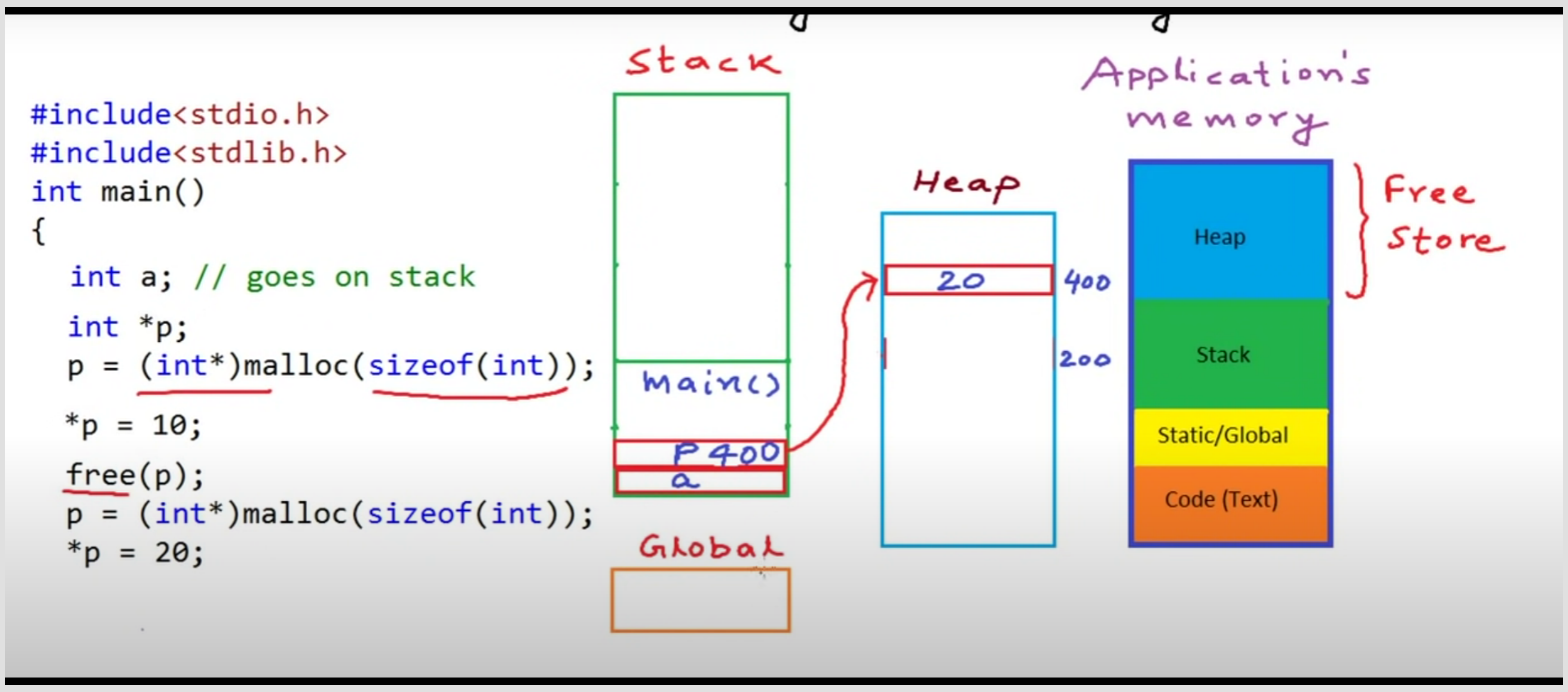
Memory:
When a computer program runs, it needs memory to store data and instructions. Memory can be divided into two main types: stack and heap.

Stack: Stack memory is used for storing local variables, function parameters, and return addresses. It's organized in a last-in-first-out (LIFO) manner, meaning the most recently allocated memory is the first to be deallocated.
Heap: Heap memory is used for dynamic memory allocation. Unlike the stack, the heap memory is more flexible and can be allocated and deallocated in any order.

Static vs. Dynamic Memory Allocation:

Static Memory Allocation: In static memory allocation, memory is allocated at compile time. This means the size of memory needed by the program is determined before the program runs, and it remains fixed throughout the execution.
Dynamic Memory Allocation: In dynamic memory allocation, memory is allocated at runtime (when the program is running). This allows programs to allocate memory as needed during execution, and the size of memory can change dynamically.
malloc() Function:

malloc() is a function in the C programming language (and similar languages) used for dynamic memory allocation.
It stands for "memory allocation".
Here's how you use malloc():



malloc() takes the number of bytes to allocate as an argument and returns a pointer to the beginning of the allocated memory block.
It's important to check if malloc() returns a null pointer, which indicates that the memory allocation failed due to insufficient memory.

```
#include <stdio.h>

int main() {
    // Static memory allocation
    int staticArray[5]; // Array size is fixed at compile time

    // Assigning values to the statically allocated array
    for (int i = 0; i < 5; i++) {
        staticArray[i] = i * 2;
    }

    // Accessing and printing the elements of the statically allocated array
    for (int i = 0; i < 5; i++) {
        printf("Element %d: %d\n", i, staticArray[i]);
    }

    return 0;
}
```

In this example, the size of the array staticArray is fixed at compile time, and the user cannot enter its size at runtime.

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    // Dynamic memory allocation
    int *dynamicArray;
    int size;

    // Prompting the user to enter the size of the array
    printf("Enter the size of the array: ");
    scanf("%d", &size);

    // Allocating memory for the dynamic array based on user input
    dynamicArray = (int *)malloc(size * sizeof(int));

    // Checking if memory allocation was successful
    if (dynamicArray == NULL) {
        printf("Memory allocation failed.\n");
        return 1; // Returning error code
    }

    // Assigning values to the dynamically allocated array
    for (int i = 0; i < size; i++) {
        dynamicArray[i] = i * 2;
    }

    // Accessing and printing the elements of the dynamically allocated array
    printf("Elements of the dynamically allocated array:\n");
    for (int i = 0; i < size; i++) {
        printf("Element %d: %d\n", i, dynamicArray[i]);
    }

    // Freeing the dynamically allocated memory
    free(dynamicArray);

    return 0;
```

In this example, the user is prompted to enter the size of the array at runtime. The program then dynamically allocates memory for the array based on the user's input, allowing for more flexibility in the size of the array.

Basics Of Memory Allocation:
Memory: When a computer program runs, it needs memory to store data and instructions. Memory can be divided into two main types: STACK AND HEAP.

Stack: Stack memory is used for storing local variables, function parameters, and return addresses. It's organized in LIFO manner, meaning the most recently allocated memory is the first memory to be deallocated.
For example, int char[2] here it's static cuz we define the memory required. There can be memory wastage if add lesser elements to array than defined size or their can be a crash if it's more than the mem allocated.

Heap: Heap memory is used for dynamic memory allocation. Unlike the stack, the heap memory is more flexible and can be allocated and deallocated in any order. We can control memory at runtime with functions like malloc, calloc etc.

System and Device Programming

Review Exercises

Stefano Quer

Dipartimento di Automatica e Informatica

Politecnico di Torino

Exercise 01 (C and UNIX IO)

Please, recall the SDP standards
C, POSIX, C++

- An ASCII file stores a sequence of records
  - Each record includes 3 fields
    - An integer value, a string, and a real number
  - All three fields have variable size and are separated by a variable number of white spaces

