Process Management

Prof. Michele Loreti

Laboratorio di Sistemi Operativi
Corso di Laurea in Informatica (L31)
Scuola di Scienze e Tecnologie

Programs, Processes, and Threads

Program colloquially refers to executable files.

Programs, Processes, and Threads

Program colloquially refers to executable files.

A process is a running program.

Programs, Processes, and Threads

Program colloquially refers to executable files.

A process is a running program.

A thread is the unit of activity of a process.

Processes

Each process is identified by a Process ID (*pid*).

Processes

Each process is identified by a Process ID (*pid*).

PID is guaranteed to be unique at any single point in time.

Processes

Each process is identified by a Process ID (*pid*).

PID is guaranteed to be unique at any single point in time.

... pid can be re-used!

... from the point of view of a process, its pid never changes!

Processes

Each process is identified by a Process ID (*pid*).

PID is guaranteed to be unique at any single point in time.

. . . pid can be re-used!

. . . from the point of view of a process, its pid never changes!

Process ID Allocation

By default, Linux kernel imposes a maximum process ID value of 32768

. . . *pid*s are allocated sequentially;

Processes

Each process is identified by a Process ID (*pid*).

PID is guaranteed to be unique at any single point in time.

... pid can be re-used!

... from the point of view of a process, its pid never changes!

Process ID Allocation

By default, Linux kernel imposes a maximum process ID value of 32768

... *pid*s are allocated sequentially;

... a new *pid* is allocated until the max id has been allocated.

Process Hierarchy

The process that spawns a new process is known as the parent; the new process is known as the child.

Process Hierarchy

The process that spawns a new process is known as the parent; the new process is known as the child.

Every process is spawned from another process (except the init process that is the first executed!).

Process Hierarchy

The process that spawns a new process is known as the parent; the new process is known as the child.

Every process is spawned from another process (except the init process that is the first executed!).

This relationship is recorded in each process's parent process ID (*ppid*), which is the pid of the child's parent.

Parent and Process ID...

The process ID is represented by the pid_t type, which is defined in the header file <sys/types.h>.

Parent and Process ID...

The process ID is represented by the pid_t type, which is defined in the header file <sys/types.h>.

Function getpid can be used to retrieve the process id:

```
#include <sys/types.h>
#include <unistd.h>

pid_t getpid(void);
```

Parent and Process ID...

The process ID is represented by the pid_t type, which is defined in the header file <sys/types.h>.

Function getpid can be used to retrieve the process id:

```
#include <sys/types.h>
#include <unistd.h>

pid_t getpid (void);
```

...while function getppid returns the *Parent Process ID*:

```
#include <sys/types.h>
#include <unistd.h>

pid_t getppid (void);
```

Executing a new process

Standard *C* library provides a set of functions that can be used to execute another process:

```
#include <unistd.h>
int execl (const char *path, const char *arg,
   ...);
```

Executing a new process

Standard *C* library provides a set of functions that can be used to execute another process:

```
#include <unistd.h>
int execl (const char *path, const char *arg,
   ...);
```

A call to execl() replaces the current process image with a new one by loading into memory the program pointed at by path.

Executing a new process

Standard *C* library provides a set of functions that can be used to execute another process:

```
#include <unistd.h>
int execl (const char *path, const char *arg,
   ...);
```

A call to execl() replaces the current process image with a new one by loading into memory the program pointed at by path.

```
int ret;
ret = execl ("/usr/bin/vi", "vi", NULL);
if (ret == -1)
   perror ("execl");
```

Fork...

A new process running the same image as the current one can be created via the fork() system call:

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork (void);
```

Fork...

A new process running the same image as the current one can be created via the fork() system call:

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork (void);
```

A successful call to fork() creates a new process, identical in almost all aspects to the invoking process.

Fork...

A new process running the same image as the current one can be created via the fork() system call:

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork (void);
```

A successful call to fork() creates a new process, identical in almost all aspects to the invoking process.

Both processes continue to run, returning from fork() as if nothing special had happened.

Fork...

A new process running the same image as the current one can be created via the fork() system call:

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork (void);
```

A successful call to fork() creates a new process, identical in almost all aspects to the invoking process.

