Sistemi Operativi

Corso di Laurea in Informatica

a.a. 2020-2021

Gabriele Tolomei

Dipartimento di Informatica

Sapienza Università di Roma

tolomei@di.uniroma1.it

SAPIENZA UNIVERSITÀ DI ROMA

The Big Picture So Far

- We have presented a number of services the OS provides to
  - abstract from actual physical (HW) resources
  - ease the interaction between users and HW resources

The Big Picture So Far

- We have presented a number of services the OS provides to
  - abstract from actual physical (HW) resources
  - ease the interaction between users and HW resources
- Different OS designs depending on how those services are implemented
  - monolithic, layered, microkernel, hybrid, etc.

Process

- A process is the OS abstraction of a running program (unit of execution)

Process

- A process is the OS abstraction of a running program (unit of execution)
- Process is dynamic, whilst a program is static (code and data only)

Process

- A process is the OS abstraction of a running program (unit of execution)
- Process is dynamic, whilst a program is static (code and data only)
- Several processes may run the same program (e.g., multiple Google Chrome instances) but each has its own state

Process

- A process is the OS abstraction of a running program (unit of execution)
- Process is dynamic, whilst a program is static (code and data only)
- Several processes may run the same program (e.g., multiple Google Chrome instances) but each has its own state
- A process executes one instruction at a time, sequentially

OS Process Management

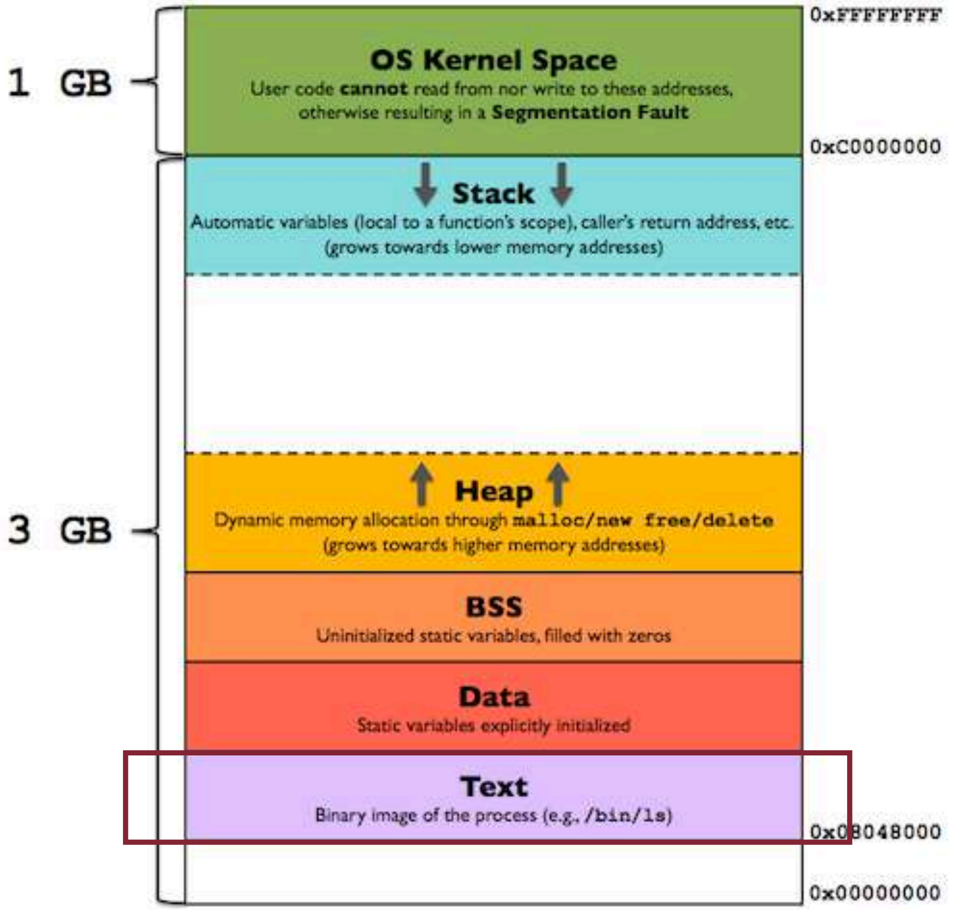
- How are processes represented in the OS?
- What are the possible states a process may be in and how the system moves from one state to another?
- How are processes created in the OS?
- How do processes communicate with each other?

Process State

- At least, process state consists of the following:
  - the code of the running program
  - the static data of the running program
  - the program counter (PC) indicating the next instruction to execute
  - CPU registers
  - the program's call chain (stack) along with frame and stack pointers
  - the space for dynamic memory allocation (heap) along with the heap pointer
  - the set of resources in use (e.g., open files)
  - the process execution state (ready, running, etc.)

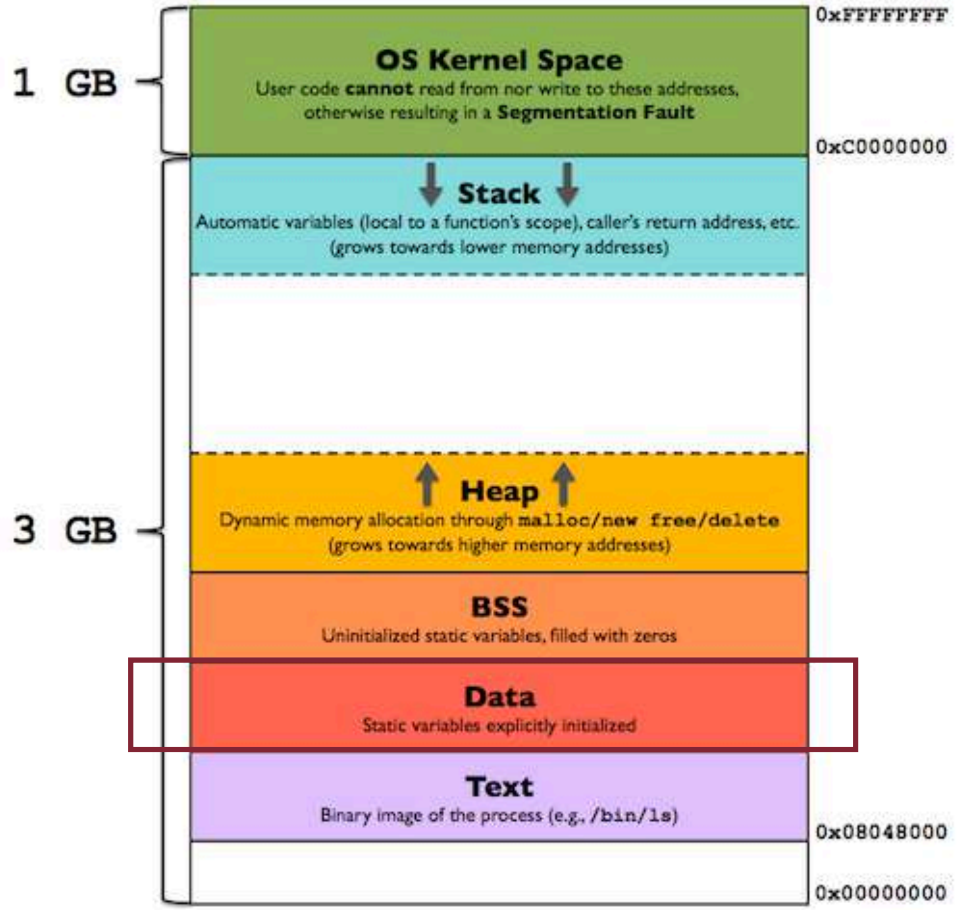
Process: In-Memory Layout

- Text → contains executable instructions



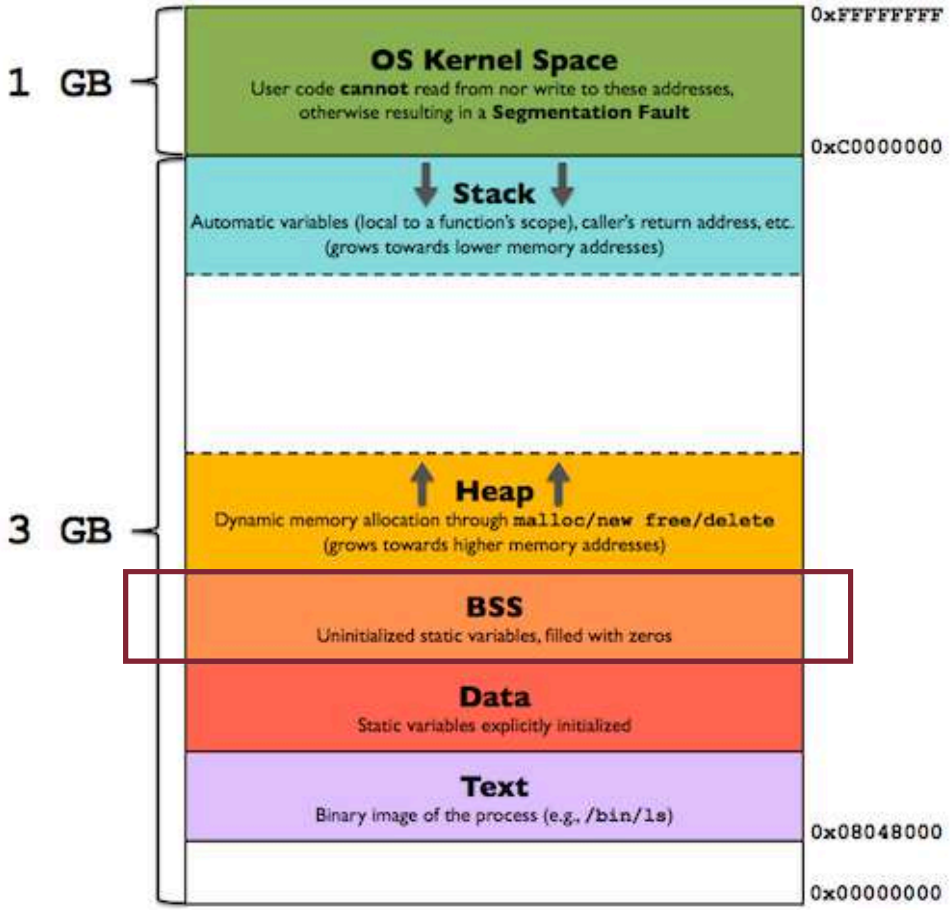
Process: In-Memory Layout

- Text → contains executable instructions
- Data → global and static variable (initialized)



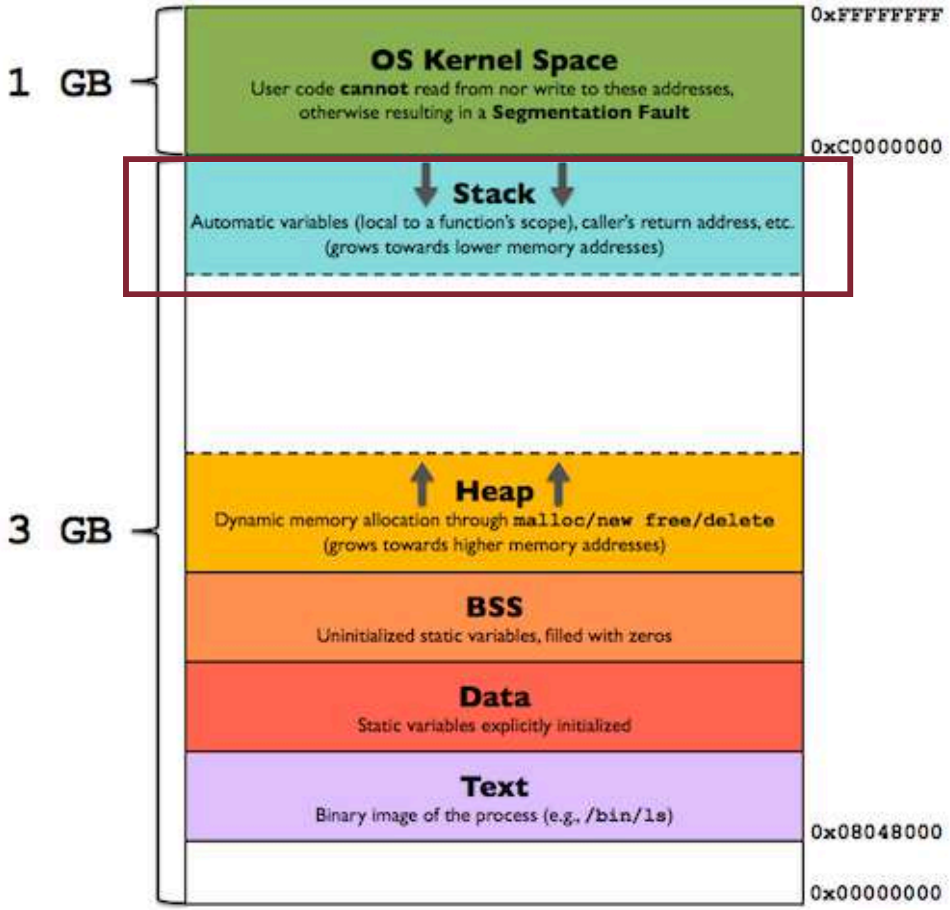
Process: In-Memory Layout

- Text → contains executable instructions
- Data → global and static variable (initialized)
- BSS → global and static variable (uninitialized or initialized to 0)



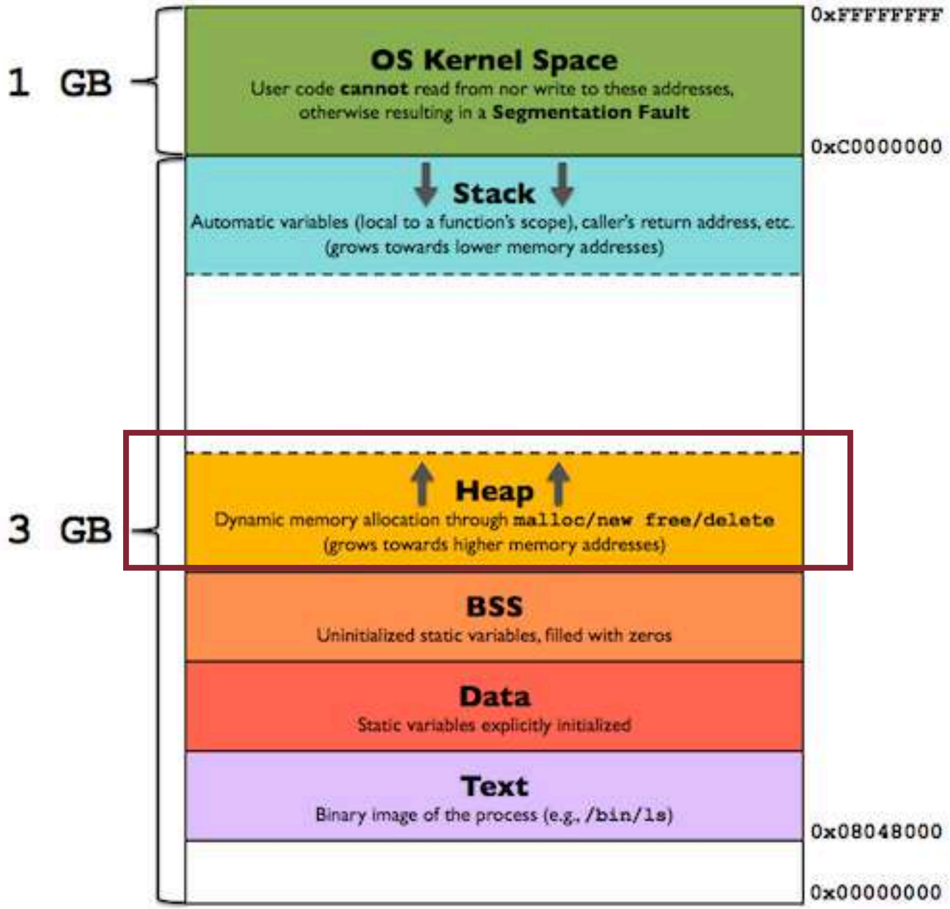
Process: In-Memory Layout

- Text → contains executable instructions
- Data → global and static variable (initialized)
- BSS → global and static variable (uninitialized or initialized to 0)
- Stack → LIFO structure used to store all the data needed by a function call (stack frame)