```
15345 Acceptable 26.50
146467 Average 23.75
. . .
```

Exercise 01 (C and UNIX IO)

- Write a segment of C code that stores the records into a **list of structures** of type record_t, using
  - The C library
  - The the POSIX/UNIX library

```
15345 Acceptable 26.50
146467 Average 23.75
. . .
```

```
#define N 100

typedef struct record_s {
  int i;
  char s[N];
  float f;
  struct record_s *next;
} record_t;
```

this line declares a mem named next within the structure. Here next is a pointer to another record_s

The typedef keyword is used to create an alias or a new name for an existing data type. In this case, it's creating a new name record_t that refers to the structure def that follows.

- Observation
  - Keep in mind that the UNIX system call **read** manipulates files in binary form

Solution 1

C Library

Simple solution:
The file has a strict and regular format

record_t is a struct defined in the previous slide that specifies the format for each data record within the file.

```
#define L 2000
```

This represents the maximum length a line within a file can have

.... indicates the file name, fp is a pointer.

We can also use
`fscanf(line,"%ld %s %f",...)!=EOF`

```
fp = fopen (..., "r");
head = NULL;
while (fgets (line, L, fp) != NULL) {
  r = (record_t *) malloc (1 * sizeof (record_t));
  if (r==NULL) { ... error ... }
  sscanf (line, "%ld %s %f", &r->n, r->s, &r->f);
  r->next = head; head = r;
}
fclose (fd);
```

head initializes a pointer to null. This pointer will likely be used to maintain a linked list of the parsed records

this iterates as long as fgets reads a line from fp successfully. Reads lines untill max of L-1 characters,

The cast (record_t *) ensures that the memory allocated by malloc is interpreted as a pointer to a record_t

This is a function call to malloc, which allocates memory dynamically on the heap. The argument to malloc specifies the number of bytes to allocate. Here:

If the allocation is successful, malloc returns the memory address of the allocated block. This address is then assigned to the pointer variable r.
r now becomes a pointer that points to the newly allocated memory, allowing you to access and manipulate the data stored within that memory using the structure members (e.g., r->i, r->s, r->f).

stack lifo structure where you insert element on top.

Solution 2

Posix is not able to read using a string , so you have to compilacete the code a little bit.

POSIX Library

Complex solution:
read manipulates binary files

```
typedef struct tmp_s {
  int i;
  char s[N];
  float f;
} tmp_t;

tmp_t r;

fd = open (..., O_RDONLY);
head = NULL;
while (read (fd, &r, sizeof (tmp_t)) != 0) {
  ...
}
close (fd);
```

The code uses the POSIX library, which is a collection of standards for operating system APIs. This particular code snippet appears to be using the read function from the POSIX library to read data from a binary file.

Buggy

Records do not have a fixed length
(sizeof (tmp_t))

Buggy: There is a comment in the code that says "Buggy". This might indicate that the solution has an error. Potentially, the issue could be related to the fact that the size of the records being read from the binary file may not match the size of the tmp_t structure.

Records without a fixed size: The comment "Records do not have a fixed length" indicates that the size of the data records in the binary file being read may vary. This could be an issue if the code is trying to read the data into a fixed-size structure (tmp_t).

Solution 3

POSIX Library

```
fd = open (..., O_RDONLY);
head = NULL;

close (fd);
```

Read entire line and convert

```
do {
  i = 0;
  do {
    not_end = read (fd, &c, sizeof(char));
    if (not_end != 0) line[i++] = c;
  } while (c!='\n' && not_end);
  if (not_end) {
    r = (record_t *) malloc (1 * sizeof (record_t));
    sscanf (line, "%ld %s %f", &r->n, r->s, &r->f);
    r->next = head; head = r;
  }
} while (not_end);
```

Solution 4

POSIX Library

Read and convert each field (long int, string, float)

```
do {
  n = 0;
  do {
    not_end = read (fd, &c, sizeof(char));
    if (c!=' ' && not_end) n = n * 10 + ((int) (c-'0'));
  } while (c!=' ' && not_end);
  c = skip_spaces (fd);
  i = 0; tmp1[i++] = c;
  do {
    not_end = read (fd, &c, sizeof(char));
    if (c!=' ' && not_end) tmp1[i++] = c;
  } while (c!=' ' && not_end);
  tmp1[i] = '\0';
```

Solution 4

```
  c = skip_spaces (fd);
  i = 0;
  tmp2[i++] = c;
  do {
    not_end = read (fd, &c, sizeof(char));
    if (c!=' ' && not_end) tmp2[i++] = c;
  } while (c!='\n' && not_end);
  if (not_end) {
    r = (record_t *) malloc (1 * sizeof (record_t));
    r->n = n; strcpy (r->s, tmp1);  r->f = atof (tmp2);
    r->next = head;
    head = r;
  }
} while (not_end);
```

A control flow graph is a graphical representation of the flow of control within a program. It consists of nodes representing basic blocks of code and edges representing control flow between these basic blocks. The CFG helps visualize the possible paths of execution in a program.

Exercise 02 (fork)

Process Generation Tree: In a program using process creation system calls like fork(), a process generation tree represents the hierarchy of parent and child processes. Each node in the tree represents a process, with child processes branching off from parent processes. The tree helps visualize how processes are created and how they relate to each other.

- Draw the control flow graph and the process generation tree of the following C code snippet
- Indicate the generated output

Control Flow Graph (CFG):
We'll identify the basic blocks of code and the control flow between them.
We'll represent the basic blocks as nodes and the control flow between them as directed edges.
Process Generation Tree:
We'll identify the parent and child processes created by the fork() calls.
We'll represent each process as a node in the tree, with child processes branching off from parent processes,