Both processes continue to run, returning from fork() as if nothing special had happened.

In the child, a successful invocation of fork() returns 0. In the parent, fork() returns the *pid* of the child.

Copy-on-write. . .

In early Unix systems, upon invocation, the kernel created copies of all internal data structures, duplicated the process's page table entries, and then performed a page-by-page copy of the parent's address space into the child's new address space.

Copy-on-write. . .

In early Unix systems, upon invocation, the kernel created copies of all internal data structures, duplicated the process's page table entries, and then performed a page-by-page copy of the parent's address space into the child's new address space.

This is time consuming!

Copy-on-write...

In early Unix systems, upon invocation, the kernel created copies of all internal data structures, duplicated the process's page table entries, and then performed a page-by-page copy of the parent's address space into the child's new address space.

This is time consuming!

Modern Unix systems have superior behaviour. Instead of a wholesale copy of the parent's address space, modern Unix systems such as Linux employ copy-on-write (COW) pages.

Copy-on-write...

In early Unix systems, upon invocation, the kernel created copies of all internal data structures, duplicated the process's page table entries, and then performed a page-by-page copy of the parent's address space into the child's new address space.

This is time consuming!

Modern Unix systems have superior behaviour. Instead of a wholesale copy of the parent's address space, modern Unix systems such as Linux employ copy-on-write (COW) pages.

Copies are performed only on write!

Terminating a process

The standard function to terminate a process is:

```
#include <stdlib.h>

void exit (int status);
```

Terminating a process

The standard function to terminate a process is:

```c
#include <stdlib.h>

void exit (int status);
```

A call to exit() performs some basic shutdown steps, then instructs the kernel to terminate the process.

Terminating a process

The standard function to terminate a process is:

```
#include <stdlib.h>

void exit (int status);
```

A call to `exit()` performs some basic shutdown steps, then instructs the kernel to terminate the process.

The status parameter is used to denote the process's exit status. Other programs-as well as the user at the shell-can check this value.

Terminating a process

The standard function to terminate a process is:

```
#include <stdlib.h>

void exit (int status);
```

A call to exit() performs some basic shutdown steps, then instructs the kernel to terminate the process.

The status parameter is used to denote the process's exit status. Other programs-as well as the user at the shell-can check this value.

The parent receives: status & 0377

Process shutdown steps

Before terminating the process, the C library performs the following shutdown steps, in order:

1. Call any functions registered with atexit () or on_exit (), in the reverse order of their registration;
2. Flush all open standard I/O streams;
3. Remove any temporary files created with the tmpfile() function.

Process shutdown steps

Before terminating the process, the C library performs the following shutdown steps, in order:

1. Call any functions registered with atexit () or on_exit (), in the reverse order of their registration;
2. Flush all open standard I/O streams;
3. Remove any temporary files created with the tmpfile() function.

These steps finish all the work the process needs to do in user space, so exit () invokes the system call _exit () to let the kernel handle the rest of the termination process.

Handling termination

Linux implements, the `atexit()` library call, used to register functions to be invoked upon process termination:

Handling termination

Linux implements, the atexit() library call, used to register functions to be invoked upon process termination:

```
#include <stdlib.h>

int atexit (void (*function)(void));
```

Handling termination

Linux implements, the atexit () library call, used to register functions to be invoked upon process termination:

```
#include <stdlib.h>

int atexit (void (*function)(void));
```

A successful invocation of atexit () registers the given function to run during normal process termination, that is, when a process is terminated via either exit () or a return from main().

Handling termination

Linux also supports the `on_exit()` library call. This is an alternative to `atexit` defined in other standars:

Handling termination

Linux also supports the on_exit() library call. This is an alternative to atexit defined in other standars:

```
#include <stdlib.h>

int on_exit (void (*function)(int, void *), void *arg);
```

Handling termination

Linux also supports the on_exit() library call. This is an alternative to atexit defined in other standars:

```
#include <stdlib.h>

int on_exit (void (*function)(int, void *), void *arg);
```

This function works the same as atexit(), but the registered function's prototype is different:

Handling termination

Linux also supports the on_exit () library call. This is an alternative to atexit defined in other standars:

```
#include <stdlib.h>

int on_exit (void (*function)(int, void *), void *arg);
```

This function works the same as atexit (), but the registered function's prototype is different:

```
void my_function (int status, void *arg);
```

Waiting Child Termination...