Process: In-Memory Layout

- Text → contains executable instructions
- Data → global and static variable (initialized)
- BSS → global and static variable (uninitialized or initialized to 0)
- Stack → LIFO structure used to store all the data needed by a function call (stack frame)
- Heap → used for dynamic allocation



Program vs. Process: Example

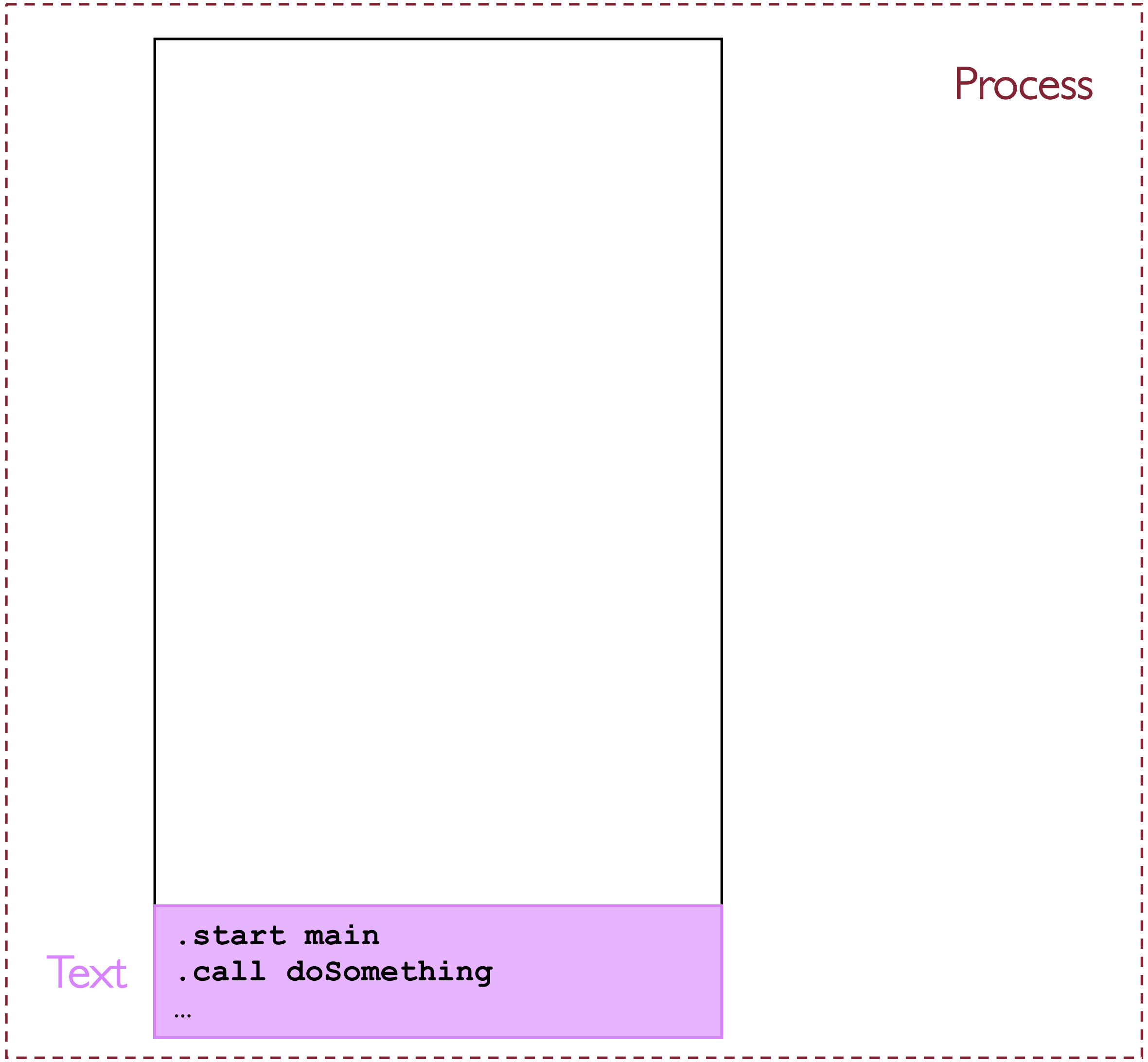
Program

```
int w = 42;
int x = 0;
float y;

void doSomething(int f) {
    int z = 37;
    z += f;
    …
}

int main() {
    char* c = malloc(128);
    int k = 12;
    doSomething(k);
    …
}
```

Program vs. Process: Example

Program

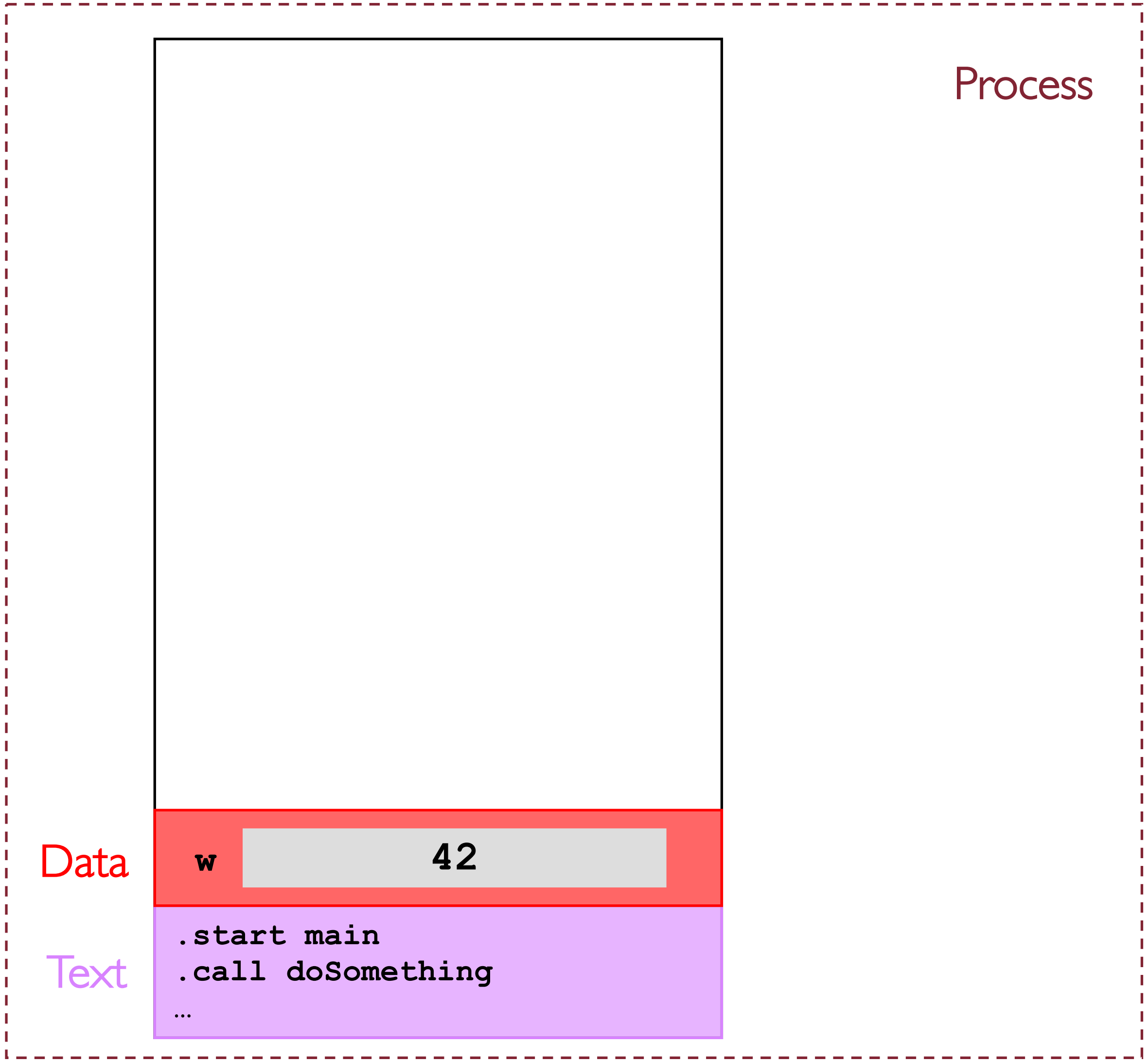
```c
int w = 42;
int x = 0;
float y;

void doSomething(int f) {
    int z = 37;
    z += f;
    …
}

int main() {
    char* c = malloc(128);
    int k = 12;
    doSomething(k);
    …
}
```



Program vs. Process: Example

Program

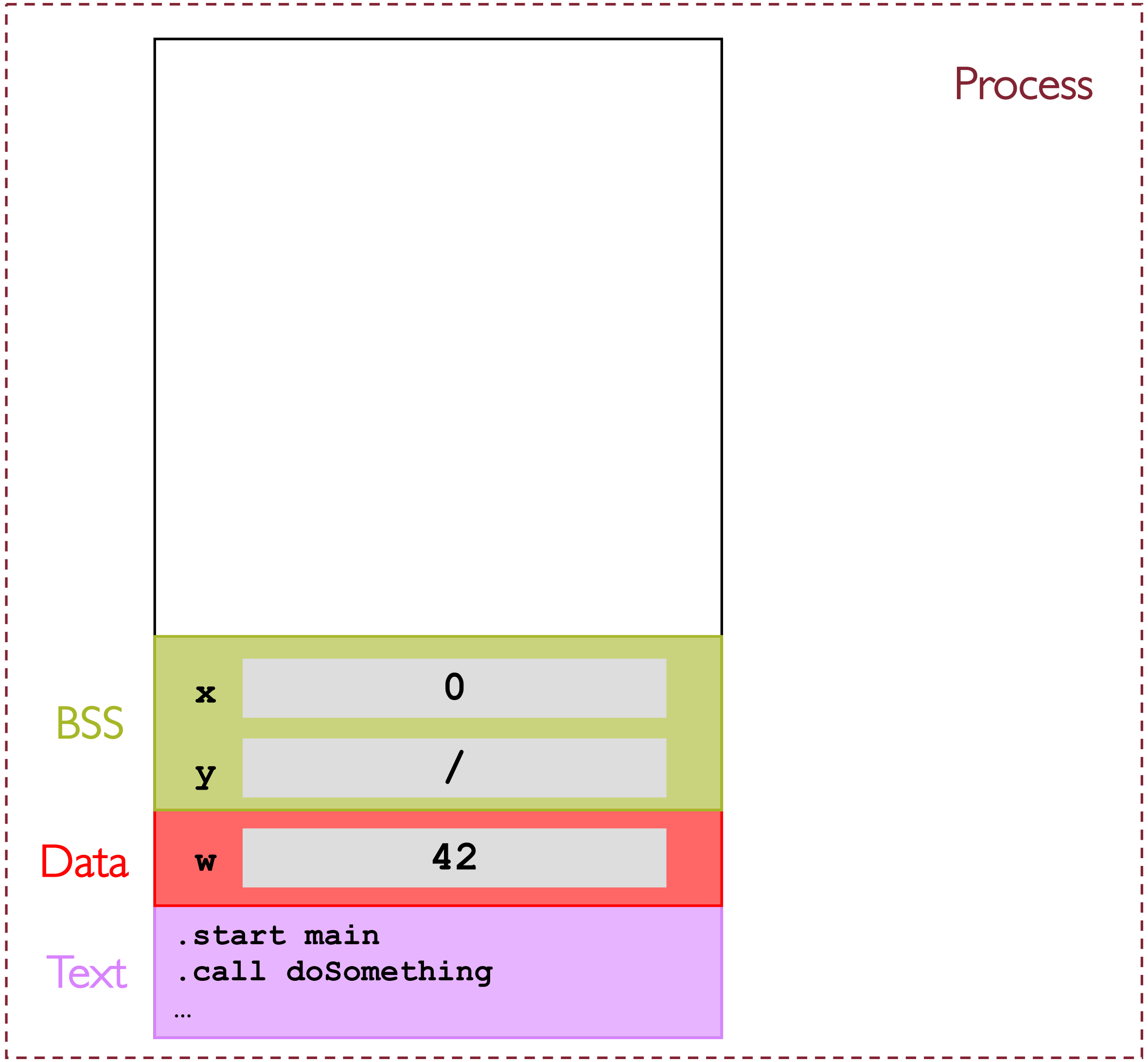
```
int w = 42;
int x = 0;
float y;

void doSomething(int f) {
    int z = 37;
    z += f;
    …
}

int main() {
    char* c = malloc(128);
    int k = 12;
    doSomething(k);
    …
}
```



Program vs. Process: Example

Program

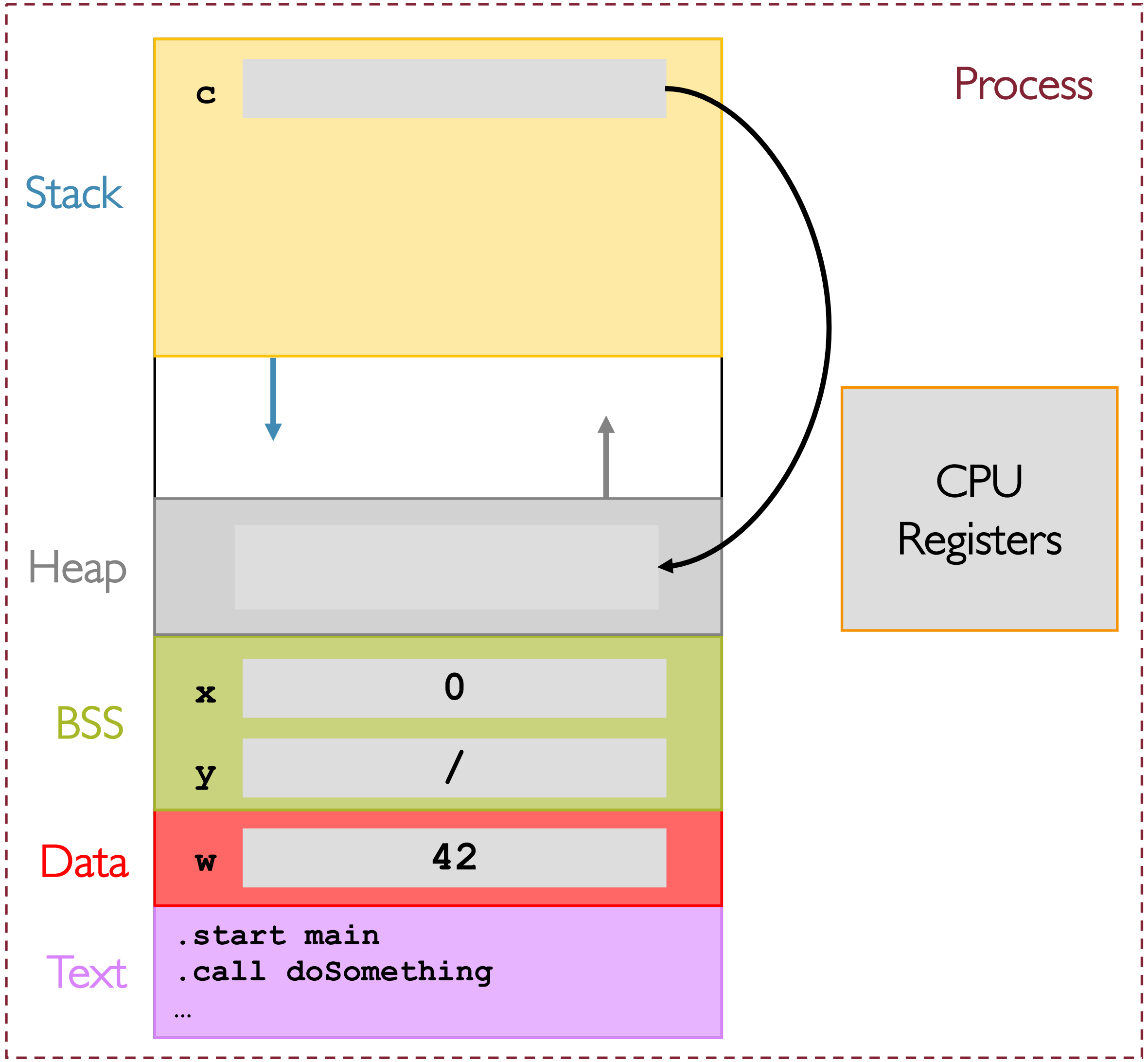
```
int w = 42;
int x = 0;
float y;

void doSomething(int f) {
     int z = 37;
     z += f;
     …
}

int main() {
     char* c = malloc(128);
     int k = 12;
     doSomething(k);
     …
}
```