```
int main () {
  int i, pid;
  setbuf(stdout,0);
  for (i=1; i<3; i++) {
    pid = fork ();
    i++;
    if (pid!=0) {
      fork();
      printf("X");
    }
  }
  printf("X");
  return 1;
}
```

p1, p2

Fork is used to create a clone of the process. So after the fork you have two process executing the same code and value

***IMP
If fork() returns a negative value, the creation of a child process was unsuccessful (typically due to a lack of system resources).
If fork() returns a zero, then the function has been called from the newly created child process.
If fork() returns a positive value, this is the process ID of the child process and this return happens in the parent process.

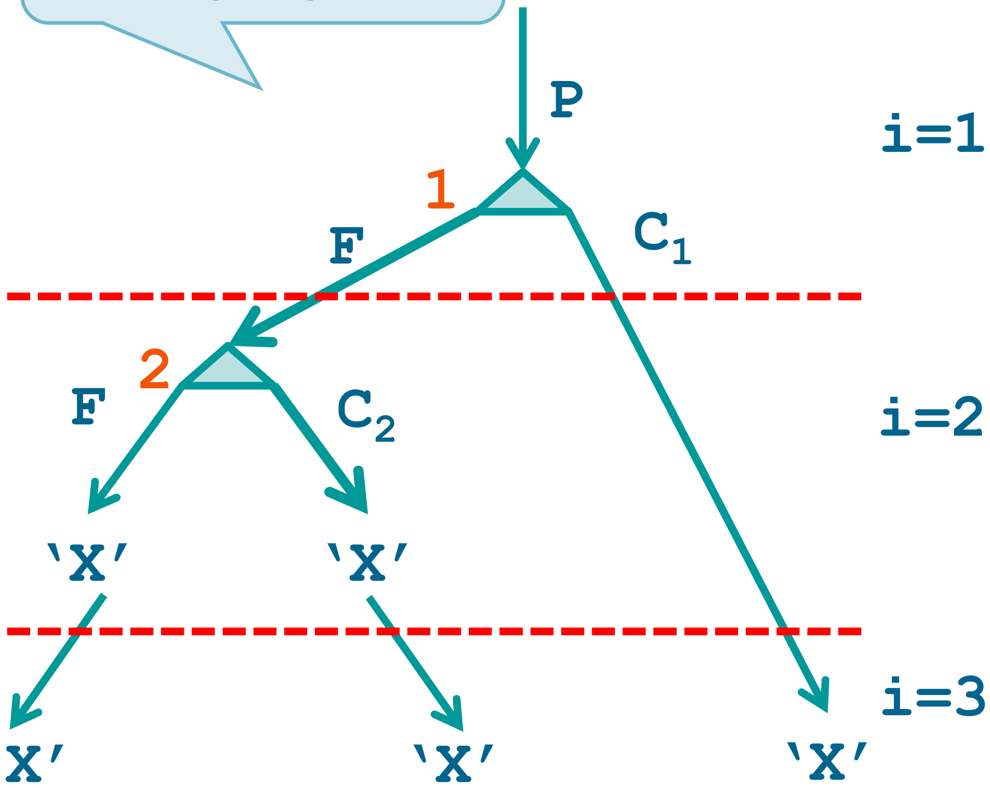
Solution

```
for (i=1; i<3; i++) {
  pid = fork ();  // 1
  i++;
  if (pid!=0) {
    fork();
    printf("X");  // 2
  }
}
printf("X");
```

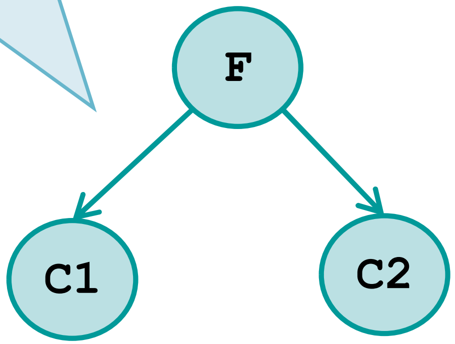
Output

```
XXXXX
```

Control Flow Graph (CFG)



Process Generation Tree



Exercise 03 (fork, exec, system)

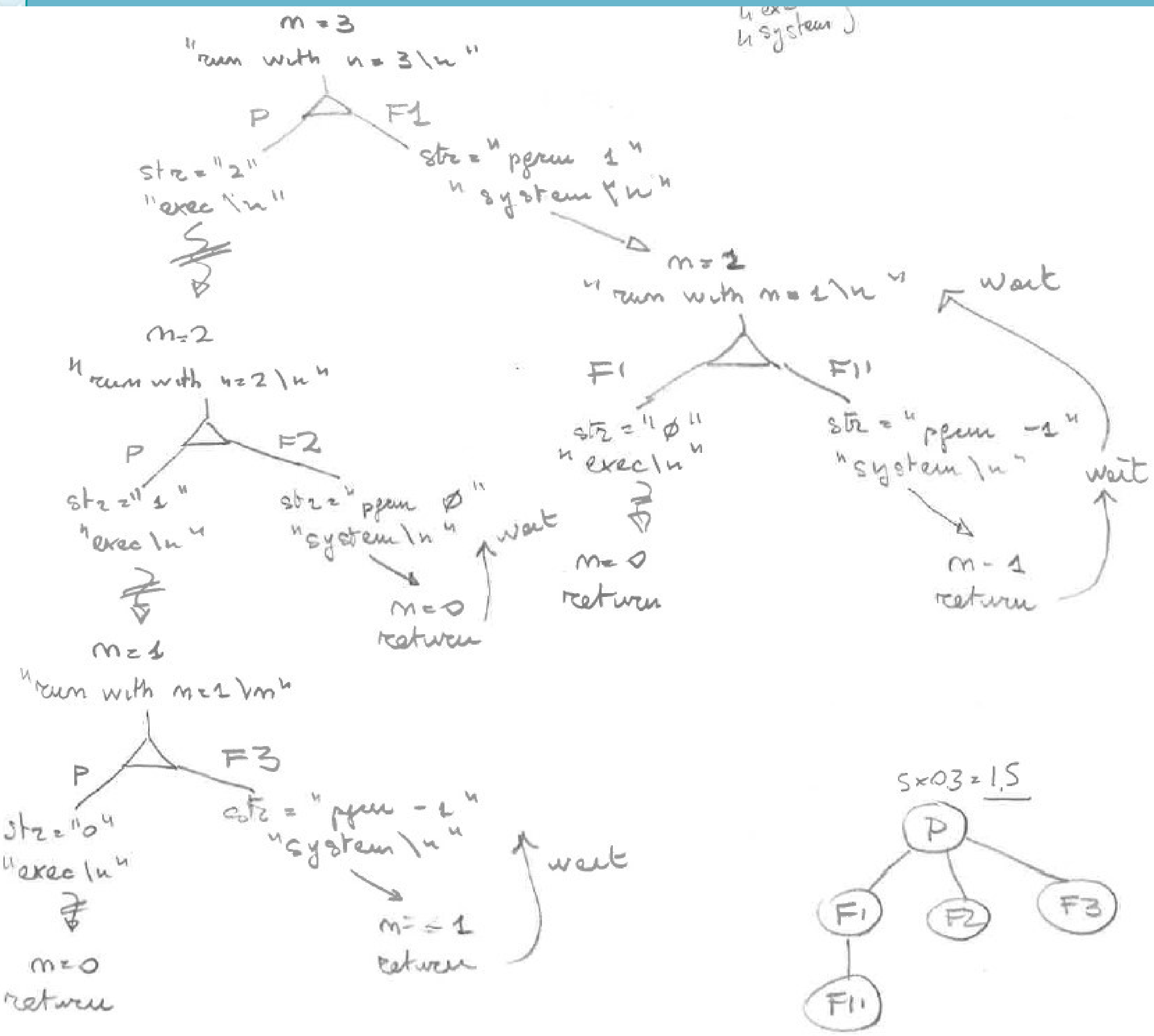
- Assume the following program runs with a unique parameter on the command line, i.e., the integer value 3
- Draw
  - The control flow graph
  - The process generation tree
- Indicate what it displays on standard output

Exercise 03 (fork, exec, system)