When a child process terminates it is placed in a special state named zombie.

Waiting Child Termination...

When a child process terminates it is placed in a special state named zombie.

In this state only minimal info about the process are maintained, and it waits for its parent to inquire about its state.

Waiting Child Termination...

When a child process terminates it is placed in a special state named zombie.

In this state only minimal info about the process are maintained, and it waits for its parent to inquire about its state.

Only after the parent obtains the information preserved about the terminated child does the process formally exit and cease to exist even as a zombie.

Waiting Child Termination...

The Linux kernel provides several interfaces for obtaining information about terminated children.

Waiting Child Termination...

The Linux kernel provides several interfaces for obtaining information about terminated children.

The simplest is wait():

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

pid_t wait (int *status);
```

Waiting Child Termination...

The Linux kernel provides several interfaces for obtaining information about terminated children.

The simplest is wait():

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

pid_t wait (int *status);
```

A call to wait() returns the pid of a terminated child or -1 on error. If no child has terminated, the call blocks until a child terminates. If a child has already terminated, the call returns immediately.

Waiting Child Termination...

On error, there are two possible errno values:

- ECHILD: The calling process does not have any children.
- EINTR: A signal was received while waiting, and the call returned early.

Waiting Child Termination...

On error, there are two possible errno values:

- ECHILD: The calling process does not have any children.
- EINTR: A signal was received while waiting, and the call returned early.

The status pointer contains additional information about the child. A family of macros is provided for interpreting the parameter:

```
#include <sys/wait.h>
int WIFEXITED (status); //Process terminated normally.
int WIFSIGNALED (status); //Signal caused termination.
int WIFSTOPPED (status);  //Process stopped.
int WIFCONTINUED (status); //Process continued.
int WEXITSTATUS (status); //Low 8-bits of exit value.
int WTERMSIG (status); //Signal caused termination.
int WSTOPSIG (status);
int WCOREDUMP (status);
```

Waiting Child Termination...

```
int main (void) {
    int status; pid_t pid;
    if (!fork ()) return 1;
    pid = wait (&status);
    if (pid == -1) perror ("wait");
    printf ("pid=%d\n", pid);
    if (WIFEXITED (status))
        printf ("Normal termination with exit status=%d\n",
                WEXITSTATUS (status));
    if (WIFSIGNALED (status))
        printf ("Killed by signal=%d%s\n",
                WTERMSIG (status),
                WCOREDUMP (status) ? " (dumped core)" : "");
    if (WIFSTOPPED (status))
        printf ("Stopped by signal=%d\n",
                WSTOPSIG (status));
    if (WIFCONTINUED (status)) printf ("Continued\n");
    return 0;
}
```

Wait for a specific process. . .

Wait for a specific process...

If you know the pid of the process you want to wait for, you can use the waitpid() system call:

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

pid_t waitpid (pid_t pid, int *status, int options);
```

Wait for a specific process. . .

If you know the pid of the process you want to wait for, you can use the waitpid() system call:

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

pid_t waitpid (pid_t pid, int *status, int options);
```

The pid parameter specifies exactly which process or processes to wait for:

- pid<−1: Wait for any child process whose process group ID is equal to the absolute value of this value.
- pid=−1: Wait for any child process (same behaviour as wait())
- pid=0: Wait for any child process that belongs to the same process group as the calling process.
- pid>0: Wait for any child process whose pid is exactly the value provided.

Wait for a specific process. . .

Wait for a specific process. . .