Program vs. Process: Example

Program

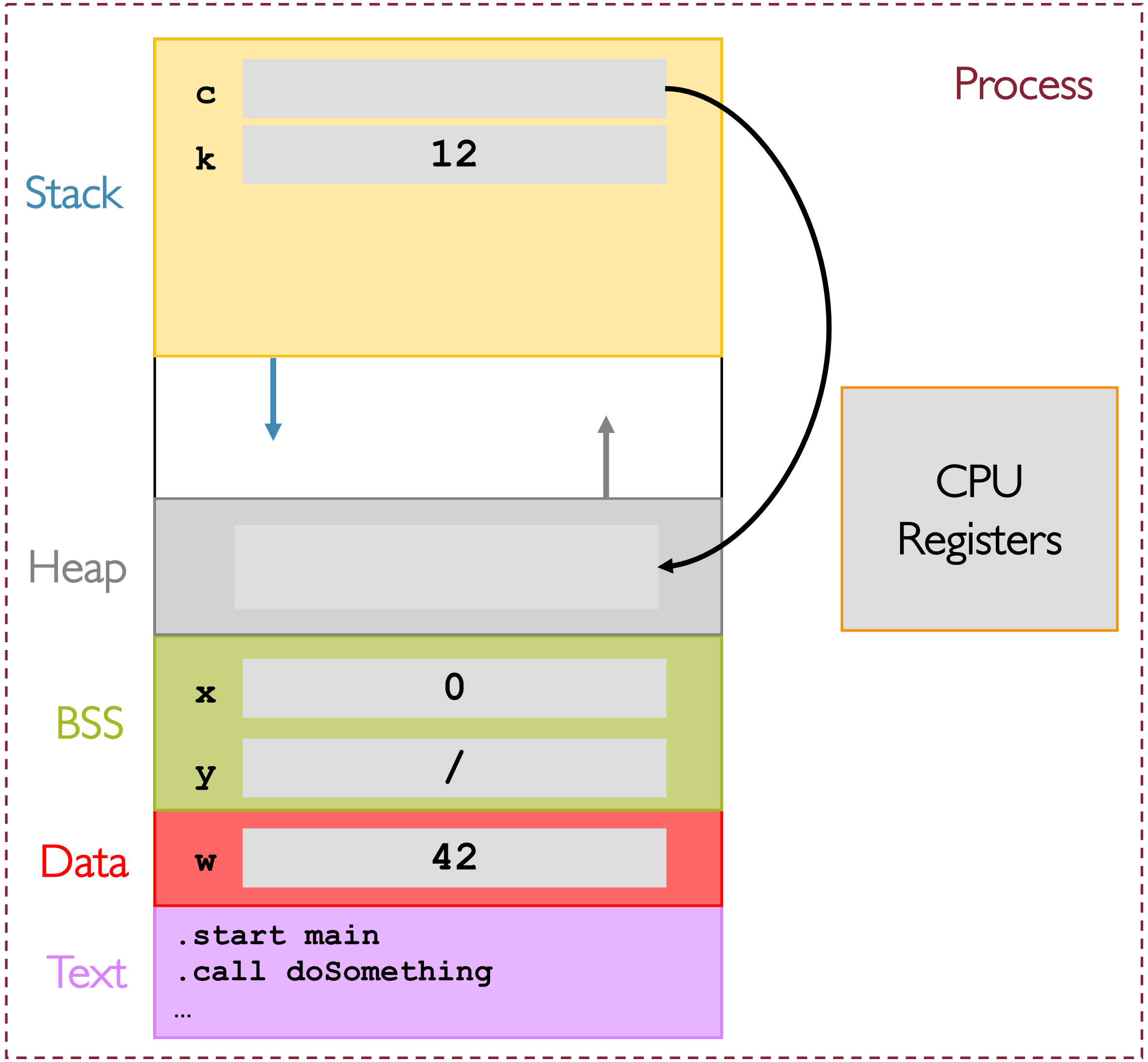
```
int w = 42;
int x = 0;
float y;

void doSomething(int f) {
    int z = 37;
    z += f;
    …
}

int main() {
    char* c = malloc(128);
    int k = 12;
    doSomething(k);
    …
}
```



Program vs. Process: Example

Program

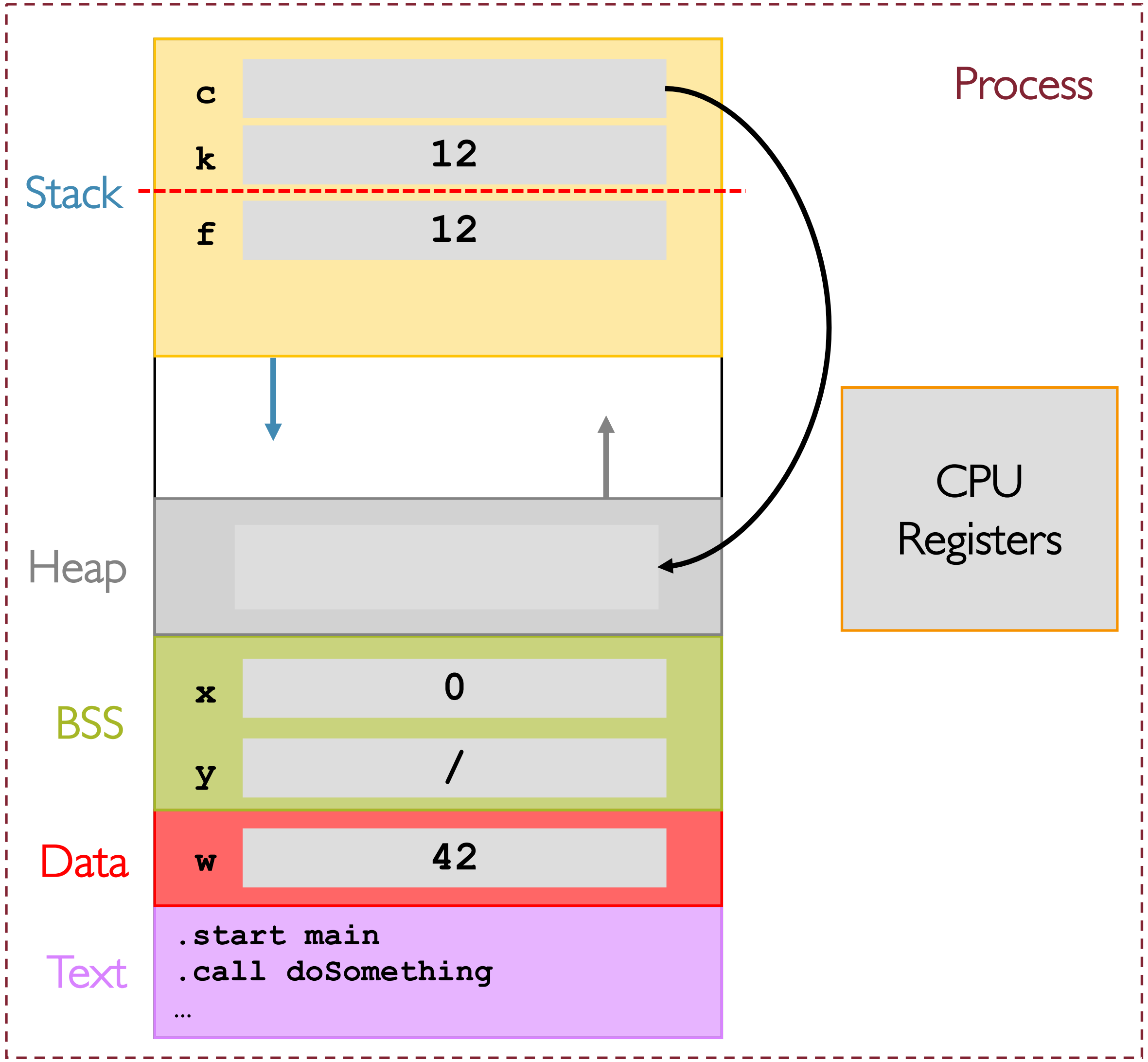
```
int w = 42;
int x = 0;
float y;

void doSomething(int f) {
    int z = 37;
    z += f;
    …
}

int main() {
    char* c = malloc(128);
    int k = 12;
    doSomething(k);
    …
}
```



Program vs. Process: Example

Program

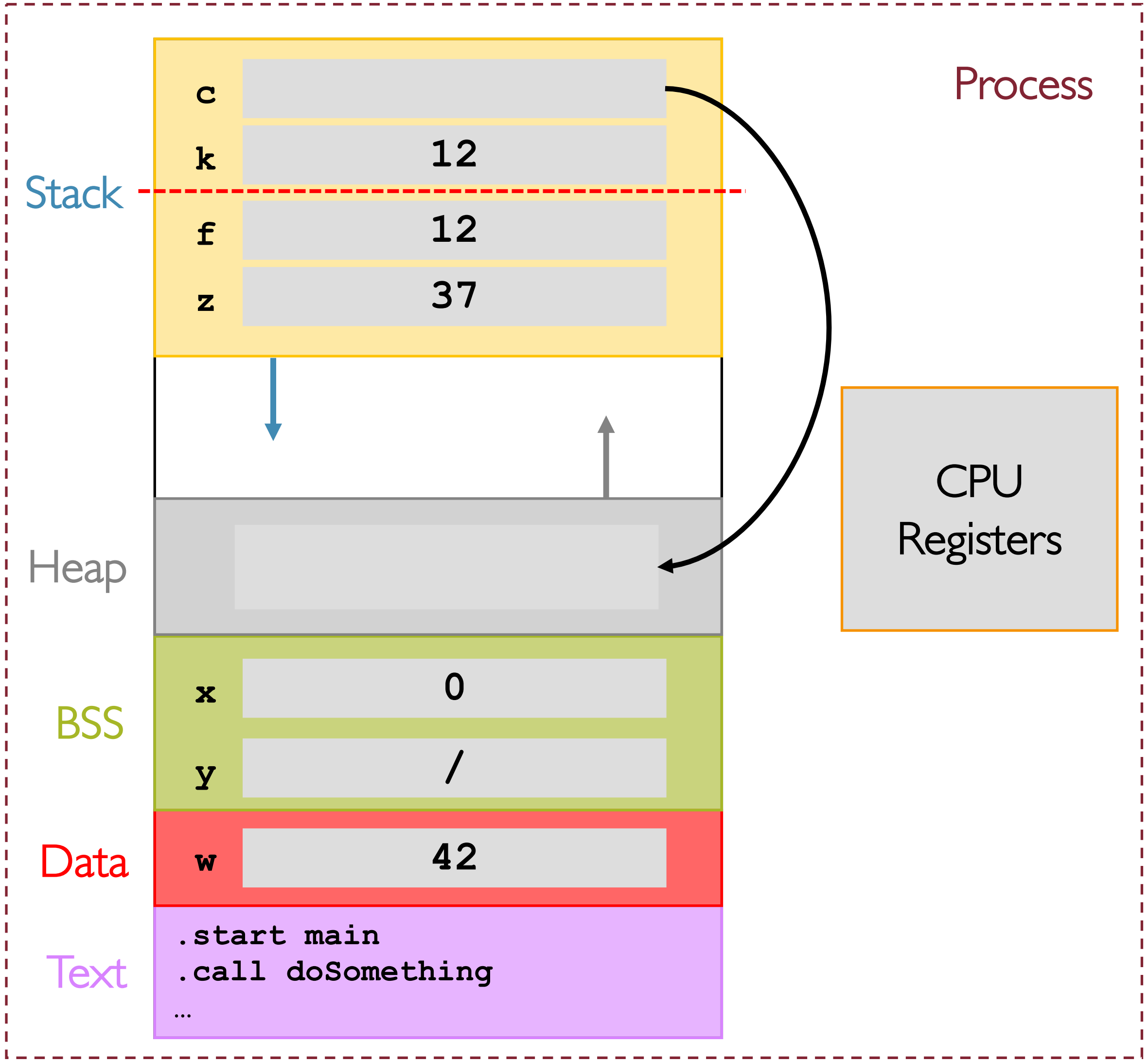
```
int w = 42;
int x = 0;
float y;

void doSomething(int f) {
    int z = 37;
    z += f;
    …
}

int main() {
    char* c = malloc(128);
    int k = 12;
    doSomething(k);
    …
}
```



Program vs. Process: Example

Program

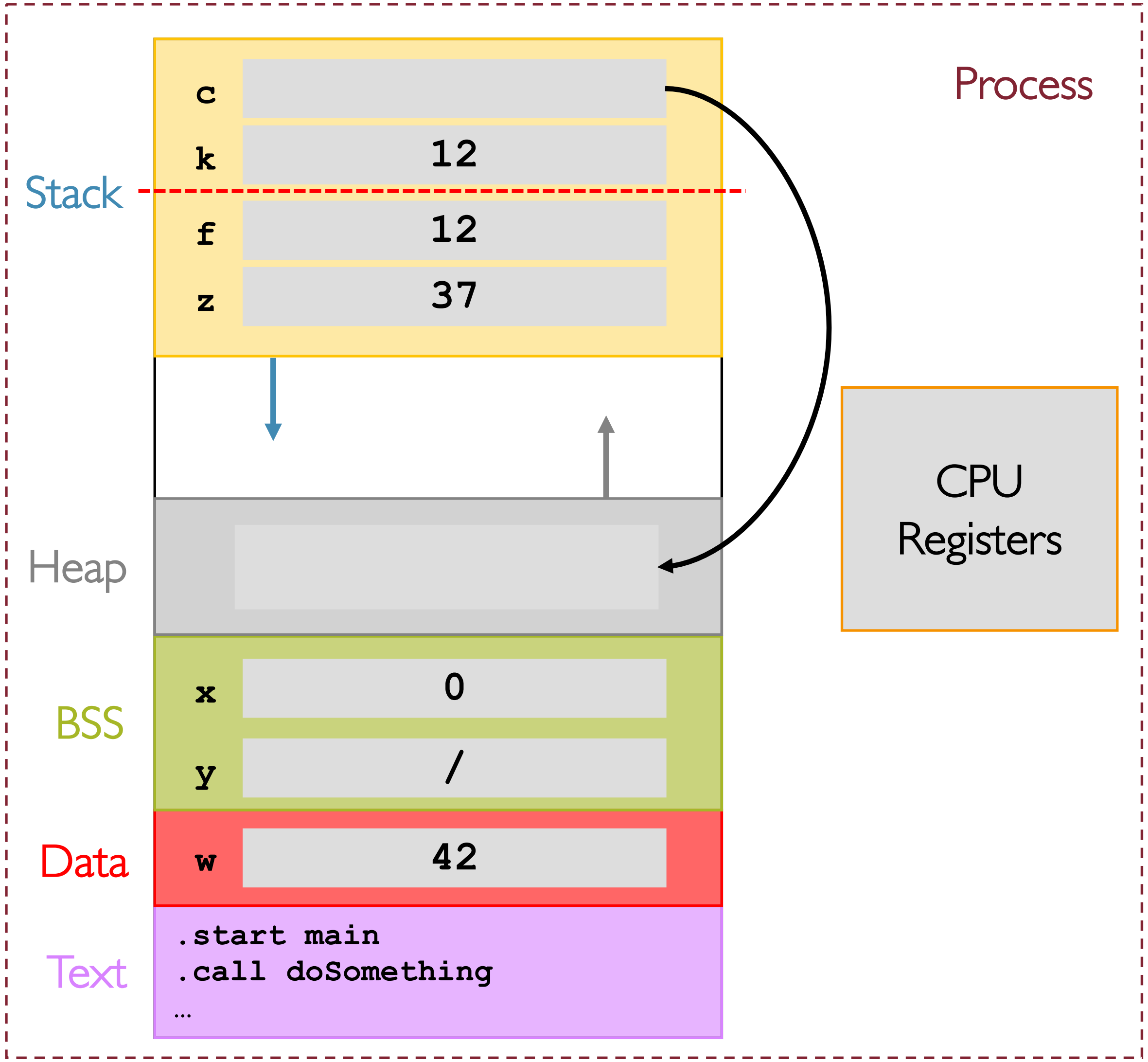
```
int w = 42;
int x = 0;
float y;

void doSomething(int f) {
    int z = 37;
    z += f;
    …
}

int main() {
    char* c = malloc(128);
    int k = 12;
    doSomething(k);
    …
}
```