```
int main (int argc, char *argv[]) {
  char str[20];
  int n = atoi (argv[1]);
  setbuf (stdout, 0);
  if (n<=0) return (1);
  printf ("run with n=%d\n", n);
  if (fork()>0) {
    sprintf (str, "%d", n-1);
    printf ("exec\n");
    execlp (argv[0], argv[0], str, (char *) 0);
  } else {
    sprintf (str, "%s %d", argv[0], n-2);
    printf ("system\n");
    system (str);
  }
  wait ((int *) 0);
   return (1);
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
```

Solution


n = 3
"run with n = 3\n"
P
F1
str = "2"
"exec \n"
str = "pgrm 1"
"system \n"
n = 1
"run with n = 1\n"
wait
n = 2
"run with n = 2\n"
F1
F11
str = "∅"
"exec\n"
str = "pgrm -1"
"system \n"
P
F2
wait
str = "1"
"exec \n"
str = "pgrm ∅"
"system \n"
wait
n = 0
return
n = 1
return
n = 0
return
n = 1
"run with n = 1 \n"
P
F3
str = "0"
"exec \n"
str = "pgrm -1"
"system \n"
wait
n = -1
return
n = 0
return
5×0.3 = 1.5
P
F1
F2
F3
F11

Solution

- Output
  - run with n=3 1 time
  - run with n=2 1 time
  - run with n=1 2 times
  - exec 4 times
  - system 4 times

Exercise 04 (orphans and zombies)

- Illustrate the following concepts
  - Orphan process
  - Zombie process
- Report two code segments
  - The first one to generate an orphan process
  - The second one to generate a zombie process

Solution

- A process
  - Is said to be an **orphan** process if its parent ends before waiting for it
    - When the parent ends, the orphan process is "adopted" by a special process, typically the init process
  - Is said to be a **zombie** process if it terminates before its parent issuer a wait
    - Its Process Control Bock (PCB) is stored by the OS to be avalable to the parent when the parent ends
    - It wastes OS resources

Solution

Parent process

Orphan process

```
if(fork()){
    exit(0);
} else{
    // This process waits for a second
    // Thus, the parent should terminate before it
    sleep(1);
    exit(0);
}
```

Parent process

Zombie process

```
if(fork()) {
  // This process waits for a second
  // Thus, the child should terminate before it
  sleep(1);
  exit(0);
} else {
  exit(0);
}
```

Solution

- An init and a zombie process on a Linux machine

```
USER   PID %CPU %MEM    VSZ   RSS TTY    STAT START   TIME COMMAND
root     1  0.3  0.0 166876 12112 ?      Ss   22:56   0:01 /sbin/init splash
root     2  0.0  0.0      0     0 ?      S    22:56   0:00 [kthreadd]
root     3  0.0  0.0      0     0 ?      I<   22:56   0:00 [rcu_gp]
root     4  0.0  0.0      0     0 ?      I<   22:56   0:00 [rcu_par_gp]
...
quer 2274  0.0  0.0  11540  5840 pts/0 Ss   22:57   0:00 bash
quer 3065  2.1  0.1 770444 74016 pts/0 Sl   23:00   0:03 emacs
quer 3082  0.0  0.0   2772   948 pts/0 S    23:00   0:00 ./a
quer 3084  0.0  0.0      0     0 pts/0 Z    23:00   0:00 [a] <defunct>
quer 3119  0.0  0.0  12992  3784 pts/0 R+   23:02   0:00 ps -aux
```

Exercise 05 (signals)

- A process P
  - Generates two child processes P1 and P2
  - It goes into a wait state and
    - Every time it receives a message from P1 (P2), it displays the message "Signal received from P1(P2)"
    - If it receives 3 signals from the same process, it terminates processes P1 and P2 and it ends
- Process P1 and P2 run through an infinite cycle
  - They wait for a random time and then send a signal to the parent
    - P1 sends signals SIGUSR1
    - P2 sends signals SIGUSR2

Solution

```
#include <signal.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
```

```
int last_sig = -1;
int last_last_sig = -1;
int finish = 0;


void sign_handler(int sig){
  if (sig==SIGUSR1)
    printf("Signal received from P1\n");
  else if (sig==SIGUSR2)
    printf("Signal received from P2\n");
  if (sig == last_sig && last_sig == last_last_sig) {
    finish = 1;
  } else {
    last_last_sig = last_sig;
    last_sig = sig;
  }
}
```

Solution

```
int main() {
  int pid1, pid2;
  char cmd[100];

  if ( (signal(SIGUSR1, sign_handler) == SIG_ERR)
    || (signal(SIGUSR2, sign_handler) == SIG_ERR) ) {
    printf("Error initializing signal handler");
    exit(-1);
  }
  pid1 = fork();
  if (pid1==0) {
    while (1) {
      sleep( rand()%2 );
      kill(getppid(), SIGUSR1);
    }
  } else {
```

Child 1
P1

Solution

```
    pid2 = fork();
    if (pid2==0) {
      while (1) {
        sleep(rand()%3);
        kill(getppid(), SIGUSR2);
      }
    } else {
      while (1) {
        pause();
        if (finish) {
          sprintf(cmd, "kill -9 %d", pid1); system(cmd);
          sprintf(cmd, "kill -9 %d", pid2); system(cmd);
          exit(0);
        }
      }
    }
  return (0);
}
```

Child 2
P2

Parent
P

Kill children
before ending

Exercise 06 (pipes)

- A program runs three processes (P1, P2, and P3) connected to each other by three pipes (P1-P2, P2-P3, and P3-P1)
- The three processes work in a circular manner
  - Each process Pi (i.e. P1, P2, or P3) reads an integer from standard input
  - Transfer the integer to the next process (i.e. P2, P3, or P1, respectively)
  - The next process in the chain sleeps a number of seconds equal to that value and it repeats the same operation
- Assume that processes do not terminate

Solution

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

static void p1(int, int);
static void p2(int, int);
static void p3(int, int);
```

Solution

If the pipe is closed at the other end:
Read returns 0
Write returns SIGPIPE

```
int main () {
  int p12[2], p23[2], p31[2];
  setbuf (stdout, 0);
  pipe (p12); pipe (p23); pipe (p31);
  if (fork()!=0) {
    close(p12[0]); close(p23[0]); close(p23[1]); close(p31[1]);
    p1 (p31[0], p12[1]);
  } else {
    if (fork()!=0) {
      close(p12[1]); close(p23[0]); close(p31[0]); close(p31[1]);
      p2 (p12[0], p23[1]);
    } else {
      close(p12[0]); close(p12[1]); close(p23[1]); close(p31[0]);
      p3 (p23[0], p31[1]);
    }
  }
  return (0);
}
```

Solution

```
static void p1 (int pr, int pw) {
  int n;
  n = 0;
  while (1) {
    fprintf (stdout, "P1 waiting: %d secs\n", n);
    sleep (n);
    fprintf (stdout, "P1 reading: ");
    scanf ("%d", &n);
    write (pw, &n, sizeof (int));
    read (pr, &n, sizeof (int));
  }
  return;
}
```

One process must write first to start the pipeline

Solution

```
static void p2 (int pr, int pw) {
  int n;
  while (1) {
    read (pr, &n, sizeof (int));
    fprintf (stdout, "P2 waiting: %d secs\n", n);
    sleep (n);
    fprintf (stdout, "P2 reading: ");
    scanf ("%d", &n);
    write (pw, &n, sizeof (int));
  }
  return;
}
```