If you know the pid of the process you want to wait for, you can use the waitpid() system call:

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

pid_t waitpid (pid_t pid, int *status, int options);
```

Wait for a specific process. . .

If you know the pid of the process you want to wait for, you can use the waitpid() system call:

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

pid_t waitpid (pid_t pid, int *status, int options);
```

The options parameter is a binary OR of zero or more of the following options:

- WNOHANG: Do not block, but return immediately if no matching child process has already terminated (or stopped or continued).
- WUNTRACED: the WIFSTOPPED bit in the returned status parameter is set, even if the calling process is not tracing the child process.
- WCONTINUED: the WIFCONTINUED bit in the returned status parameter is set even if the calling process is not tracing the child process.

Wait for a specific process...

Example:

```
int status; pid_t pid;
pid = waitpid (1742, &status, WNOHANG);

if (pid == -1)
  perror ("waitpid");
else {
  printf ("pid=%d\n", pid);
  if (WIFEXITED (status))
    printf ("Normal termination with exit status=%d\n",
        WEXITSTATUS (status));
  if (WIFSIGNALED (status))
    printf ("Killed by signal=%d%s\n",
        WTERMSIG (status),
        WCOREDUMP (status) ? " (dumped core)" : "");
}
```

More Waiting Versatility

XSI extension to POSIX defines, and Linux provides, waitid():

```
#include <sys/wait.h>

int waitid (idtype_t idtype,
            id_t id,
            siginfo_t *infop,
            int options);
```

More Waiting Versatility

XSI extension to POSIX defines, and Linux provides, waitid():

```
#include <sys/wait.h>

int waitid (idtype_t idtype,
            id_t id,
            siginfo_t *infop,
            int options);
```

The idtype and id arguments specify which children to wait for.

More Waiting Versatility

XSI extension to POSIX defines, and Linux provides, waitid():

```
#include <sys/wait.h>

int waitid (idtype_t idtype,
            id_t id,
            siginfo_t *infop,
            int options);
```

The idtype and id arguments specify which children to wait for.

idtype may be one of the following values:

- P_PID: Wait for a child whose pid matches id.
- P_GID: Wait for a child whose process group ID matches id.
- P_ALL: Wait for any child; id is ignored.

More Waiting Versatility

The options parameter is a binary OR of one or more of the following values:

- WEXITED: The call will wait for children that have terminated.
- WSTOPPED: The call will wait for children that have stopped execution in response to receipt of a signal.
- WCONTINUED: The call will wait for children that have continued execution in response to receipt of a signal.
- WNOHANG: The call will never block, but will return immediately if no matching child process has already terminated (or stopped, or continued).
- WNOWAIT: The call will not remove the matching process from the zombie state. The process may be waited upon in the future.

More Waiting Versatility

Upon successfully waiting for a child, waitid() fills in the infop parameter, which must point to a valid siginfo_t type.

More Waiting Versatility

Upon successfully waiting for a child, `waitid()` fills in the `infop` parameter, which must point to a valid `siginfo_t` type.

The exact layout of the `siginfo_t` structure is implementation-specific, but a handful of fields are valid after a call to `waitid()`.

More Waiting Versatility

Upon successfully waiting for a child, waitid() fills in the infop parameter, which must point to a valid siginfo_t type.

The exact layout of the siginfo_t structure is implementation-specific, but a handful of fields are valid after a call to waitid().

A successful invocation will ensure that the following fields are filled in:

- si_pid : The child's pid.
- si_uid : The child's uid.
- si_code : Set to one of CLD_EXITED, CLD_KILLED, CLD_STOPPED, or CLD_CONTINUED in response to the child terminating.
- si_signo : Set to SIGCHLD.
- si_status : If si_code is CLD_EXITED, this field is the exit code of the child process. Otherwise, this field is the number of the signal delivered to the child that caused the state change.

Launching and waiting new processes

Both ANSI C and POSIX define an interface that couples spawning a new process and waiting for its termination.

Launching and waiting new processes

Both ANSI C and POSIX define an interface that couples spawning a new process and waiting for its termination.

If a process is spawning a child only to immediately wait for its termination, it makes sense to use this interface:

```
#define _XOPEN_SOURCE /* if we want WEXITSTATUS, etc. */
#include <stdlib.h>


int system (const char *command);
```

The system() function is so named because the synchronous process invocation is called shelling out to the system.