Program vs. Process: Example

Program

```
int w = 42;
int x = 0;
float y;

void doSomething(int f) {
    int z = 37;
    z += f;
    …
}

int main() {
    char* c = malloc(128);
    int k = 12;
    doSomething(k);
    …
}
```



Process Execution State

- At each time a process can be in one of the following 5 states:

Process Execution State

- At each time a process can be in one of the following 5 states:
  - New → The OS has set up the process state

Process Execution State

- At each time a process can be in one of the following 5 states:
  - New → The OS has set up the process state
  - Ready → The process is ready to be executed yet waiting to be scheduled on to the CPU

Process Execution State

- At each time a process can be in one of the following 5 states:
  - New → The OS has set up the process state
  - Ready → The process is ready to be executed yet waiting to be scheduled on to the CPU
  - Running → The process is actually executing instructions on the CPU

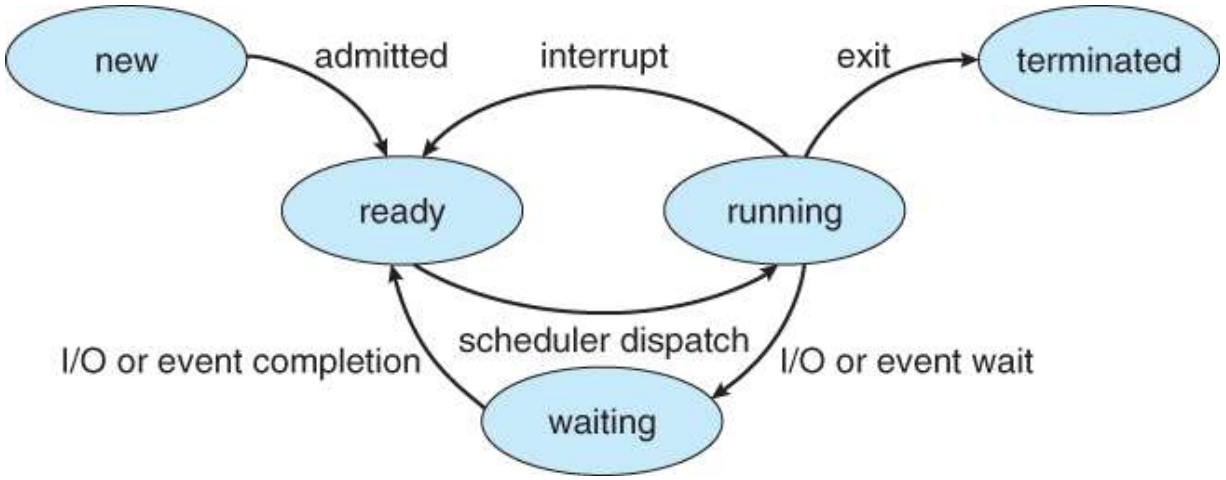
Process Execution State

- At each time a process can be in one of the following 5 states:
  - New → The OS has set up the process state
  - Ready → The process is ready to be executed yet waiting to be scheduled on to the CPU
  - Running → The process is actually executing instructions on the CPU
  - Waiting → The process is suspended waiting for a resource to be available or an event to complete/occur (e.g., keyboard input, disk access, timer, etc.)

Process Execution State

- At each time a process can be in one of the following **5 states**:
  - New → The OS has set up the process state
  - Ready → The process is ready to be executed yet waiting to be scheduled on to the CPU
  - Running → The process is actually executing instructions on the CPU
  - Waiting → The process is suspended waiting for a resource to be available or an event to complete/occur (e.g., keyboard input, disk access, timer, etc.)
  - Terminated → The process is finished and the OS can destroy it

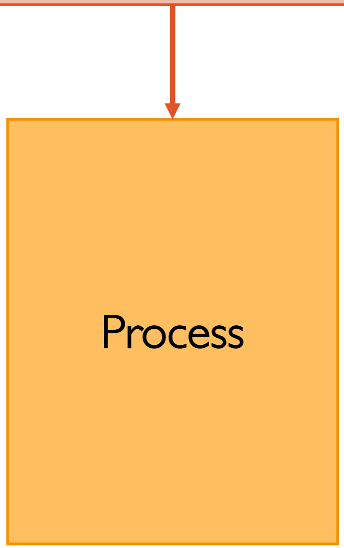
Process Execution State Diagram



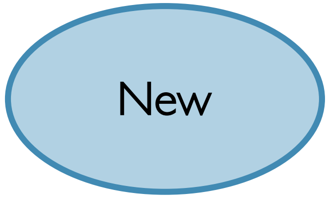
- As the process executes, it moves from state to state depending on:
  - program actions (e.g., system calls)
  - OS actions (e.g., scheduling)
  - external actions (e.g., interrupts)

Process Execution State: Example

```
int main() {
    …
    printf("Hello World!");
    …
}
```



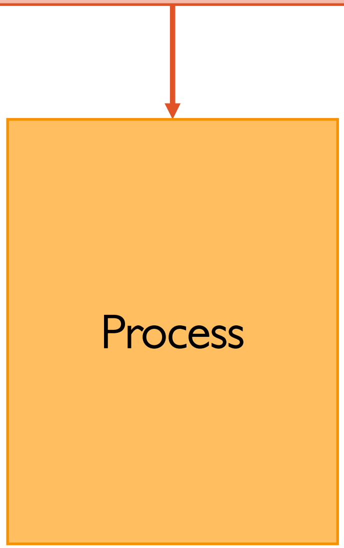
Main Memory



The process has been successfully set up

Process Execution State: Example

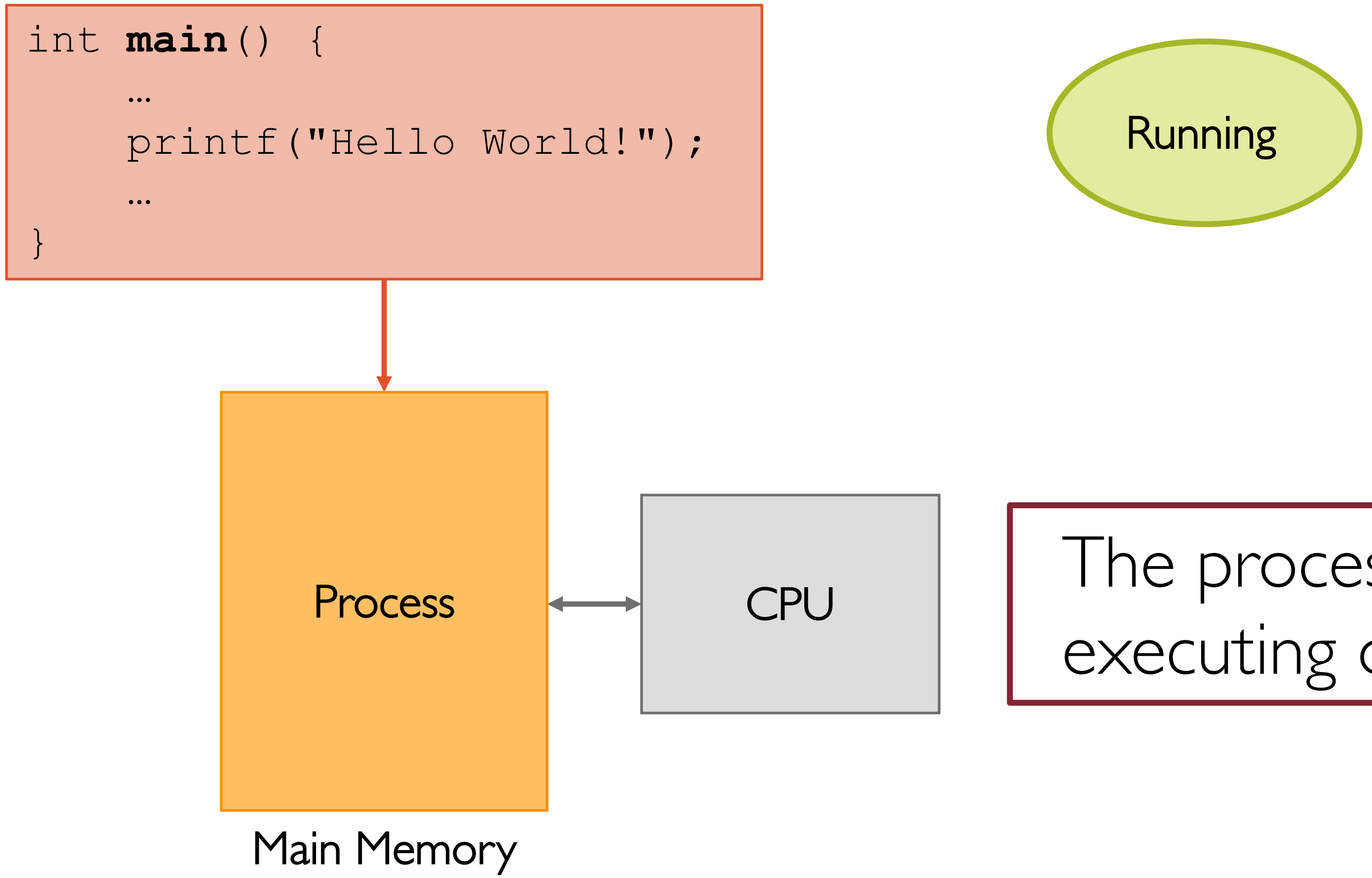
```c
int main() {
    …
    printf("Hello World!");
    …
}
```



Main Memory

The process is ready to be executed on the CPU

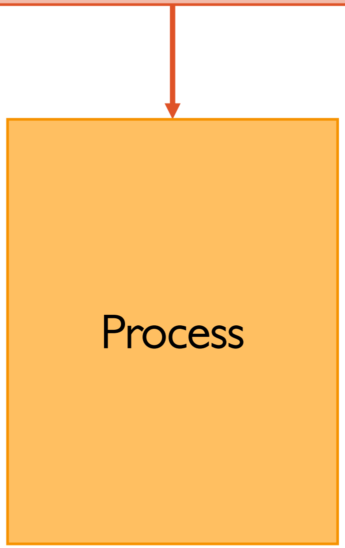
Process Execution State: Example



The process is actually executing on the CPU

Process Execution State: Example

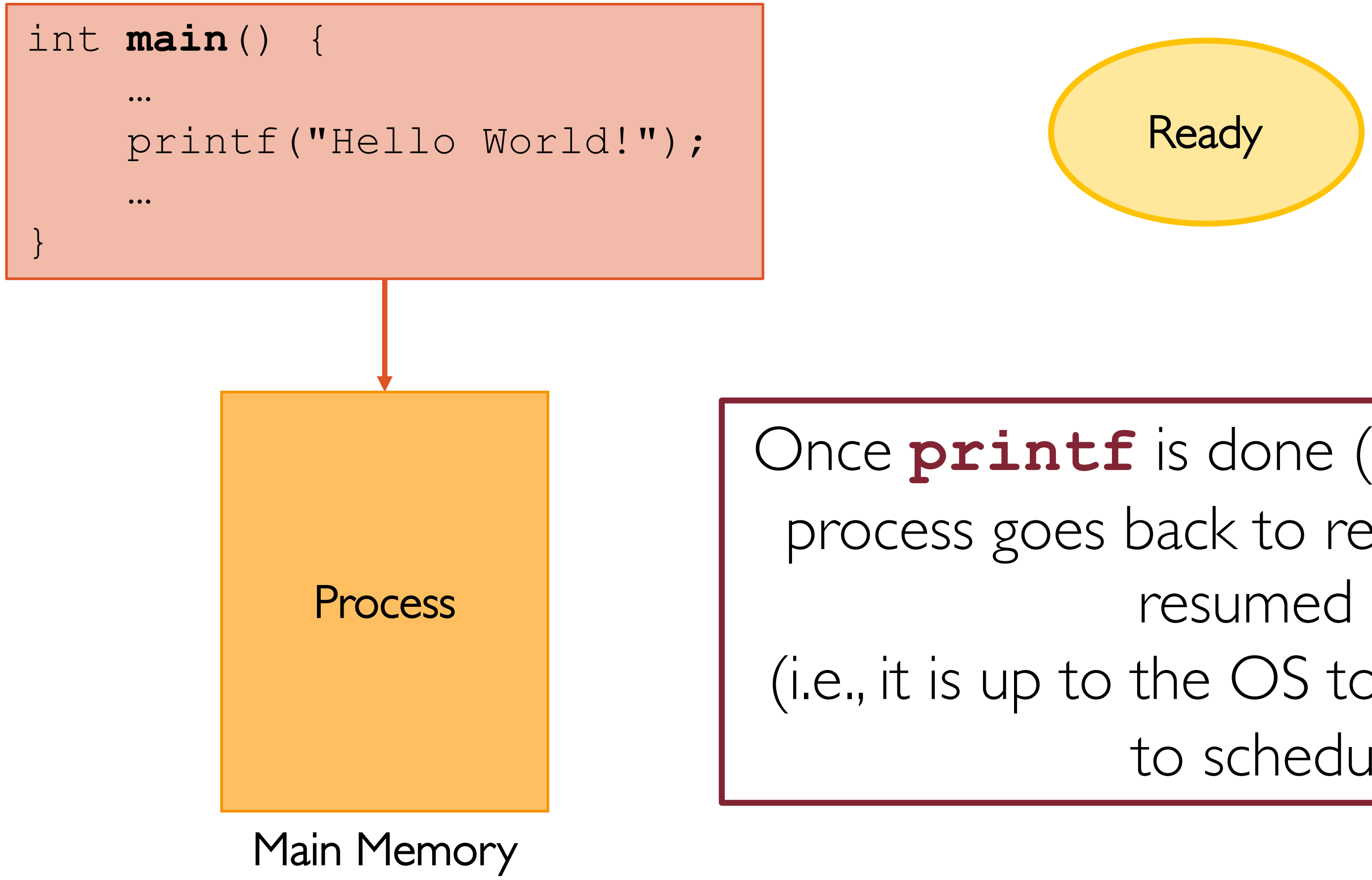
```
int main() {
    …
    printf("Hello World!");
    …
}
```



