0)  { reply_request(&r); exit(0); }   /* CHILD */
   if (pid!=0)   { nchildren++; }                              /* PARENT*/
}
```

Implementation (3/3)


```
        fprintf(stderr, "Checking %d children\n", nchildren);
        while (nchildren>0) {
            pid = waitpid(-1, NULL, 0); /* Blocking wait */
            if (pid>0) { nchildren--;  }
        } ;

    t2 = time(NULL);

    double diff = difftime(t2,t1);
    printf("Time: %lf\n",diff);

    return 0;
}
```

Execution


```
$ time app/procsrv
Receiving request
Request 0 received after 3 seconds
Checking children
Receiving request
Sending reply 0
Request 1 received after 1 seconds
Checking children
Receiving request
Sending reply 1
Request 2 received after 2 seconds
Checking children
Receiving request
Request 3 received after 0 seconds
Checking children
Sending reply 2
Receiving request
Sending reply 3
Request 4 received after 3 seconds
Checking children
Checking 5 children
Sending reply 4
Request 0 sent after 6 seconds (computed 999999.000000)
Request 3 sent after 13 seconds (computed
        999999.000000)
Request 2 sent after 15 seconds (computed
        999999.000000)
Request 1 sent after 17 seconds (computed
        999999.000000)
Request 4 sent after 15 seconds (computed
        999999.000000)
Time: 24.000000

real  0m24.004s
user  0m0.001s
sys   0m0.012s
```

Evaluation

Normal	Processes	Thread per request	*Thread Pool*
62 seg.	24 seg.		

Problems

- A new process must be started (**fork**) for each incoming request.
- A process must be terminated (**exit**) for each served request.
- **Excessive** system **resources utilization.**
- No **admission control.**
  - Problems with **Quality of Service.**

Solutions with threads

- **On-demand threads**.
  - Each time a request is received a new thread is created.

- **Thread pool**.
  - A fixed number of pre-created threads.
  - Each time a request is received a free thread is assigned to serve the request.
    - Communication using a **request queue**.

Contents

On-demand

- A receiver thread is in charge of receiving the requests.

- Each time a request arrives a new thread is created, and a **copy** of the request is passed to the new created thread.
  - It must be a copy of the request as the original request could be modified.

Implementation


```
#include “request.h"
#include <stdio.h>
#include <time.h>
#include <pthread.h>
#include <semaphore.h>

sem_t snchildren;

int main()  {
  time_t t1, t2;
  double diff;
  pthread_t thr;
```

```
  t1 = time(NULL);
  sem_init(&snchildren, 0, 0);
  pthread_create(&thr, NULL,
                         receiver, NULL);
  pthread_join(thr, NULL);
  sem_destroy(&snchildren);
  t2 = time(NULL);

  diff = difftime(t2,t1);
  printf("Time: %f\n",diff);

  return 0;
}
```

Implementation: receiver


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

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

  while (nservice>0) {
    fprintf(stderr, "Performing wait\n");
    sem_wait(&snchildren);
    nservice--;
    fprintf(stderr, "Exiting from wait\n");
  }

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

Implementation: service


```
void * service(void * r) {
    request_t  req;

    copy_request(&req,(request_t*)r);


    fprintf(stderr, “Starting service\n");
    reply_request(&req);
    sem_post(&snchildren);

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

Execution


```
$ app/mtsrv
Receiving request
Request 0 received after 3 seconds
Receiving request
Starting service
Sending reply 1
Request 1 received after 1 seconds
Receiving request
Request 2 received after 0 seconds
Starting service
Sending reply 2
Receiving request
Starting service
Sending reply 3
Request 3 received after 3 seconds
Receiving request
Starting service
Sending reply 4
Request 4 received after 2 seconds
Performing wait
Starting service
Sending reply 4
Request 3 sent after 6 seconds (computed 999999.000000)
Finishing service for 3
Exiting from wait
Performing wait
Request 4 sent after 1 seconds (computed 999999.000000)
Finishing service for 4
Exiting from wait
Performing wait
```