Launching and waiting new processes

Both ANSI C and POSIX define an interface that couples spawning a new process and waiting for its termination.

If a process is spawning a child only to immediately wait for its termination, it makes sense to use this interface:

```
#define _XOPEN_SOURCE /* if we want WEXITSTATUS, etc. */
#include <stdlib.h>


int system (const char *command);
```

The system() function is so named because the synchronous process invocation is called shelling out to the system.

It is common to use system() to run a simple utility or shell script, often with the explicit goal of simply obtaining its return value.

To be continued...

Progetto Appelli Giugno/Luglio

Prof. Michele Loreti

Laboratorio di Sistemi Operativi
Corso di Laurea in Informatica (L31)
Scuola di Scienze e Tecnologie

Progetto Sessioni Giugno/Luglio…

Obiettivo: Sviluppare una applicazione di sistema `Unix/Linux` chiamata swordx che sia in grado di leggere un insieme di file (di testo) da una o più sorgenti e che produca in output un tile di testo contenente la lista delle parole che occorrono nei file letti con la relativa occorrenza.

Progetto Sessioni Giugno/Luglio...

Obiettivo: Sviluppare una applicazione di sistema `Unix/Linux` chiamata swordx che sia in grado di leggere un insieme di file (di testo) da una o più sorgenti e che produca in output un tile di testo contenente la lista delle parole che occorrono nei file letti con la relativa occorrenza.

Regole:

1. Il progetto dovr essere consegnato in un archivio .tgz contenente, oltre al codice, una relazione descrittiva del lavoro svolto;
2. Il progetto può essere svolto in gruppo (di al più tre persone);
3. La valutazione del progetto terrà conto di:
   - Corretto funzionamento;
   - Organizzazione del codice;
   - Conoscenza da parte di tutti i membri del gruppo.

Progetto Sessioni Giugno/Luglio...

Obiettivo: Sviluppare una applicazione di sistema `Unix/Linux` chiamata swordx che sia in grado di leggere un insieme di file (di testo) da una o più sorgenti e che produca in output un tile di testo contenente la lista delle parole che occorrono nei file letti con la relativa occorrenza.

Regole:

1. Il progetto dovr essere consegnato in un archivio .tgz contenente, oltre al codice, una relazione descrittiva del lavoro svolto;
2. Il progetto può essere svolto in gruppo (di al più tre persone);
3. La valutazione del progetto terrà conto di:
   - Corretto funzionamento;
   - Organizzazione del codice;
   - Conoscenza da parte di tutti i membri del gruppo.

Date di consegna: 18/06/2018, 02/07/2018, 16/07/2018.

To be continued...

Threads

Prof. Michele Loreti

Laboratorio di Sistemi Operativi
Corso di Laurea in Informatica (L31)
Scuola di Scienze e Tecnologie

Threading...

Threading is the creation and management of multiple units of execution within a single process.

Threading...

Threading is the creation and management of multiple units of execution within a single process.

Threading is a significant source of programming error, through the introduction of data races and deadlocks.

Threading. . .

Threading is the creation and management of multiple units of execution within a single process.

Threading is a significant source of programming error, through the introduction of data races and deadlocks.

The topic of threading can—and indeed does—fill whole books. Those works tend to focus on the myriad interfaces in a particular threading library.

Threading...

Threading is the creation and management of multiple units of execution within a single process.

Threading is a significant source of programming error, through the introduction of data races and deadlocks.

The topic of threading can—and indeed does—fill whole books. Those works tend to focus on the myriad interfaces in a particular threading library.

While we will focus on basics of the Linux threading API.

Binaries, Processes and Threads...

Binaries are dormant programs residing on a storage medium, compiled to a format accessible by a given operating system and machine architecture, ready to execute but not yet in motion.

Binaries, Processes and Threads…

Binaries are dormant programs residing on a storage medium, compiled to a format accessible by a given operating system and machine architecture, ready to execute but not yet in motion.