Main Memory

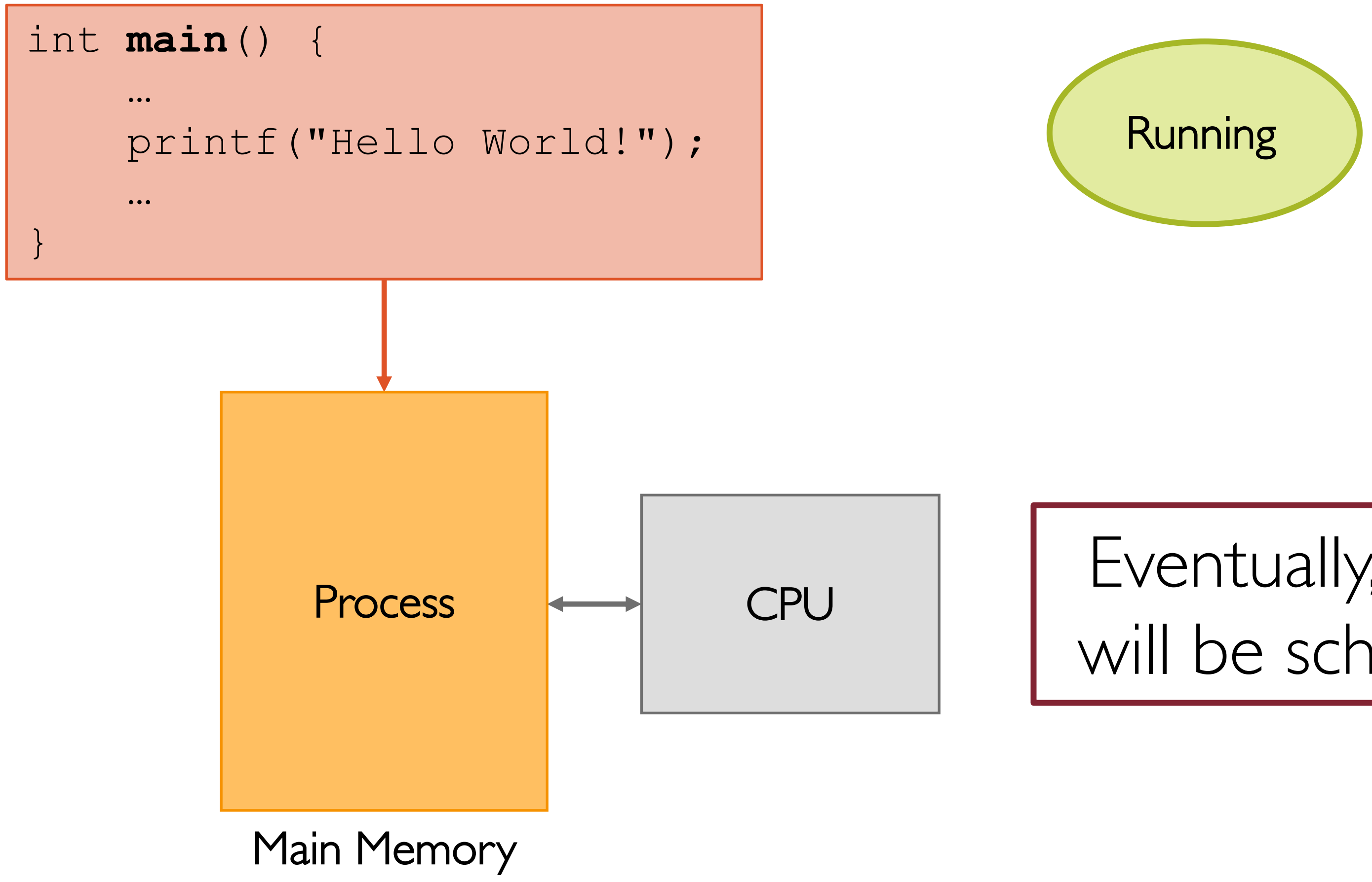
`printf` delegates off to a blocking I/O system call:
The current process is suspended in order for the OS to schedule another process which is ready to run

Process Execution State: Example



Once **`printf`** is done (e.g., HW interrupt) the process goes back to ready status yet it is not resumed directly!
(i.e., it is up to the OS to decide which process to schedule next)

Process Execution State: Example



Eventually, the process will be scheduled again

Process Execution State: Example

```
int main() {
    …
    printf("Hello World!");
    …
}
```

Terminated

Finally, the process terminates

Blocking vs. Non-Blocking Calls (Again)

- Most system calls (e.g., I/O ones) are blocking

Blocking vs. Non-Blocking Calls (Again)

- Most system calls (e.g., I/O ones) are blocking
  - the caller process (user space) can't do anything until the system call returns

Blocking vs. Non-Blocking Calls (Again)

- Most system calls (e.g., I/O ones) are blocking
  - the caller process (user space) can't do anything until the system call returns
  - the OS (kernel space):
    - sets the current process to a waiting state (i.e., waiting for the system call to return)
    - schedules a different ready process to avoid the CPU being idle

Blocking vs. Non-Blocking Calls (Again)

- Most system calls (e.g., I/O ones) are blocking
  - the caller process (user space) can't do anything until the system call returns
  - the OS (kernel space):
    - sets the current process to a waiting state (i.e., waiting for the system call to return)
    - schedules a different ready process to avoid the CPU being idle
  - once the system call returns the previously blocked process is ready to be scheduled for execution again

Blocking vs. Non-Blocking Calls (Again)

- Most system calls (e.g., I/O ones) are blocking
  - the caller process (user space) can't do anything until the system call returns
  - the OS (kernel space):
    - sets the current process to a waiting state (i.e., waiting for the system call to return)
    - schedules a different ready process to avoid the CPU being idle
  - once the system call returns the previously blocked process is ready to be scheduled for execution again
- NOTE: the whole system is not blocked, only the process which has requested the blocked call is!

Process Control Block (PCB)

- The main data structure used by the OS to keep track of any process
- The PCB keeps track of the execution state and location of a process
- The OS allocates a new PCB upon the creation of a process and places it into a state queue
- The OS deallocates a PCB as soon as the associated process terminates

Process Control Block (PCB)

- At least, the PCB contains the following:

Process Control Block (PCB)

- At least, the PCB contains the following:
  - Process state → ready, waiting, running, etc.

Process Control Block (PCB)

- At least, the PCB contains the following:
  - Process state → ready, waiting, running, etc.
  - Process number (i.e., unique identifier)

Process Control Block (PCB)

- At least, the PCB contains the following:
  - Process state → ready, waiting, running, etc.
  - Process number (i.e., unique identifier)
  - Program Counter (PC) + Stack Pointer (SP) + general purpose registers

Process Control Block (PCB)

- At least, the PCB contains the following:
  - Process state → ready, waiting, running, etc.
  - Process number (i.e., unique identifier)
  - Program Counter (PC) + Stack Pointer (SP) + general purpose registers
  - CPU scheduling information → priority and pointers to state queues

Process Control Block (PCB)

- At least, the PCB contains the following:
  - Process state → ready, waiting, running, etc.
  - Process number (i.e., unique identifier)
  - Program Counter (PC) + Stack Pointer (SP) + general purpose registers
  - CPU scheduling information → priority and pointers to state queues
  - Memory management information → page tables

Process Control Block (PCB)

- At least, the PCB contains the following:
  - Process state → ready, waiting, running, etc.
  - Process number (i.e., unique identifier)
  - Program Counter (PC) + Stack Pointer (SP) + general purpose registers
  - CPU scheduling information → priority and pointers to state queues
  - Memory management information → page tables
  - Accounting information → user and kernel CPU time consumed, owner

Process Control Block (PCB)

- At least, the PCB contains the following:
  - Process state → ready, waiting, running, etc.
  - Process number (i.e., unique identifier)
  - Program Counter (PC) + Stack Pointer (SP) + general purpose registers
  - CPU scheduling information → priority and pointers to state queues
  - Memory management information → page tables
  - Accounting information → user and kernel CPU time consumed, owner
  - I/O status → list of open files

Process Control Block (PCB)

Process Scheduling

- 2 main goals of the process scheduling system:
  - keep the CPU busy at all times
  - deliver "acceptable" response times for all programs, particularly for interactive ones

Process Scheduling

- 2 main goals of the process scheduling system:
  - keep the CPU busy at all times
  - deliver "acceptable" response times for all programs, particularly for interactive ones
- The process scheduler must meet these objectives by implementing suitable policies for swapping processes in and out of the CPU

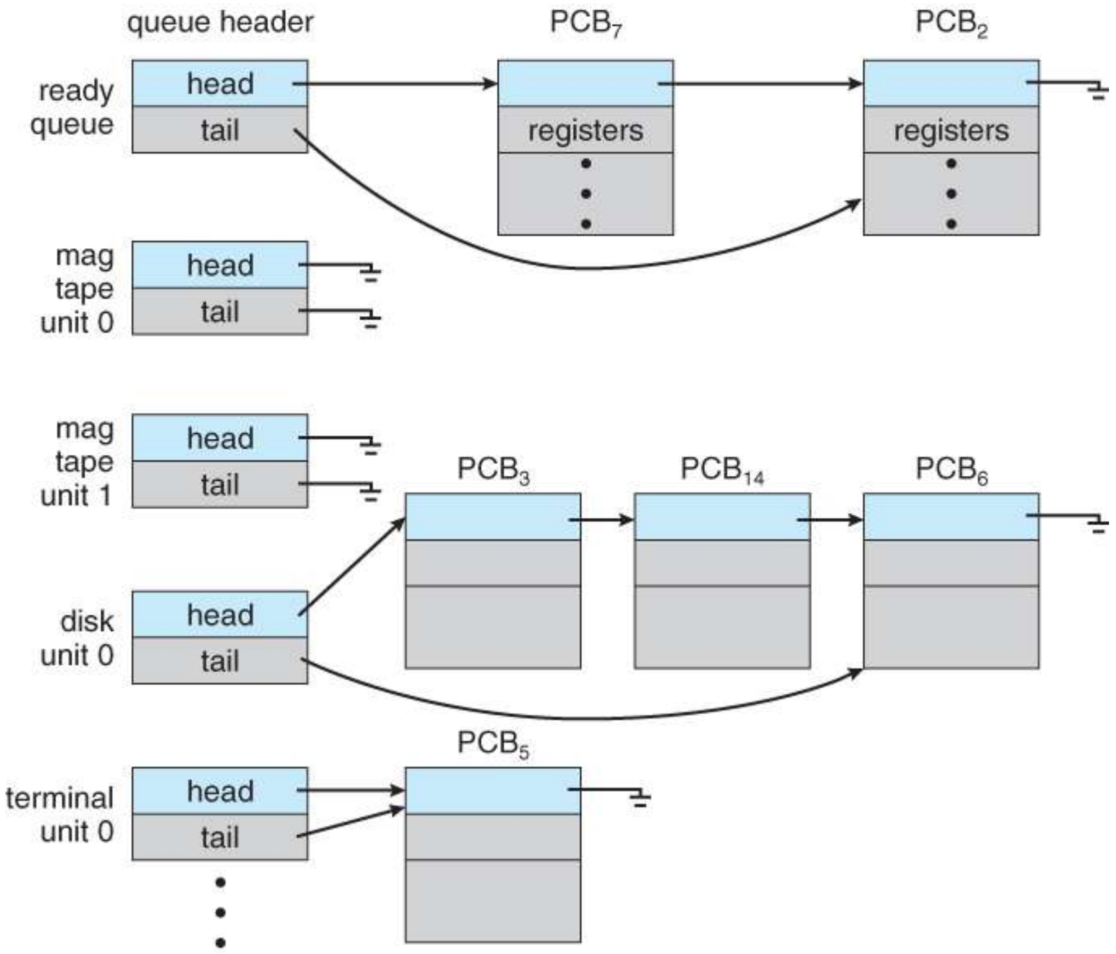
Process Scheduling

- 2 main goals of the process scheduling system:
  - keep the CPU busy at all times
  - deliver "acceptable" response times for all programs, particularly for interactive ones
- The process scheduler must meet these objectives by implementing suitable policies for swapping processes in and out of the CPU
- Note that these objectives can be conflicting!
  - Every time the OS steps in to swap processes it takes up time on the CPU to do so, which is thereby "lost" from doing any useful productive work

Process State Queues

- The OS mantains the PCBs of all the processes in state queues
- There is one queue for each of the 5 states a process can be in
- There is typically one queue for each I/O device (where processes wait for a device to become available or to deliver data)
- When the OS change the status of a process (e.g., from ready to running) the PCB is unlinked from the current queue and moved to the new one
- The OS may use different policies to manage each state queue

Process State Queues: Example



Process State Queues: Considerations

- How many PCBs can be in the Running Queue?

Process State Queues: Considerations

- How many PCBs can be in the Running Queue?
  - The Running Queue is bound by the number of cores available on the system
  - At each time, only one process can be executed on a CPU

Process State Queues: Considerations

- How many PCBs can be in the Running Queue?
  - The Running Queue is bound by the number of cores available on the system
  - At each time, only one process can be executed on a CPU
- What about the other queues?

Process State Queues: Considerations

- How many PCBs can be in the Running Queue?
  - The Running Queue is bound by the number of cores available on the system
  - At each time, only one process can be executed on a CPU
- What about the other queues?
  - They are basically unbounded as there is no theoretical limit on the number processes in new/ready/waiting/terminated states

Schedulers

- A long-term scheduler runs infrequently and is typical of a batch system or a very heavily loaded system

Schedulers

- A long-term scheduler runs infrequently and is typical of a batch system or a very heavily loaded system
- A short-term scheduler runs very frequently (about every 100 milliseconds) and must very quickly swap one process out of the CPU and swap in another one

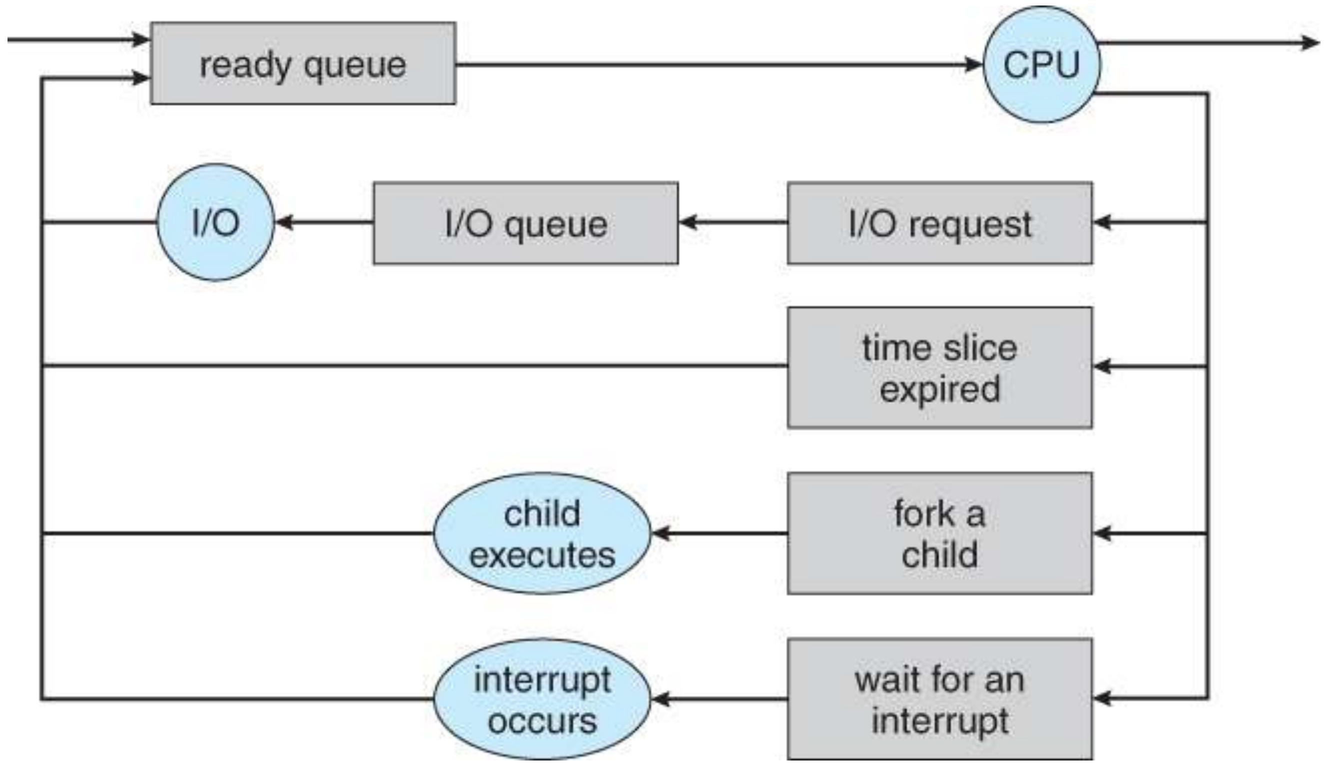
Schedulers

- A long-term scheduler runs infrequently and is typical of a batch system or a very heavily loaded system
- A short-term scheduler runs very frequently (about every 100 milliseconds) and must very quickly swap one process out of the CPU and swap in another one
- Some systems also employ a medium-term scheduler: when system loads get high, this scheduler allows smaller faster jobs to finish up quickly and clear the system

Schedulers

- A long-term scheduler runs infrequently and is typical of a batch system or a very heavily loaded system
- A short-term scheduler runs very frequently (about every 100 milliseconds) and must very quickly swap one process out of the CPU and swap in another one
- Some systems also employ a medium-term scheduler: when system loads get high, this scheduler allows smaller faster jobs to finish up quickly and clear the system
- An efficient scheduling system will select a good mix of CPU-bound processes and I/O bound processes

Schedulers: Queuing Diagram



Context Switch: What?