Execution

Request 3 sent after 6 seconds (computed 999999.000000)

Finishing service for 3

Exiting from wait

Performing wait

Request 4 sent after 1 seconds (computed 999999.000000)

Finishing service for 4

Exiting from wait

Performing wait

Request 4 sent after 9 seconds (computed 999999.000000)

Finishing service for 4

Exiting from wait

Performing wait

Request 2 sent after 15 seconds (computed 999999.000000)

Finishing service for 2

Exiting from wait

Performing wait

Request 1 sent after 17 seconds (computed 999999.000000)

Finishing service for 1

Exiting from wait

Time: 20.000000

real	0m20.004s
user	0m0.013s
sys	0m0.000s

Question

- Can a race condition happen?

```
jdgarcia@gavilan:~/ejcpp/server/build$ valgrind --tool=helgrind app/mtsrv
==7970== Helgrind, a thread error detector
==7970== Copyright (C) 2007-2013, and GNU GPL'd, by OpenWorks LLP et al.
==7970== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==7970== Command: app/mtsrv
==7970==
....
==7970== Possible data race during write of size 4 at 0x5C1FE38 by thread #2
==7970== Locks held: none
==7970==    at 0x400F9B: receive_request (in /home/jdgarcia/ejcpp/server/build/app/mtsrv)
==7970==    by 0x400ED3: receiver (in /home/jdgarcia/ejcpp/server/build/app/mtsrv)
==7970==    by 0x4C30E26: mythread_wrapper (hg_intercepts.c:233)
==7970==    by 0x4E45181: start_thread (pthread_create.c:312)
==7970==    by 0x515547C: clone (clone.S:111)
```

Race condition

- Can a race condition happen?

```
void * receiver (void * param)

request_t  r; [redacted]

receive_request(&r);
nservice++;
pthread_create(&child, NULL, servicio, &r);

receive_request(&r);
nservice++;
pthread_create(&child, NULL, servicio, &r);

...
```

Race condition

- Can a race condition happen?

```
void * receiver(void * param)

request_t  r;

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

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

...
```

Race condition

- Can a race condition happen?

```
void * receiver(void * param)

request_t r;  2

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

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

...
```

Race condition

- Can a race condition happen?

```
void * receiver(void * param)

request_t r;   2

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

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

...
```

Race condition

- Can a race condition happen?

```
void * receiver(void * param)

request_t r;  2

receive_request(&r);
nservice++;
pthread_create(&hijo, NULL, service, &r);

receive_request(&r);
nservice++;
pthread_create(&hijo, NULL, service, &r);

...
```

```
void * service(void * r)

request_t req;

copy_request(&req, r);
reply_request(&req);
```

Race condition

- Can a race condition happen?

```
void * receiver(void * param)

request_t r;   2

receive_request(&r);
nservice++;
pthread_create(&hijo, NULL, service, &r);

receive_request(&r);
nservice++;
pthread_create(&hijo, NULL, service, &r);

...
```

```
void * service(void * r)

request_t req;

copy_request(&req, r);
reply_request(&req);
```

Race condition

- Can a race condition happen?

```
void * receiver(void * param)

request_t  r;   2

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

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

...
```

```
void * service(void * r)

request_t  req;
copy_request(&req, r);
reply_request(&req);
```

Race condition

- Can a race condition happen?

```
void * receiver(void * param)

request_t r;  3

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

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

...
```

```
void * service(void * r)

request_t req;
copy_request(&req, r);
reply_request(&req);
```

Race condition

- Can a race condition happen?

```
void * receiver(void * param)

request_t  r;  3

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

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

...
```


```
void * service(void * p)

request_t  req;

copy_request(&req, r);
reply_request(&req);
```

Implementation


```
#include “request.h"
#include <stdio.h>
#include <time.h>
#include <pthread.h>
#include <semaphore.h>

sem_t snchildren;
sem_t sparam;

int main() {
  time_t t1, t2;
  double diff;
  pthread_t thr;
```