The other processes must read first

Solution

```
static void p3 (int pr, int pw) {
  int n;
  while (1) {
    read (pr, &n, sizeof (int));
    fprintf (stdout, "P3 waiting: %d secs\n", n);
    sleep (n);
    fprintf (stdout, "P3 reading: ");
    scanf ("%d", &n);
    write (pw, &n, sizeof (int));
  }
  return;
}
```

Equivalent to p2
(can share ther same function)

The other processes must read first

Exercise 07 (IO and pipes)

- A file stores an undefined number of strings, each one stored on a different file line
- Two processes P1 and P2 want to
  - Use the file to exchange strings between them
  - Synchronize their R/W operations on the file using one or more pipes
    - When P1 runs P2 waits and vice versa

Exercise 07 (IO and pipes)

- Each process
  - Reads the contents of the file and displays it on standard output
  - Store a new set of strings into the file
    - The new set of strings is read from the keyboard
    - The string end stop this phase
- The entire process must terminate when either P1 or P2 reads the string END

Solution

```
... includes ...
#define N 100
... prototypes ...

int main (int argc, char **argv) {
  int fd1[2], fd2[2];
  int childPid;
  pipe (fd1);
  pipe (fd2);
  childPid = fork();
  if (childPid == 0) {
    child (argv[1], fd1, fd2);
  } else {
    parent (argv[1], fd1, fd2);
  }
  wait ((void *)0);
  exit (1);
}
```

See source code (in u02s02e) for all implementation details

Solution

```
void  parent (char *name, int *fd1, int *fd2) {
  char c;
  int stop = 0;
  close (fd1[0]); close (fd2[1]);
  while (stop==0) {
    printf ("PARENT WAKE UP (PID=%d) \n", getpid());
    read_file (name);
    stop = write_file (name);
    c = (stop==0) ? '0' : '1';
    write (fd1[1], &c, sizeof (char));
    if (stop==1)
      break;
    read (fd2[0], &c, sizeof (char));
    stop = (c=='0') ? 0 : 1;
  }
  close (fd1[1]); close (fd2[0]);
  return;
}
```

Solution

```
void child (char *name, int *fd1, int *fd2) {
  char c;
  int stop = 0;
  close (fd1[1]); close (fd2[0]);
  while (stop==0) {
    read (fd1[0], &c, sizeof (char));
    printf ("CHILD WAKE UP (PID=%d) \n", getpid());
    if (c=='1')
      break;
    read_file (name);
    stop = write_file (name);
    c = (stop==0) ? '0' : '1';
    write (fd2[1], &c, sizeof (char));
  }
  close (fd1[0]); close (fd2[1]);
  return;
}
```

Solution

```
void read_file (char *name) {
  FILE *fp;
  char str[N];

  fp = fopen (name, "r");
  while (fscanf (fp, "%s", str) != EOF) {
    fprintf (stdout, "   %s\n", str);
  }
  fclose (fp);

  return;
}
```

Solution

```
int write_file (char *name) {
  FILE *fp;
  char str[N];
  fp = fopen (name, "w");
  do {
    fprintf (stdout, "   str: ");
    scanf ("%s", str);
    if (strcmp (str, "end") != 0 && strcmp (str, "END") != 0) {
      fprintf (fp, "   %s\n", str);
    }
  } while (strcmp (str, "end") != 0 && strcmp (str, "END") != 0);
  fclose (fp);
  if (strcmp (str, "END") == 0)
    return 1;
  else
    return 0;
}
```

Exercise 08 (thread analysis)

- Suppose that the following program is executed passing the value 5 in the command line
- Report the exact output it generates

See source code (in u02s02e) for all implementation details

passed here in *argv[]

```
... includes ...

... prototypes ...

int main (int argc, char *argv[]) {
  pthread_t thread;
  int n = atoi (argv[1]);
  setbuf (stdout, 0);
  pthread_create (&thread, NULL, T1, &n);
  pthread_join (thread, NULL);
  return 1;
}
```

we a thread t1 with n =5
and we just wait for the thread to

Here we wait for the thread to end

5-2=3
3+1 = 4
4-2=2
2+1=2
3-2=1
1+1 =2
2-2=0
start exiting
start exiting
and so on and so forth

Exercise

n = 5

```
void *T1 (void *p){
  pthread_t thread;
  int *pn = (int *) p;   Here we convert the null pointer to an integer
  int n = *pn;   We are getting the integer value which is 5
  if (n>0) {
    n-=2;
    printf ("(%d)", n);   here n =3
                                        n=3 passed to thread 2
    pthread_create (&thread, NULL, T2, &n);
  }  else{
  pthread_join (thread, NULL);
  pthread_exit (NULL);   }
}
```

n=3

```
void *T2 (void *p){
  pthread_t thread;
  int *pn = (int *) p;   same here we convert null pointer to int
  int n = *pn;   get the int value which is 3
  if (n>0) {
    n+=1;
    printf ("[%d]", n);   print 4 (n = 4)
                                        again we pass 4
    pthread_create (&thread, NULL, T1, &n);
  }
  pthread_join (thread, NULL);
  pthread_exit (NULL);
}
```

(3) (4) (2) ...... thihs happens until n is lesser than 0. And then it exits. pthread_join waits for it to en

Exercise

```
void *T1 (void *p){
  pthread_t thread;
  int *pn = (int *) p;
  int n = *pn;
  if (n>0) {
    n-=2;
    printf ("(%d)", n);
    pthread_create (&thread, NULL, T2, &n);
  }
  pthread_join (thread, NULL);
  pthread_exit (NULL);
}
```

```
void *T2 (void *p){
  pthread_t thread;
  int *pn = (int *) p;
  int n = *pn;
  if (n>0) {
    n+=1;
    printf ("[%d]", n);
    pthread_create (&thread, NULL, T1, &n);
  }
  pthread_join (thread, NULL);
  pthread_exit (NULL);
}
```

Output

```
(3)[4](2)[3](1)[2](0)
```

Exercise 09 (synchronization)

- A program executes a high number of threads T
- In order not to exceed the hardware resources, it forces the code section SC to be executed simultaneously by a maximum of N threads, with N << T
  - This behavior is achieved by defining a semaphore sem initialized to the value N, and then accessing SC only through the following code section

```
sem_wait (sem);
SC
sem_post (sem);
```

Exercise 09 (synchronization)

- Later on, the programmer wants to run some other threads that consume twice the hardware resources of the former threads
  - The programmer's idea is to manage the access to SC of these threads through the following piece of code

```
sem_wait (sem);
sem_wait (sem);
SC
sem_post (sem);
sem_post (sem);
```

- Is this correct?