- It is the procedure used by the CPU to suspend the currently executing process in order to run a ready one

Context Switch: What?

- It is the procedure used by the CPU to suspend the currently executing process in order to run a ready one
- It is a highly costly operation because:
  - stopping the current process involves saving all of its internal state (PC, SP, other registers, etc.) to its PCB

Context Switch: What?

- It is the procedure used by the CPU to suspend the currently executing process in order to run a ready one
- It is a highly costly operation because:
  - stopping the current process involves saving all of its internal state (PC, SP, other registers, etc.) to its PCB
  - starting a ready process consists of loading all of its internal state (PC, SP, other registers, etc.) from its PCB

Context Switch: When?

- A context switch occurs due to any incoming trap
  - system calls, exceptions, or HW interrupts

Context Switch: When?

- A context switch occurs due to any incoming trap
  - system calls, exceptions, or HW interrupts
- Whenever an trap arrives, the CPU must:
  - perform a state-save of the currently running process
  - switch into kernel mode to handle the interrupt
  - perform a state-restore of the interrupted process

Context Switch: Fairness

- I/O-bound processes eventually get switched due to I/O requests
- CPU-bound processes, instead, could theoretically never issue any I/O requests
- To avoid CPU-bound processes hog the CPU, context switch is also triggered via HW timer interrupts (time quantum or slice)

Context Switch: Time Slice

- The maximum amount of time between two context switches

Context Switch: Time Slice

- The maximum amount of time between two context switches
- To ensure that at least a context switch occurs every, say, 50 ms
    - in practice, it can happen more frequently than that (e.g., due to I/O requests)

Context Switch: Time Slice

- The maximum amount of time between two context switches
- To ensure that at least a context switch occurs every, say, 50 ms
  - in practice, it can happen more frequently than that (e.g., due to I/O requests)
- Can be easily implemented in HW through timer interrupt

Context Switch: Time Slice

- The maximum amount of time between two context switches
- To ensure that at least a context switch occurs every, say, 50 ms
  - in practice, it can happen more frequently than that (e.g., due to I/O requests)
- Can be easily implemented in HW through timer interrupt
- Mechanism used by modern time-sharing multi-tasking OSs to increase system responsiveness (pseudo-parallelism)

Context Switch: How Often?

- The time taken to complete a context switch is just wasted CPU time

Context Switch: How Often?

- The time taken to complete a context switch is just wasted CPU time
- A smaller time slice results in more frequent context switches
  - maximizing responsiveness

Context Switch: How Often?

- The time taken to complete a context switch is just wasted CPU time
- A smaller time slice results in more frequent context switches
  - maximizing responsiveness
- A larger time slice results in less frequent context switches
  - minimizing wasted CPU time, therefore maximizing CPU utilization

Context Switch: How Often?

- The time taken to complete a context switch is just wasted CPU time
- A smaller time slice results in more frequent context switches
  - maximizing responsiveness
- A larger time slice results in less frequent context switches
  - minimizing wasted CPU time, therefore maximizing CPU utilization

Trade-off

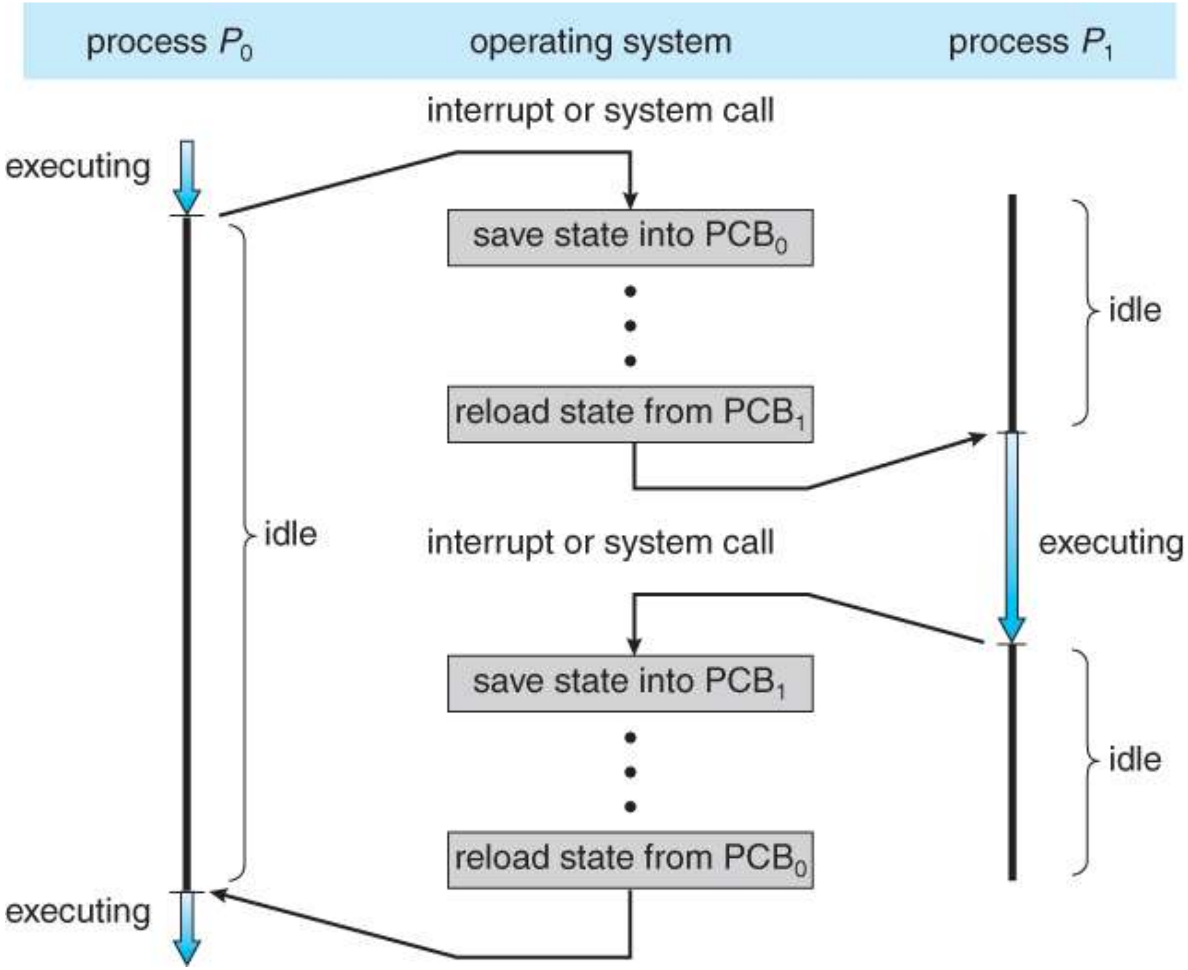
Context Switch: How Often?

- The time taken to complete a context switch is just wasted CPU time
- A smaller time slice results in more frequent context switches
  - maximizing responsiveness
- A larger time slice results in less frequent context switches
  - minimizing wasted CPU time, therefore maximizing CPU utilization

Trade-off

- Typical values of time slice are between 10 and 100 ms, and context switch takes around 10 μs, so the overhead is small relative to time slice

Context Switch: Example



Process Creation

- Processes may create other processes through specific system calls
  - The creator process is called parent of the new process, which is called child
  - The parent shares resources and privileges to its children
  - A parent can either wait for a child to complete, or continue in parallel

Process Creation

- Processes may create other processes through specific system calls
  - The creator process is called parent of the new process, which is called child
  - The parent shares resources and privileges to its children
  - A parent can either wait for a child to complete, or continue in parallel
- Each process is given an integer identifier (a.k.a. process identifier or PID)

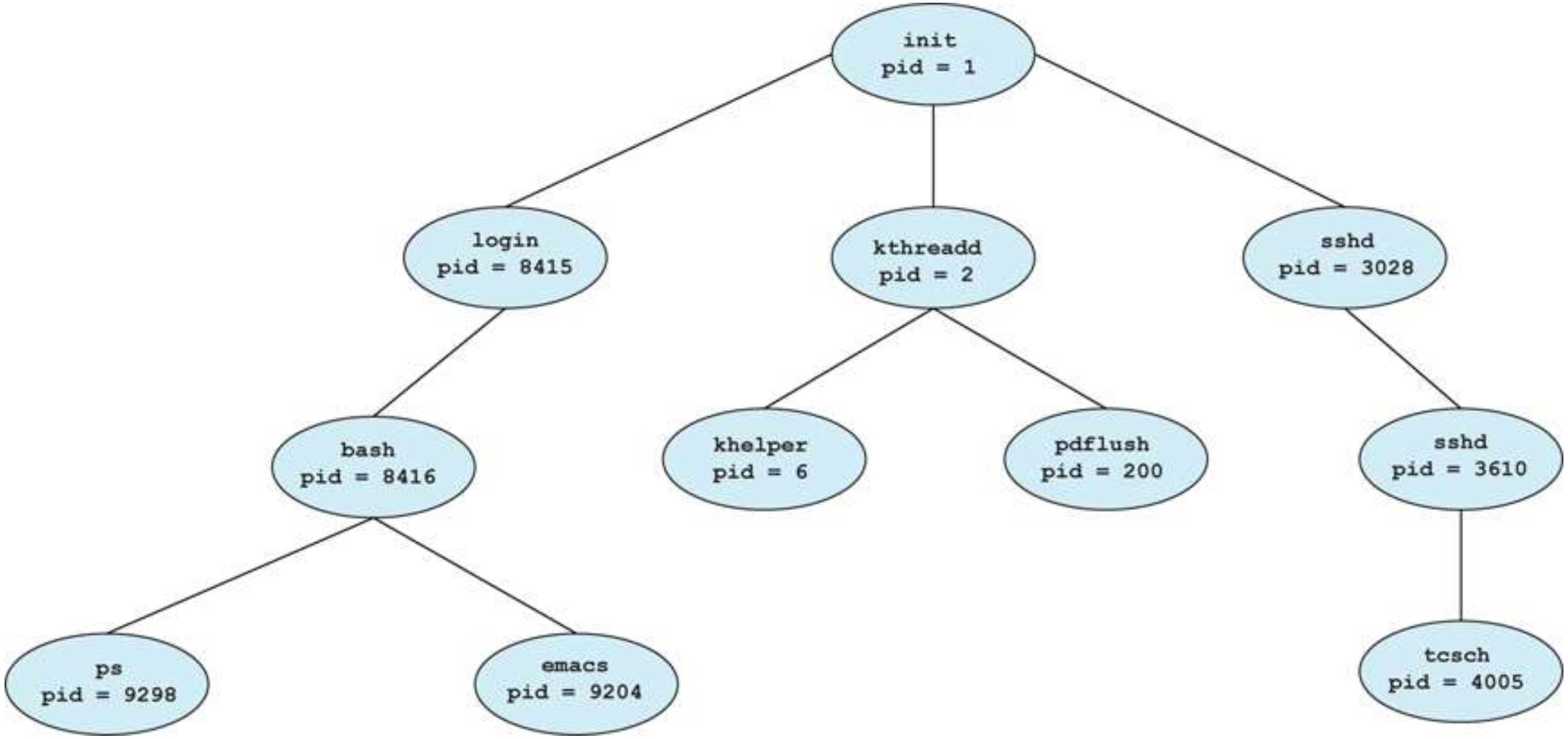
Process Creation

- Processes may create other processes through specific system calls
  - The creator process is called parent of the new process, which is called child
  - The parent shares resources and privileges to its children
  - A parent can either wait for a child to complete, or continue in parallel
- Each process is given an integer identifier (a.k.a. process identifier or PID)
- The parent PID (PPID) is also stored for each process

Process Creation: UNIX/Linux

- On typical UNIX systems the process scheduler is named sched, and is given PID 0
- The first thing it does at system startup time is to launch init, which gives that process PID 1
- init then launches all system daemons and user logins, and becomes the ultimate parent of all other processes
- Processes are created through the **`fork()`** system call

Process Creation: UNIX/Linux



Process Creation: Parent vs. Child Resources

- 2 possibilities for the address space of the child relative to the parent:

Process Creation: Parent vs. Child Resources

- 2 possibilities for the address space of the child relative to the parent:
  - The child may be an exact duplicate of the parent, sharing the same program and data segments in memory
    - Each will have their own PCB, including program counter, registers, and PID
    - This is the behavior of the **`fork`** system call in UNIX

Process Creation: Parent vs. Child Resources

- 2 possibilities for the address space of the child relative to the parent:
  - The child may be an exact duplicate of the parent, sharing the same program and data segments in memory
    - Each will have their own PCB, including program counter, registers, and PID
    - This is the behavior of the **`fork`** system call in UNIX
  - The child process may have a new program loaded into its address space, with all new code and data segments
    - This is the behavior of the **`spawn`** system calls in Windows
    - UNIX systems implement this as a second step, using the **`exec`** system call

Process Creation: Parent vs. Child Execution

- 2 options for the parent process after creating the child:

Process Creation: Parent vs. Child Execution

- 2 options for the parent process after creating the child:
  - Wait for the child process to terminate before proceeding by issuing a **`wait`** system call, for either a specific child or for any child

    (usual behavior of UNIX shell)