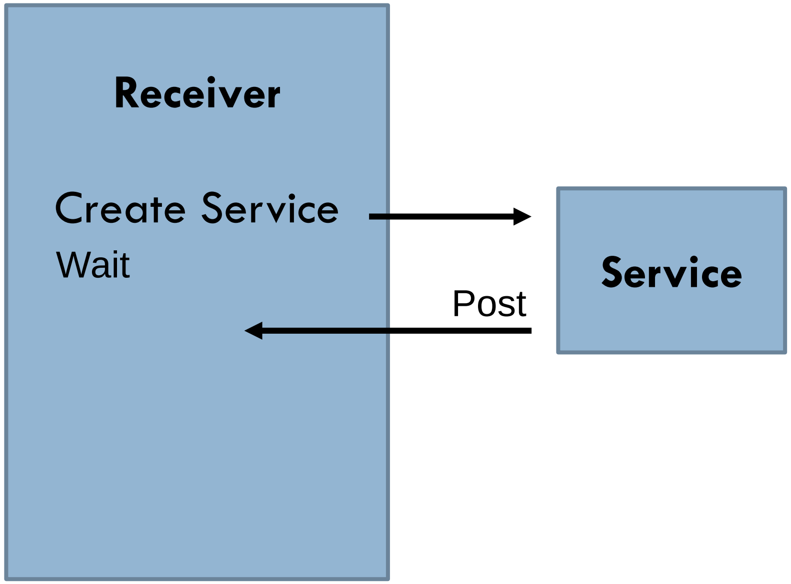
```
  t1 = time(NULL);
  sem_init(&snchildren, 0, 0);
  sem_init(&sparam, 0, 0);
  pthread_create(&thr, NULL,
                      receiver, NULL);
  pthread_join(thr, NULL);
  sem_destroy(&snchildren);
  sem_destroy(&sparam);
  t2 = time(NULL);

  diff = difftime(t2,t1);
  printf(“Time: %lf\n",diff);

  return 0;
}
```

Implementation: receiver


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

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

  while (nservice>0) {
    fprintf(stderr, “Performing wait\n");
    sem_wait(&snchildren);
    nservice--;
    fprintf(stderr, “Exiting from wait\n");
  }

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

Implementation: service


```
void * service(void * r) {
    request_t  req;

    copy_request(&req,(request_t*)r);
    sem_post(&sparam);

    fprintf(stderr, “Starting service\n");
    reply_request(&req);
    sem_post(&snchildren);

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



Evaluation

Normal	Processes	Thread per request	*Thread Pool*
54 seg.	17 seg.	20 seg.	

Problem

- Thread creation and termination has a cost which is lower than for processes,
  - But still a cost!
- No admission control:
  - What if many requests arrive or received requests do not finish?

Contents

- Request severs.
- Process based solution.
- Threads on demand solution.
- **Thread pool solution.**

Thread Pool

- A thread pool is a set of threads created at startup to run a service:
  - Each time a request arrives it is placed in the queue of pending requests.
  - All threads wait until some request is in the queue and they take it from the queue to process it.

Implementation: main (1/3)


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

#define MAX_BUFFER 128
request_t  buffer[MAX_BUFFER];
int  n_elements;
int  pos_service = 0;
pthread_mutex_t  mutex;
pthread_cond_t     non_full;
pthread_cond_t     non_empty;

pthread_mutex_t  mend;
int  end=0;
```

Implementation: main (2/3)


```
int main()
{
  time_t t1, t2;
  double diff;
  pthread_t thr;
  const int MAX_SERVICE = 5; int i;
  pthread_t ths[MAX_SERVICE];

  t1 = time(NULL);

  pthread_mutex_init(&mutex,NULL);
  pthread_cond_init(&non_full,NULL);
  pthread_cond_init(&non_empty,NULL);
  pthread_mutex_init(&mend,NULL);

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

Implementation: main (3/3)


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

  pthread_mutex_destroy(&mutex);
  pthread_cond_destroy(&non_full);
  pthread_cond_destroy(&non_empty);
  pthread_mutex_destroy(&mend);

  t2 = time(NULL);

  diff = difftime(t2,t1);
  printf("Time: %lf\n",diff);

  return 0;
}
```