Processes are the operating system abstraction representing those binaries in action: the loaded binary, virtualised memory, kernel resources such as open files, an associated user, and so on.

Binaries, Processes and Threads. . .

Binaries are dormant programs residing on a storage medium, compiled to a format accessible by a given operating system and machine architecture, ready to execute but not yet in motion.

Processes are the operating system abstraction representing those binaries in action: the loaded binary, virtualised memory, kernel resources such as open files, an associated user, and so on.

Threads are the unit of execution within a process: a virtualised processor, a stack, and program state.

Binaries, Processes and Threads...

Binaries are dormant programs residing on a storage medium, compiled to a format accessible by a given operating system and machine architecture, ready to execute but not yet in motion.

Processes are the operating system abstraction representing those binaries in action: the loaded binary, virtualised memory, kernel resources such as open files, an associated user, and so on.

Threads are the unit of execution within a process: a virtualised processor, a stack, and program state.

Processes are running binaries and threads are the smallest unit of execution schedulable by an operating system's process scheduler.

Threading. . .

A process contains one or more threads.

Threading. . .

A process contains one or more threads.

If a process contains but one thread, there is only a single unit of execution in the process and only one thing going on at a time. We call such processes single threaded.

Threading...

A process contains one or more threads.

If a process contains but one thread, there is only a single unit of execution in the process and only one thing going on at a time. We call such processes single threaded.

If a process contains more than one thread, then there is more than one thing going on at once. We call such processes multithreaded.

Threading. . .

Modern operating systems provide two fundamental virtualised abstractions to user space:

- virtual memory
- and a virtualised processor.

Threading. . .

Modern operating systems provide two fundamental virtualised abstractions to user space:

- virtual memory
- and a virtualised processor.

Each running process has the illusion that it alone consumes the machine's resources!

Threading...

Modern operating systems provide two fundamental virtualised abstractions to user space:

- virtual memory
- and a virtualised processor.

Each running process has the illusion that it alone consumes the machine's resources!

Virtualised memory is associated with the process and not the thread. Thus, **each process has a unique view of memory that is shared by all threads in that process**.

Multithreading. . .

There are six primary benefits to multithreading:

Multithreading...

There are six primary benefits to multithreading:

1. Programming abstraction

Multithreading...

There are six primary benefits to multithreading:

1. Programming abstraction
2. Parallelism

Multithreading...

There are six primary benefits to multithreading:

1. Programming abstraction
2. Parallelism
3. Improving responsiveness

Multithreading...

There are six primary benefits to multithreading:

1. Programming abstraction
2. Parallelism
3. Improving responsiveness
4. Blocking I/O

Multithreading...

There are six primary benefits to multithreading:

1. Programming abstraction
2. Parallelism
3. Improving responsiveness
4. Blocking I/O
5. Context switching

Multithreading...

There are six primary benefits to multithreading:

1. Programming abstraction
2. Parallelism
3. Improving responsiveness
4. Blocking I/O
5. Context switching
6. Memory savings

Threading Models

Threading Models

Kernel-level threading: This is the simplest model, where kernel provides native support for threads, and each of those kernel threads translates directly to the user-space concept of a thread (1-process, 1-thread).

Threading Models

Kernel-level threading: This is the simplest model, where kernel provides native support for threads, and each of those kernel threads translates directly to the user-space concept of a thread (1-process, 1-thread).

User-level threading: in this model user space is the key to the system's threading support, as it implements the concept of a thread. A process with N threads will map to a single kernel process (1-process, N-threads).

Threading Models

Kernel-level threading: This is the simplest model, where kernel provides native support for threads, and each of those kernel threads translates directly to the user-space concept of a thread (1-process, 1-thread).

User-level threading: in this model user space is the key to the system's threading support, as it implements the concept of a thread. A process with N threads will map to a single kernel process (1-process, N-threads).

Hybrid Threading: A mix of Kernel-level and User-level (M-processes, N-threads).

Concurrency, Parallelism, and Races

Threads create two related but distinct phenomena: concurrency and parallelism.