Solution

- The solution is prone to deadlock
  - If N threads execute one sem_wait, before the second sem_wait, is executed by at least one of them, they would be blocked because no thread could enter the critical section (CS), and consequently execute the sem_post, which allows other threads to enter the critical section
- A possible solution is to make the couple of sem_wait "atomic"
  - We can use mutual exclusion on sem_wait
  - Pay attention to single sem_wait to

Solution

```
pthread_muted_t m;

...

pthread_mutex_init (&m, NULL);

...

ptrhead_mutex_lock (&m);
sem_wait (sem);
sem_wait (sem);
ptrhead_mutex_unlock (&m);
SC
sem_post (sem);
sem_post (sem);
```

Exercise 10 (synch - pseudo-code)

- In a system there are **two** threads
  - One of type A and one of type B
  - Thread A can call function **fA**
  - Thread B can call function **fB**
- The
  - First thread to call **fA** or **fB** is blocked
  - Second thread to call **fA** or **fB** is not blocked and it also unlocks the thread previously blocked
- Functions **fA** and **fB** can be reused by threads A and B an indefinite number of times

Exercise 10 (synch - pseudo-code)

- Using the pseudo-code primitives wait and signal write functions **fA** and **fB**
  - Use the minimum number of semaphores
  - Define all global variables and all semaphore initialization values
- Example
  - A calls the function **fA** and it blocks
  - B calls the function **fB** and it unlocks A
  - B calls the function **fB** and it blocks
  - A calls the function **fA** and unlocks B

Example Scenario:

Thread A Calls fA First:
Thread A calls function fA.
Since this is the first call to fA, thread A becomes blocked, waiting for thread B to call its function.
Thread B is free to execute its function.
Thread B Calls fB:
Thread B calls function fB.
Since thread A is blocked on fA, thread B executes fB without being blocked.
After finishing fB, thread B unblocks thread A, allowing it to proceed.
Thread A Calls fA Again:
Now that thread A is unblocked, it can call function fA again.
Thread A executes fA without being blocked.
After finishing fA, thread A unblocks thread B.
Thread B Calls fB Again:
Thread B, now unblocked, calls function fB again.
Thread B executes fB without being blocked.
After finishing fB, thread B unblocks thread A.

Solution

m mutext and s is semaphore

s is 0

```
init(m, 1);
init(s, 0);
int flag=0;
```

If flag==0 no one is waiting
flag==1 someone is waiting

```
fA() {
  wait(m);
  if (flag==0) {
    flag=1;
    signal(m);
    wait(s);
  } else {
    flag=0;
    signal(m);
    signal(s);
  }
}
```

Noone is waiting
Go to sleep

fB is waiting
(free it)

m is the mutual exclusion mutex

```
fB() {
  wait(m);
  if (flag==0) {
    flag=1;
    signal(m);
    wait(s);
  } else {
    flag=0;
    signal(m);
    signal(s);
  }
}
```

fA is waiting
(free it)

s is the waiting semaphore

Exercise 11 (synch - C)

- A program includes several threads and a shared variable that indicates one of two possible states (RED or GREEN)
- Each thread can call the
  - Function **change** to modify the state variable
    - If a thread calls change when the state is GREEN (RED) it changes the shared state variable to RED (GREEN)
  - Function **red** (**green**)
    - It blocks a thread when the state is GREEN (RED)
    - Does not block a thread when the state is RED (GREEN)

Exercise 11 (synch - C)

- Implement functions change, red, and green
  - Assume that the initial state is RED for all threads
- Example of execution
  - T1 calls red and it does not block
  - T2 calls red and it does not block
  - T3 calls green and it blocks
  - T4 calls green and it blocks
  - T1 calls red and it does not block
  - T5 calls change, the status changes to GREEN, and T3 and T4 are released (unblocked)
  - T5 calls green and it does not block

Solution

```
#define RED   0
#define GREEN 1
int state = RED;
sem_t s_red;
sem_t s_green;
sem_t m;
sem_init(&s_red, 0, 1);
sem_init(&s_green, 0, 0);
sem_init(&m, 0, 1);
red(){
  sem_wait(&s_red);
  sem_post(&s_red);
}
green(){
  sem_wait(&s_green);
  sem_post(&s_green);
}
```

```
change(){
  sem_wait(&m);
  if(state == RED){
    state = GREEN;
    sem_wait(&s_red);
    sem_post(&s_green);
  } else {
    state = RED;
    sem_wait(&s_green);
    sem_post(&s_red);
  }
  sem_post(&m);
}
```

Exercise 12 (synch)

- A program runs N threads of type A and N threads of type B
  - N is a positive integer passed on the command line to the program (e.g., 10)
  - All threads are identified by their category (i.e., A or B) and a creation index (i.e., 1, 2, ..., N)
- We want to coordinate the effort of all threads to always **run** a thread of the category A **before** one thread of the category B

Exercise 12 (synch)

- In the following there are **two** examples of correct executions with N=10

```
A3 B1 A1 B8 A2 B7 A8 B6 A9 B3 A7 B9 A6 B2 A5 B4 A4 B5 A10 B10
A10 B5 A6 B1 A7 B3 A1 B4 A3 B2 A2 B9 A8 B8 A5 B6 A4 B10 A9 B7
```

- Note that the order in which the threads of each category are executed is not fixed but depends on the threads scheduling
- Write the code using the C language
  - Realize the complete program, including the creation of the threads

Solution

```
... includes ...
... prototypes ...
sem s_a, s_b;

int main(int argc, char *argv[]){
  int N ;
  int i = 0;
  pthread_t *threads_A;
  pthread_t *threads_B;
  int *id;

  N = atoi(argv[1]);

  sem_init(&s_a, 0, 1);
  sem_init(&s_b, 0, 0);
  threads_A = (pthread_t*)malloc(N*sizeof(pthread_t));
  threads_B = (pthread_t*)malloc(N*sizeof(pthread_t));
  id = (int*)malloc(N*sizeof(int));
```

Semaphores initialization

Data structure allocation

OS IDs

User IDs

Solution

Thread generation

```
  for (i=0; i<N;i++){
    id[i] = i;
    pthread_create(&threads_A[i], NULL, TA, (void *)&(id[i]));
    pthread_create(&threads_B[i], NULL, TB, (void *)&(id[i]));
  }

  for (i=0; i<N;i++){
    pthread_join(threads_A[i], NULL);
    pthread_join(threads_B[i], NULL);
  }

  free(threads_A);
  free(threads_B);
  free(id);
  return 0;
}
```

Thread wait (join)

Data structure deallocation

Solution

Semaphores initialization (main) !

```
sem_init(&s_a, 0, 1);
sem_init(&s_b, 0, 0);
```

```
static void *TA(void *arg){
  int id;
  id = *((int*)arg);
  sem_wait(&s_a);
  printf("A%d ", id);
  sem_post(&s_b);
  pthread_exit(NULL);
}
```

```
static void *TB(void *arg){
  int id;
  id = *((int*)arg);
  sem_wait(&s_b);
  printf("B%d ", id);
  sem_post(&s_a);
  pthread_exit(NULL);
}
```

Exercise 13 (IO, thread, synch)

- A file, of undefined length and in ASCII format, contains a list of integer numbers
- Write a program that, after receiving a value **k** (integer) and a string from command line, generates **k** threads and wait them
- Each thread
  - Reads the file **concurrently**, and performs the **sum** of the integer numbers it reads from file
  - When the end of file is reached, it **prints** the number of integer numbers it has read and the computed sum
  - Terminates

Exercise 13 (IO, thread, synch)