Implementation: receiver (1/2)

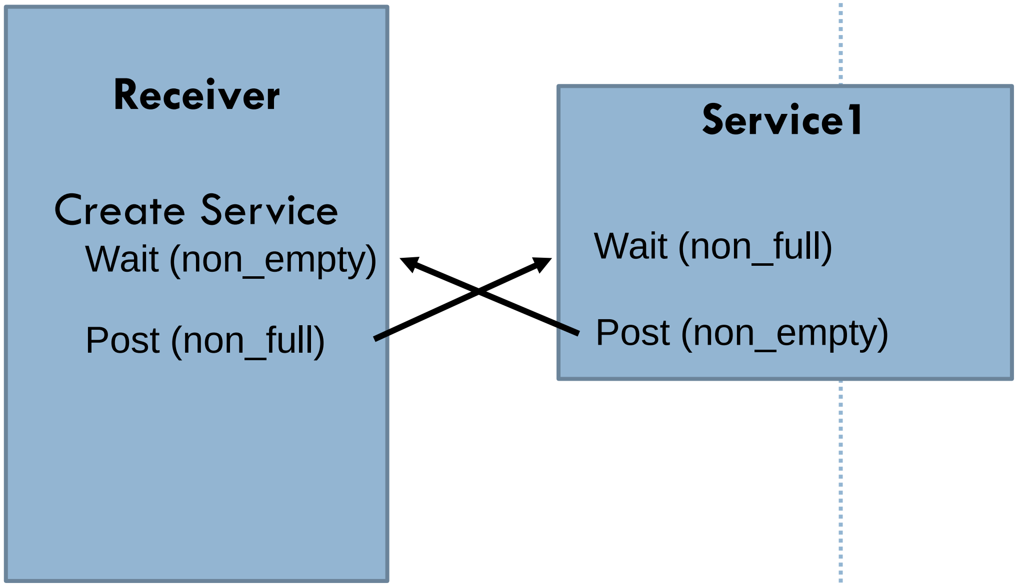

```
void * receiver (void * param)
 {
     const int MAX_REQUESTS = 5;
     request_t r;
     int i, pos=0;

    for (i=0;i<MAX_REQUESTS;i++)
    {
        receive_request(&r);
        pthread_mutex_lock(&mutex);
        while (n_elements == MAX_BUFFER)
                pthread_cond_wait(&non_full, &mutex);
        buffer[pos] = r;
        pos = (pos+1) % MAX_BUFFER;
        n_elements++;
        pthread_cond_signal(&non_empty);
        pthread_mutex_unlock(&mutex);
    }
```

Implementation: receiver (2/2)


```
    fprintf(stderr,"Finishing receiver\n");
    pthread_mutex_lock(&mend);
    end=1;
    pthread_mutex_unlock(&mend);

    pthread_mutex_lock(&mutex);
    pthread_cond_broadcast(&non_empty);
    pthread_mutex_unlock(&mutex);

    fprintf(stderr, "Finished receiver\n");
    pthread_exit(0);
    return NULL;
} /* receiver*/
```

Implementation: service (1/2)


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

    for (;;)  {
        pthread_mutex_lock(&mutex);
        while (n_elements == 0) {
            if (end==1) {
                fprintf(stderr,"Finishing service\n");
                pthread_mutex_unlock(&mutex);
                pthread_exit(0);
            }
            pthread_cond_wait(&non_empty, &mutex);
        } // while
```



Implementation: service (2/2)


```
        fprintf(stderr, “Serving position %d\n", pos_service);
        r = buffer[pos_service];
        pos_service = (pos_service + 1) % MAX_BUFFER;
        n_elements --;
        pthread_cond_signal(&non_full);
        pthread_mutex_unlock(&mutex);
        reply_request(&r);
    }

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

Evaluation

Normal	Processes	Thread per request	*Thread Pool*
54 seg.	17 seg.	20 seg.	12 seg