Process Creation: Parent vs. Child Execution

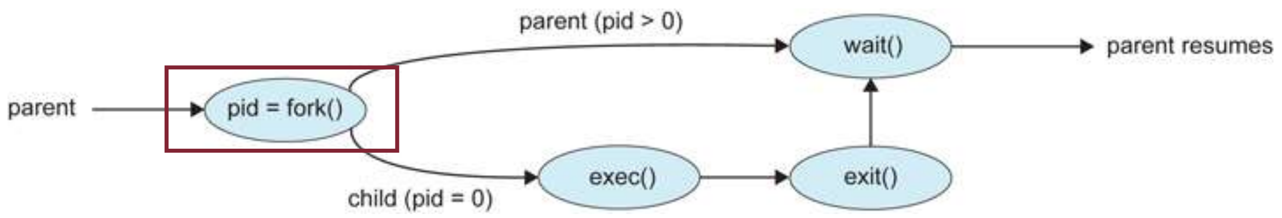
- 2 options for the parent process after creating the child:
  - Wait for the child process to terminate before proceeding by issuing a **`wait`** system call, for either a specific child or for any child

    (usual behavior of UNIX shell)
  - Run concurrently with the child, continuing to process without being blocked (when a UNIX shell runs a process as a background task using "&")

Process Creation: UNIX/Linux Code

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

int main()
{
pid_t pid;

   /* fork a child process */
   pid = fork();

   if (pid < 0) {/* error occurred */
      fprintf(stderr, "Fork Failed");
      exit(-1);
   }
   else if (pid == 0) {/* child process */
      execlp("/bin/ls","ls",NULL);
   }
   else {/* parent process */
      /* parent will wait for the child to complete */
      wait(NULL);
      printf("Child Complete");
      exit(0);
   }
}
```

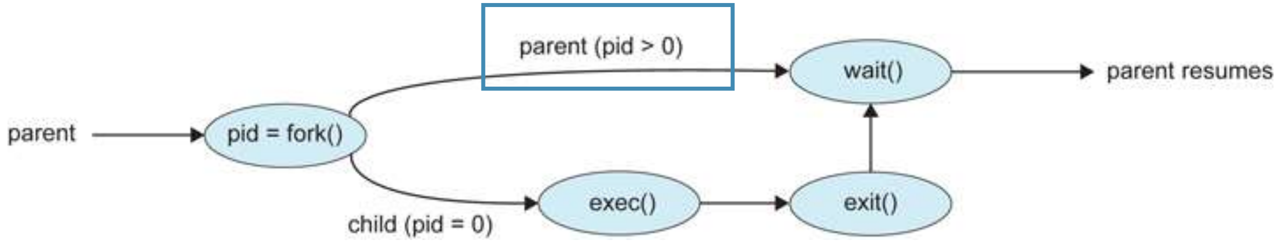
Figure 3.10 C program forking a separate process.



Process Creation: UNIX/Linux Code

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

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) {/* error occurred */
      fprintf(stderr, "Fork Failed");
      exit(-1);
    }
    else if (pid == 0) {/* child process */
      execlp("/bin/ls","ls",NULL);
    }
    else {/* parent process */
      /* parent will wait for the child to complete */
      wait(NULL);
      printf("Child Complete");
      exit(0);
    }
}
```

Figure 3.10 C program forking a separate process.

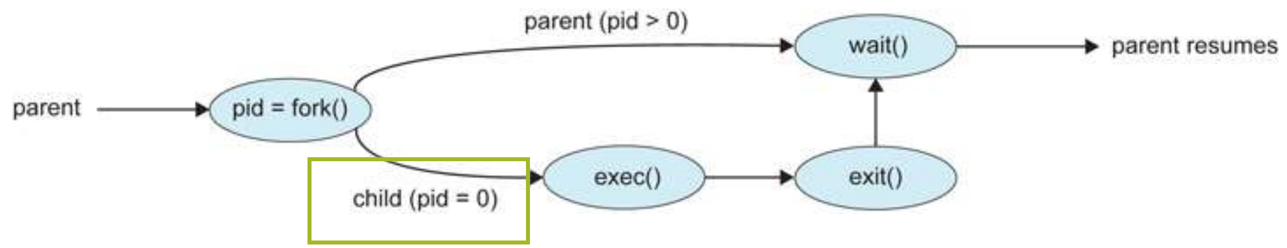


In the parent process, **`fork()`** returns the PID of the child

Process Creation: UNIX/Linux Code

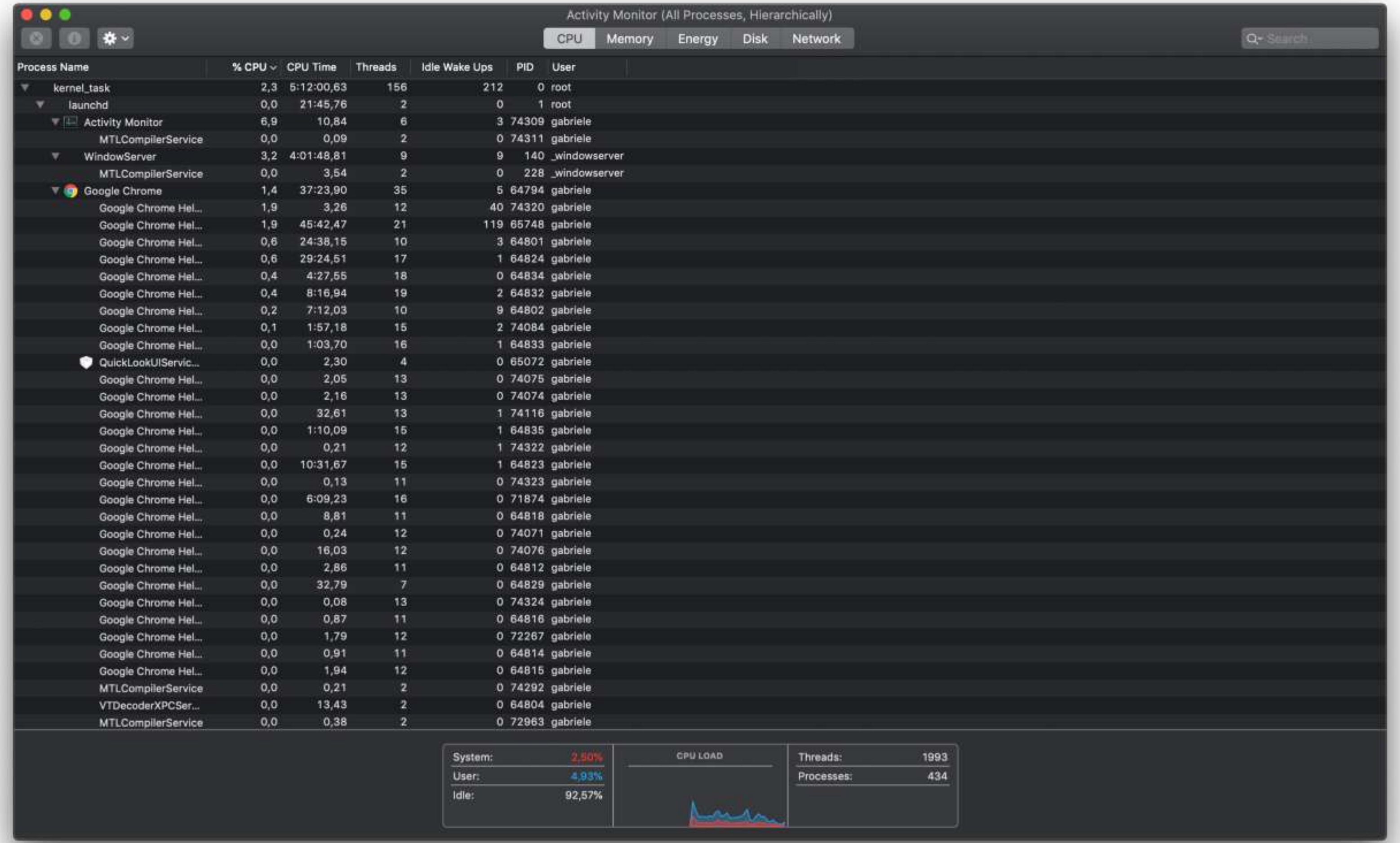
```c
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) {/* error occurred */
      fprintf(stderr, "Fork Failed");
      exit(-1);
    }
    else if (pid == 0) {/* child process */
      execlp("/bin/ls","ls",NULL);
    }
    else {/* parent process */
      /* parent will wait for the child to complete */
      wait(NULL);
      printf("Child Complete");
      exit(0);
    }
}
```

Figure 3.10 C program forking a separate process.



In the child process, it returns 0

Process Creation: Activity Monitor

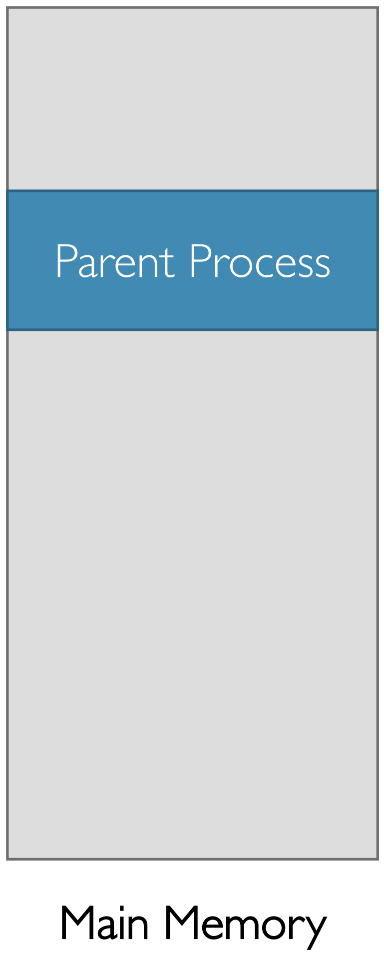


Activity Monitor (All Processes, Hierarchically)

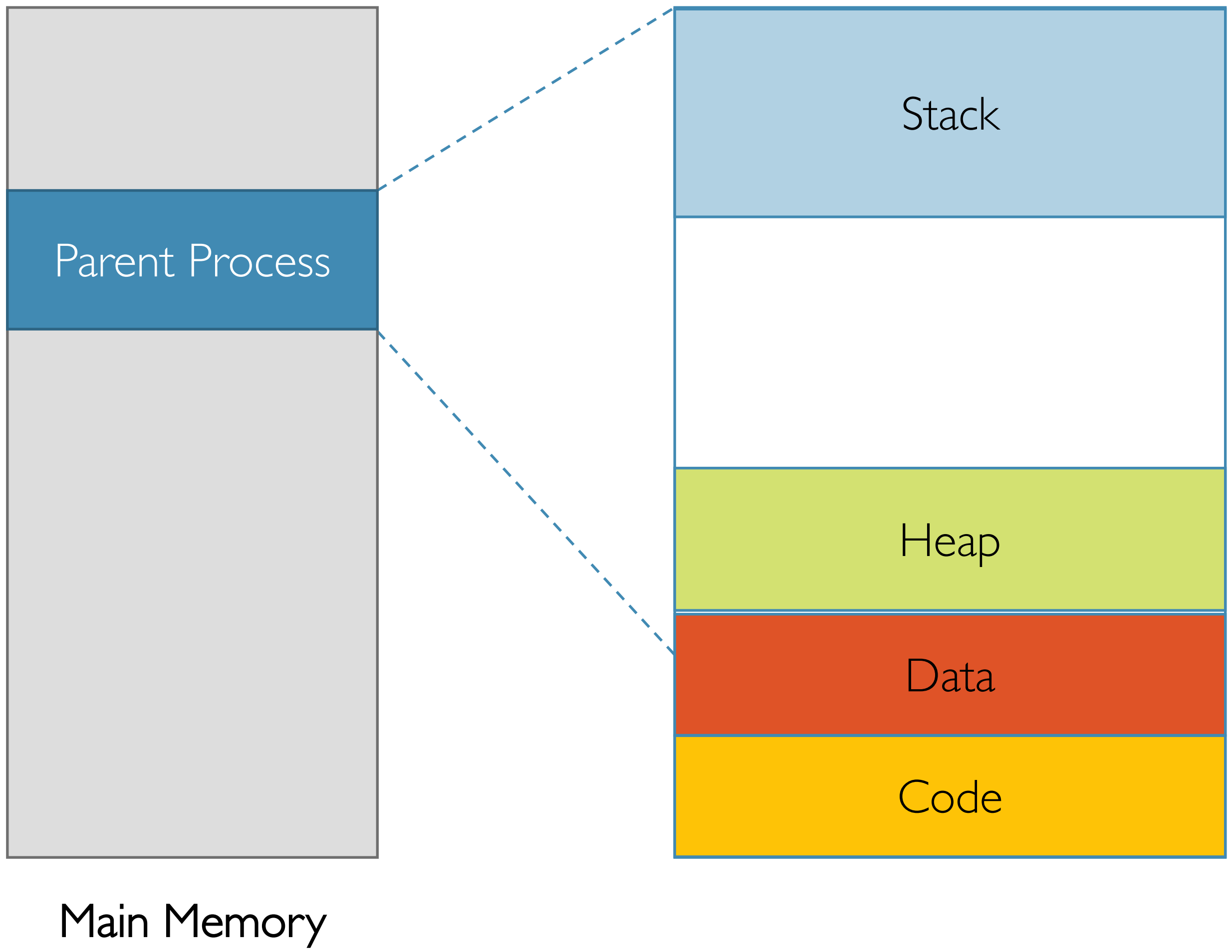
Process Name	% CPU	CPU Time	Threads	Idle Wake Ups	PID	User
kernel_task	2,3	5:12:00,63	156	212	0	root
launchd	0,0	21:45,76	2	0	1	root
Activity Monitor	6,9	10,84	6	3	74309	gabriele
MTLCompilerService	0,0	0,09	2	0	74311	gabriele
WindowServer	3,2	4:01:48,81	9	9	140	_windowserver
MTLCompilerService	0,0	3,54	2	0	228	_windowserver
Google Chrome	1,4	37:23,90	35	5	64794	gabriele
Google Chrome Hel...	1,9	3,26	12	40	74320	gabriele
Google Chrome Hel...	1,9	45:42,47	21	119	65748	gabriele
Google Chrome Hel...	0,6	24:38,15	10	3	64801	gabriele
Google Chrome Hel...	0,6	29:24,51	17	1	64824	gabriele
Google Chrome Hel...	0,4	4:27,55	18	0	64834	gabriele
Google Chrome Hel...	0,4	8:16,94	19	2	64832	gabriele
Google Chrome Hel...	0,2	7:12,03	10	9	64802	gabriele
Google Chrome Hel...	0,1	1:57,18	15	2	74084	gabriele
Google Chrome Hel...	0,0	1:03,70	16	1	64833	gabriele
QuickLookUIServic...	0,0	2,30	4	0	65072	gabriele
Google Chrome Hel...	0,0	2,05	13	0	74075	gabriele
Google Chrome Hel...	0,0	2,16	13	0	74074	gabriele
Google Chrome Hel...	0,0	32,61	13	1	74116	gabriele
Google Chrome Hel...	0,0	1:10,09	15	1	64835	gabriele
Google Chrome Hel...	0,0	0,21	12	1	74322	gabriele
Google Chrome Hel...	0,0	10:31,67	15	1	64823	gabriele
Google Chrome Hel...	0,0	0,13	11	0	74323	gabriele
Google Chrome Hel...	0,0	6:09,23	16	0	71874	gabriele
Google Chrome Hel...	0,0	8,81	11	0	64818	gabriele
Google Chrome Hel...	0,0	0,24	12	0	74071	gabriele
Google Chrome Hel...	0,0	16,03	12	0	74076	gabriele
Google Chrome Hel...	0,0	2,86	11	0	64812	gabriele
Google Chrome Hel...	0,0	32,79	7	0	64829	gabriele
Google Chrome Hel...	0,0	0,08	13	0	74324	gabriele
Google Chrome Hel...	0,0	0,87	11	0	64816	gabriele
Google Chrome Hel...	0,0	1,79	12	0	72267	gabriele
Google Chrome Hel...	0,0	0,91	11	0	64814	gabriele
Google Chrome Hel...	0,0	1,94	12	0	64815	gabriele
MTLCompilerService	0,0	0,21	2	0	74292	gabriele
VTDecoderXPCSer...	0,0	13,43	2	0	64804	gabriele
MTLCompilerService	0,0	0,38	2	0	72963	gabriele