Concurrency, Parallelism, and Races

Threads create two related but distinct phenomena: concurrency and parallelism.

Concurrency is the ability of two or more threads to execute in overlapping time periods.

Concurrency, Parallelism, and Races

Threads create two related but distinct phenomena: concurrency and parallelism.

Concurrency is the ability of two or more threads to execute in overlapping time periods.

Parallelism is the ability to execute two or more threads simultaneously.

Thread Race

Consider the following *C* function:

Thread Race

Consider the following *C* function:

```
int withdraw (struct account *account, int amount) {
  const int balance = account->balance;

  if (balance < amount)
    return -1;

  account->balance = balance - amount;

  disburse_money (amount);

  return 0;
}
```

Thread Race

Consider the following *C* function:

```
int withdraw (struct account *account, int amount) {
  const int balance = account->balance;

  if (balance < amount)
    return -1;

  account->balance = balance - amount;

  disburse_money (amount);

  return 0;
}
```

What can happen if two processes execute the code above at the concurrently?

Thread Race

Consider the following *C* instruction:

```
x++; //x is an integer.
```

Thread Race

Consider the following *C* instruction:

```
x++; //x is an integer.
```

Question: can the statement above be executed concurrently?

Thread Race

Consider the following *C* instruction:

```
x++; //x is an integer.
```

Question: can the statement above be executed concurrently?

C compiler transforms the code above in:

```
load x into register
add 1 to register
store register in x
```

Thread Race

We assume x=0

Case 1:

```
(Th1) load x into register
(Th1) add 1 to register
(Th1) store register in x
                        (Th2) load x into register
                        (Th2) add 1 to register
                        (Th2) store register in x
```

Thread Race

We assume x=0

Case 1:

```
(Th1) load x into register
(Th1) add 1 to register
(Th1) store register in x
                        (Th2) load x into register
                        (Th2) add 1 to register
                        (Th2) store register in x
```

Result: x=2

Thread Race

We assume x=0

Case 2:

```
                      (Th2) load x into register
                      (Th2) add 1 to register
                      (Th2) store register in x
(Th1) load x into register
(Th1) add 1 to register
(Th1) store register in x
```

Thread Race

We assume x=0

Case 2:

```
                        (Th2) load x into register
                        (Th2) add 1 to register
                        (Th2) store register in x
(Th1) load x into register
(Th1) add 1 to register
(Th1) store register in x
```

Result: x=2

Thread Race

We assume x=0

Case 3:

```
(Th1) load x into register
                        (Th2) load x into register
                        (Th2) add 1 to register
(Th1) add 1 to register
(Th1) store register in x
                        (Th2) store register in x
```

Thread Race

We assume x=0

Case 3:

```
(Th1) load x into register
                        (Th2) load x into register
                        (Th2) add 1 to register
(Th1) add 1 to register
(Th1) store register in x
                        (Th2) store register in x
```

Result: x=1

Thread Synchronisation

The fundamental source of races is that critical regions are a window during which correct program behaviour requires that threads do not interleave execution.

Thread Synchronisation

The fundamental source of races is that critical regions are a window during which correct program behaviour requires that threads do not interleave execution.

To prevent race conditions, then, the programmer needs to synchronise access to that window, ensuring mutually exclusive access to the critical region.

Thread Synchronisation

The fundamental source of races is that critical regions are a window during which correct program behaviour requires that threads do not interleave execution.

To prevent race conditions, then, the programmer needs to synchronise access to that window, ensuring mutually exclusive access to the critical region.

An operation (or set of operations) is atomic if it is indivisible, unable to be interleaved with other operations.

Thread Synchronisation

The fundamental source of races is that critical regions are a window during which correct program behaviour requires that threads do not interleave execution.

To prevent race conditions, then, the programmer needs to synchronise access to that window, ensuring mutually exclusive access to the critical region.

An operation (or set of operations) is atomic if it is indivisible, unable to be interleaved with other operations.

To the rest of the system, an atomic operation (or operations) appears to occur instantaneously. And that's the problem with critical regions: they are not indivisible, they don't occur instantaneously, they aren't atomic

To be continued...