- After all threads terminate, the main thread has to print the total number of integer numbers and the total sum
- Example

File format: file.txt

```
7
9
2
-4
15
0
3
```

Example of execution

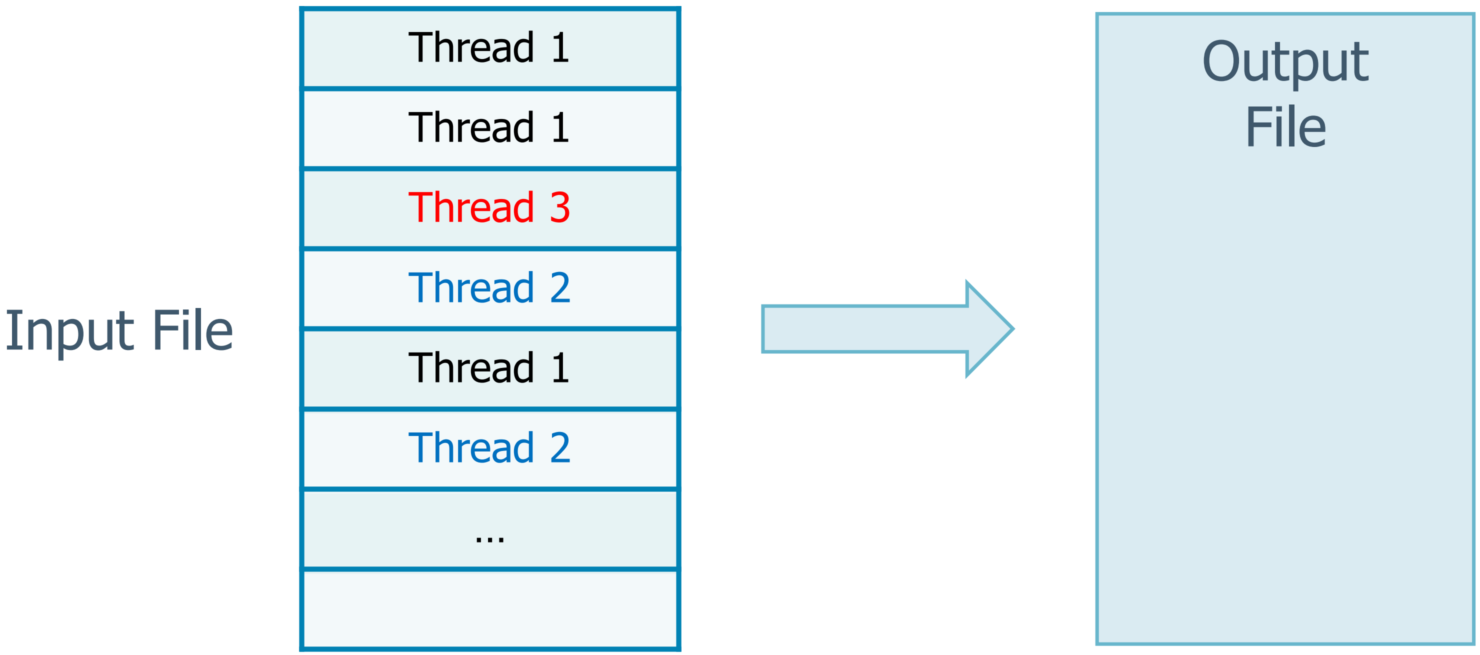
2 Threads

```
➢ pgrm 2 file.txt
Thread 1: Sum=18 #Line=3
Thread 2: Sum=14 #Line=4
Total   : Sum=32 #Line=7
```

Solution

The faster thread gets next record

- Solution 1
  - Let the threads run freely (**dynamic** partition)
    - Simple implementation
      - Just protect file I/O (read/write)
    - **High** thread contention
      - Quite slow in practice

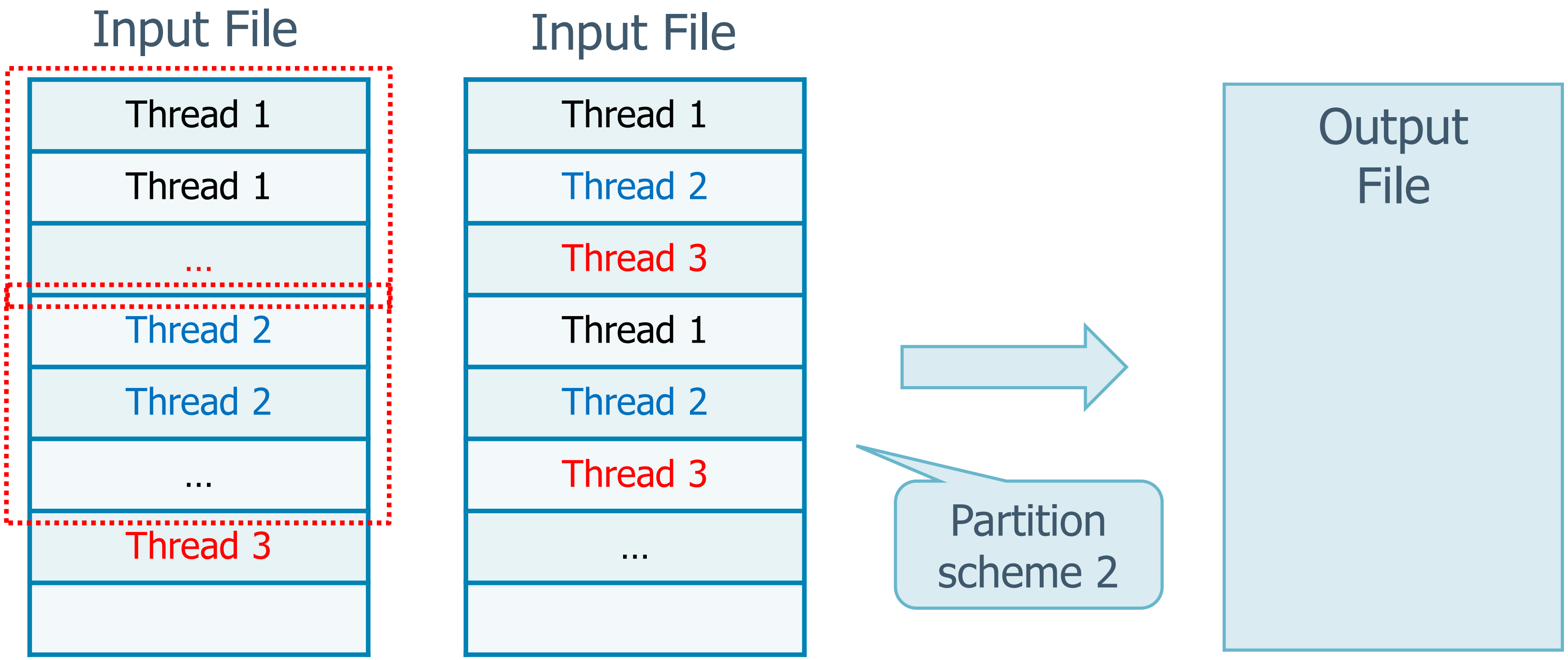


Solution

Each thread gets its own part of the file

- Solution 2
  - Assign to each thread 1/N of the file (**static** partition)
    - No contention
    - Workload: Efficiency is limited by the slower thread

Partition scheme 1

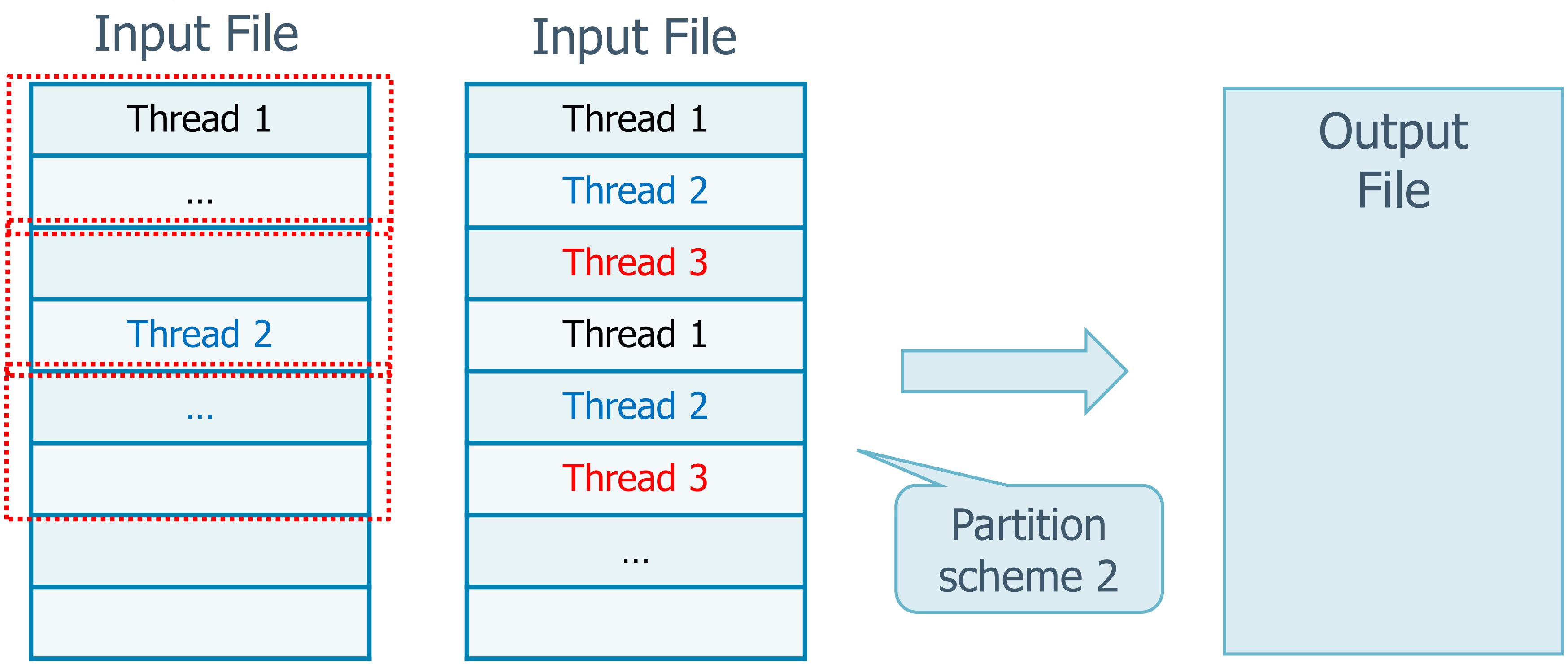


Solution

Each thread gets more sections of the file

- Solution 3
  - As solution 2 but create more partitions than threads as solution 1
    - Limited contention
    - Workload: Balanced through multiple partitions

Partition scheme 1



Solution

Implementation following solution 1

Includes, variables and prototypes

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <unistd.h>
#include <sys/types.h>
#include <semaphore.h>
#include <pthread.h>

#define L 100
struct threadData {
  pthread_t threadId;
  int id;
  FILE *fp;
  int line;
  int sum;
};
static void *readFile (void *);
sem_t sem;
```

It must be unique (i.e., it is global or it is passed as parameter to threads through this structure)

Solution

Main
Part 1

```
int main (int argc,char *argv[]) {
  int i, nT, total, line;
  struct threadData *td;
  void *retval;
  FILE *fp;

  nT = atoi (argv[1]);
  td = (struct threadData *) malloc
         (nT * sizeof (struct threadData));
  fp = fopen (argv[2], "r");
  if (td==NULL || fp==NULL) {
    fprintf (stderr, "Error ...\n");
    exit (1);
  }
  sem_init (&sem, 0, 1);
```

I/O Protection-Synch
A mutex is sufficient

Not shared

Init to 1

Solution

Main
Part 2

File pointer common to all the threads

```
  for (i=0; i<nT; i++) {
    td[i].id = i;
    td[i].fp = fp;
    td[i].line = td[i].sum = 0;
    pthread_create (&(td[i].threadId),
      NULL, readFile, (void *) &td[i]);
  }
  total = line = 0;
  for (i=0; i<nT; i++) {
    pthread_join (td[i].threadId, &retval);
    total += td[i].sum;
    line += td[i].line;
  }
  fprintf (stdout, "Total: Sum=%d #Line=%d\n",
    total, line);
  sem_destroy (&sem);
  fclose (fp);
  return (1);
}
```

Solution

Thread function

```
static void *readFile (void *arg){
  int n, retVal;
  struct threadData *td;
  td = (struct threadData *) arg;
  do {
    sem_wait (&sem);
    retVal = fscanf (td->fp, "%d", &n);
    sem_post (&sem);
    if (retVal!=EOF) {
      td->line++;
      td->sum += n;
    }
    sleep (1);  // Delay Threads
  } while (retVal!=EOF);
  fprintf (stdout, "Thread: %d Sum=%d #Line=%d\n",
    td->id, td->sum, td->line);
  pthread_exit ((void *) 1);
}
```

Mutual exclusion for file reading

Exercise 14 (barrier)

- A function f is formed by two distinct sessions A and B, executed only once in sequence

```
... f (...) {
  A
  B
}
```

- The function f is executed by N threads in parallel
  - Each thread can run f at most once
  - N is an integer value, defined but unknown

Exercise 14 (barrier)

- The user would like to synchronize the N threads such that section B is executed by a thread only after all threads have terminated the execution of section A
  - In other words, the N threads must synchronize after running on A and before one of them is running on B
- Write the require C code to implement such a behavior

```
... f (...) {
  A
  B
}
```

Barrier will be formally introduced in Unit 06, Section 07

Solution

```
#define N ...

sem_t m;
sem_t sync;
int nwait = 0;

sem_init(&m, 0, 1);
sem_init(&sync, 0, 0);
```

```
void f(void) {
  A();
  sem_wait(&m);
  nwait++;
  if(nwait==N){
    for(c=0; c<N; c++)
      sem_post (&sync);
  }
  sem_post (&m);
  sem_wait(&sync);
  B();
  sem_wait(&m);
  nwait--;
  sem_post(&m);
}
```

m is the mutual exclusion mutex

sync is the waiting semaphore