Hierarchy of Processes
(i.e., process tree)

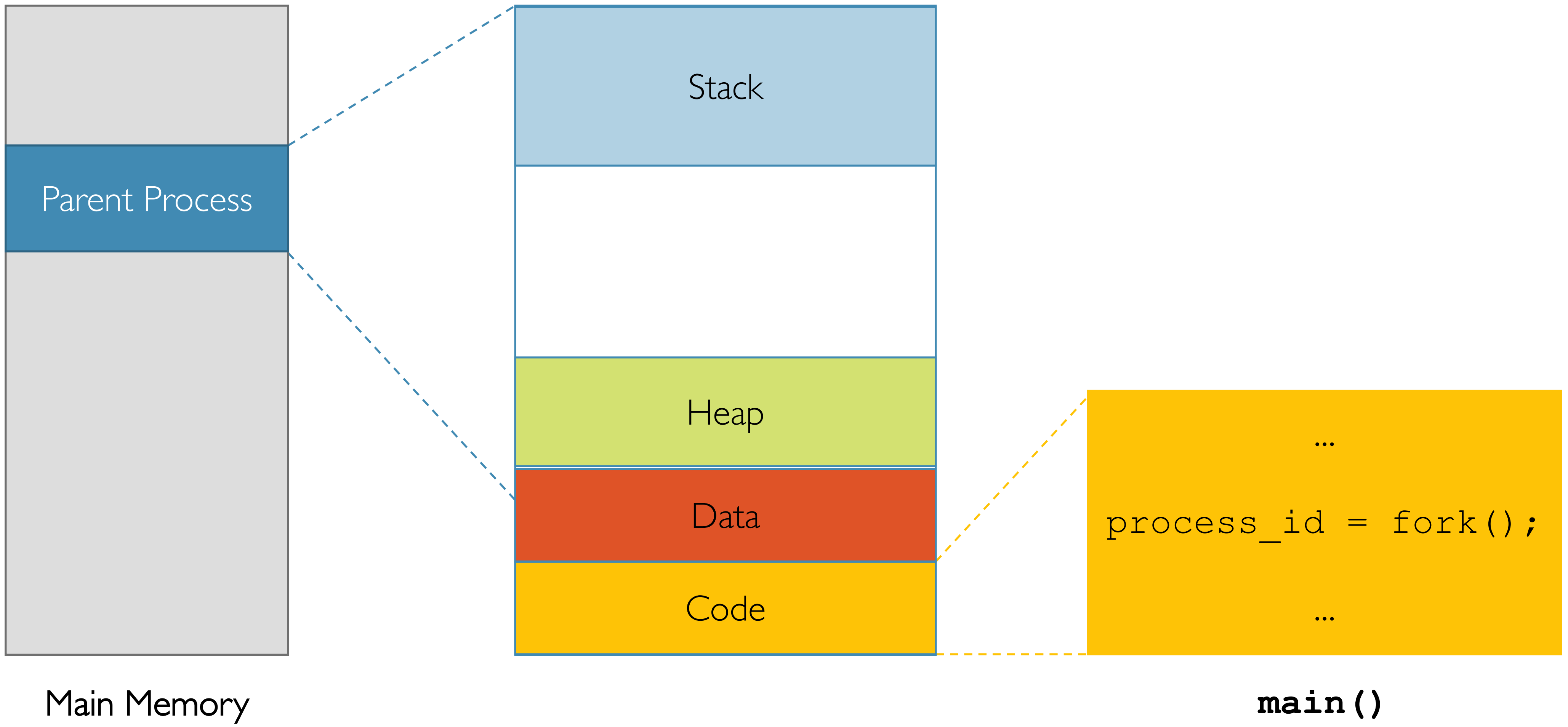
Process Creation: Parent vs. Child Layout



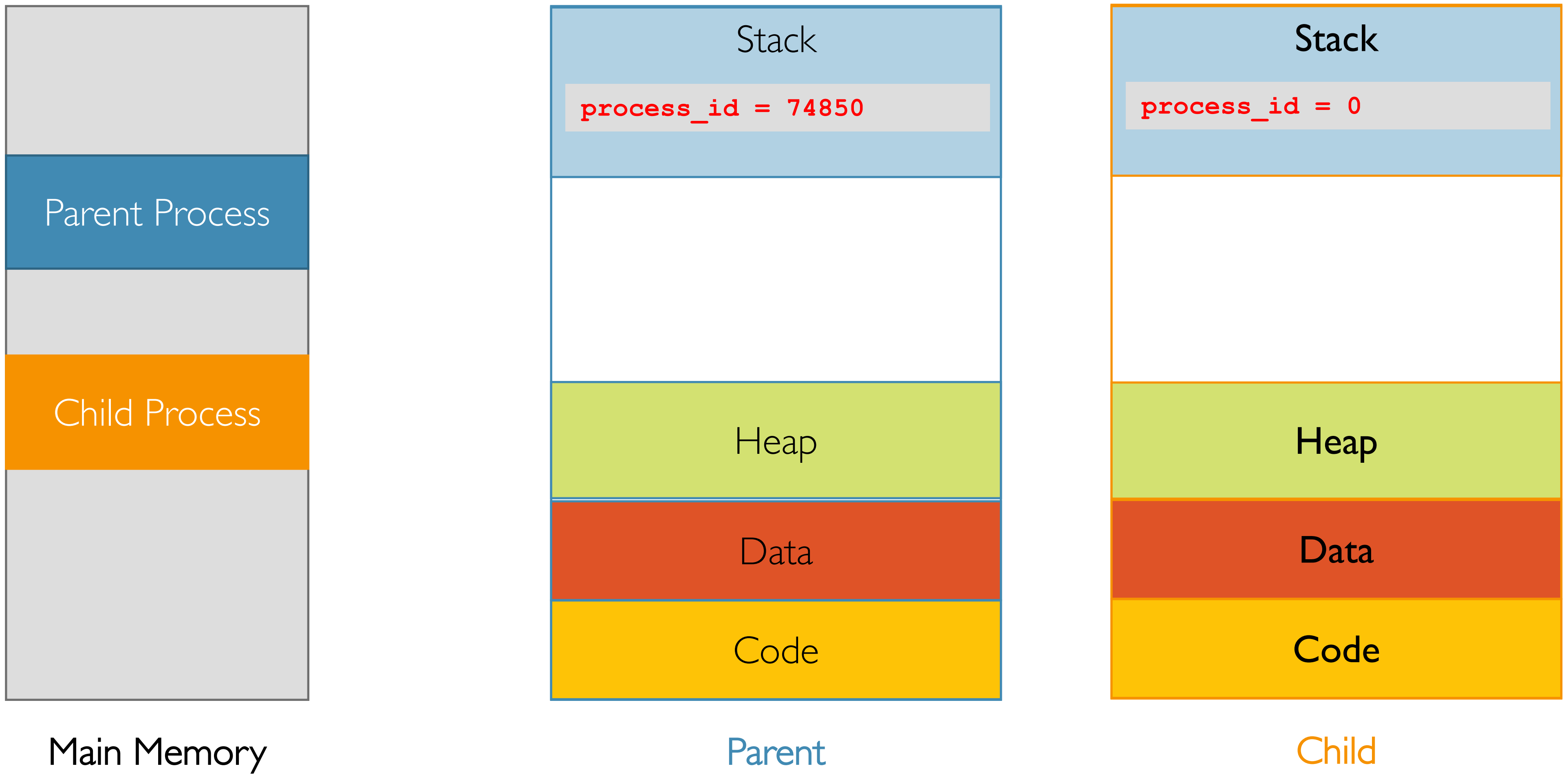
Process Creation: Parent vs. Child Layout



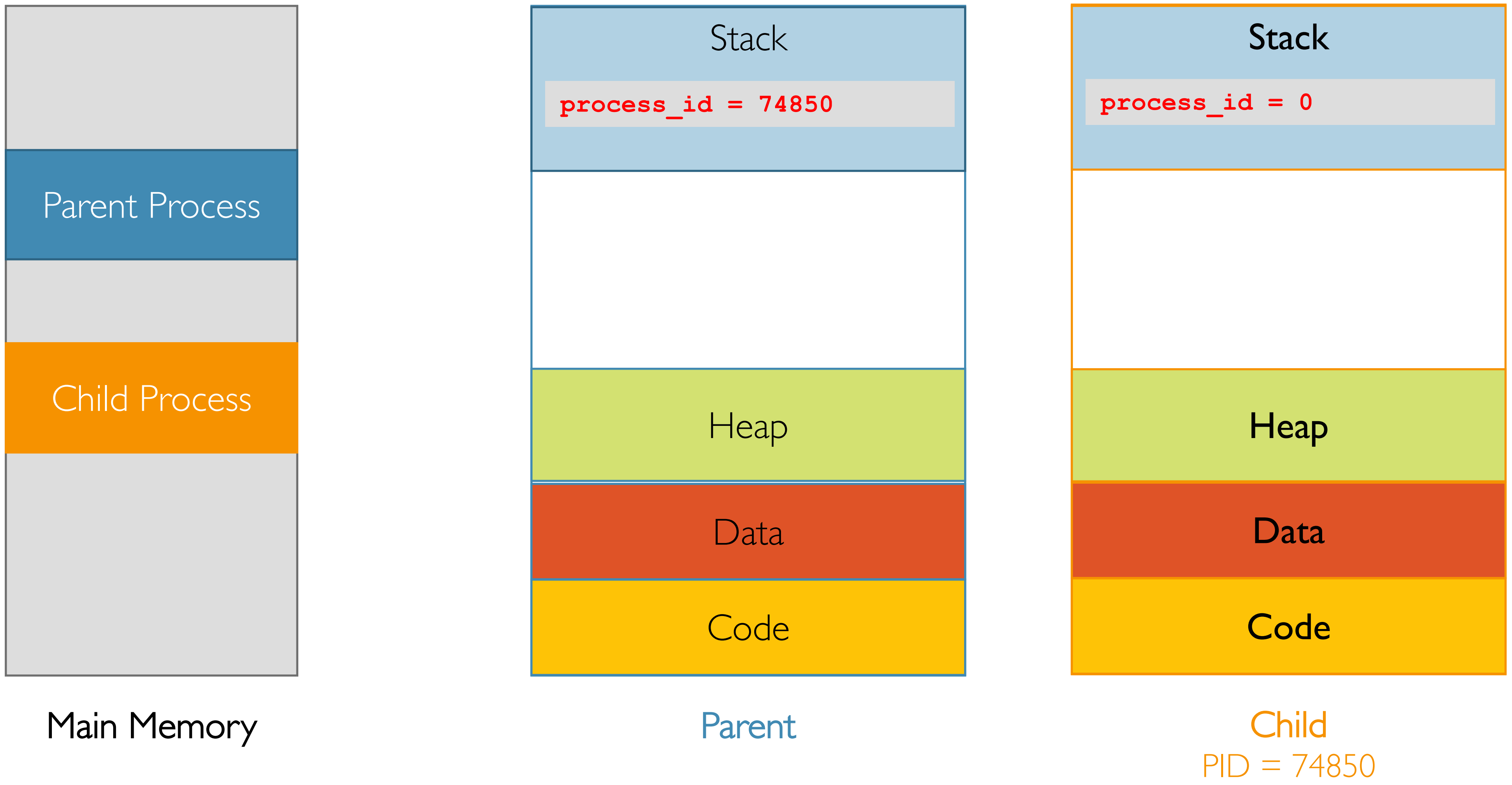
Process Creation: Parent vs. Child Layout



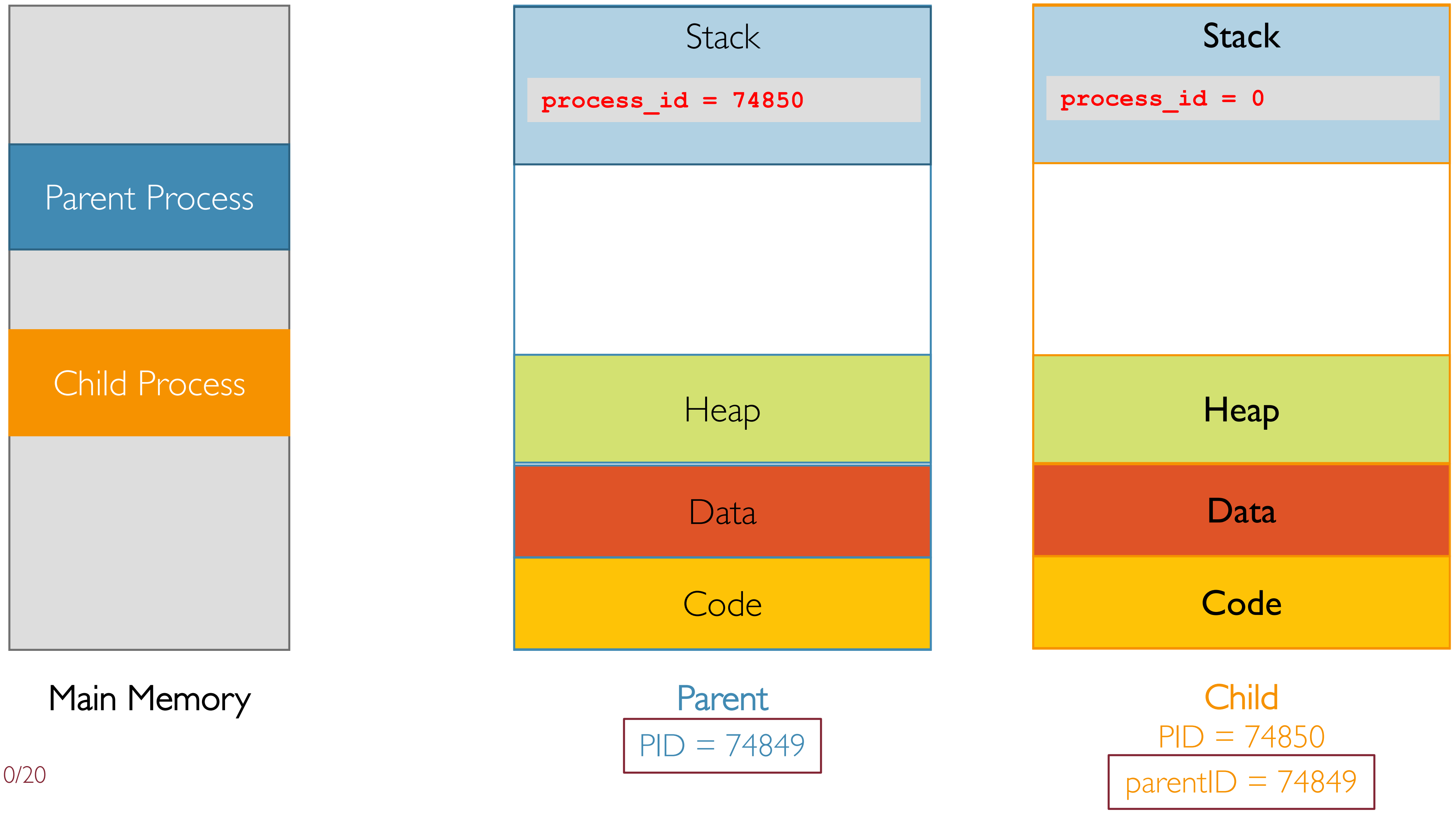
Process Creation: Parent vs. Child Layout



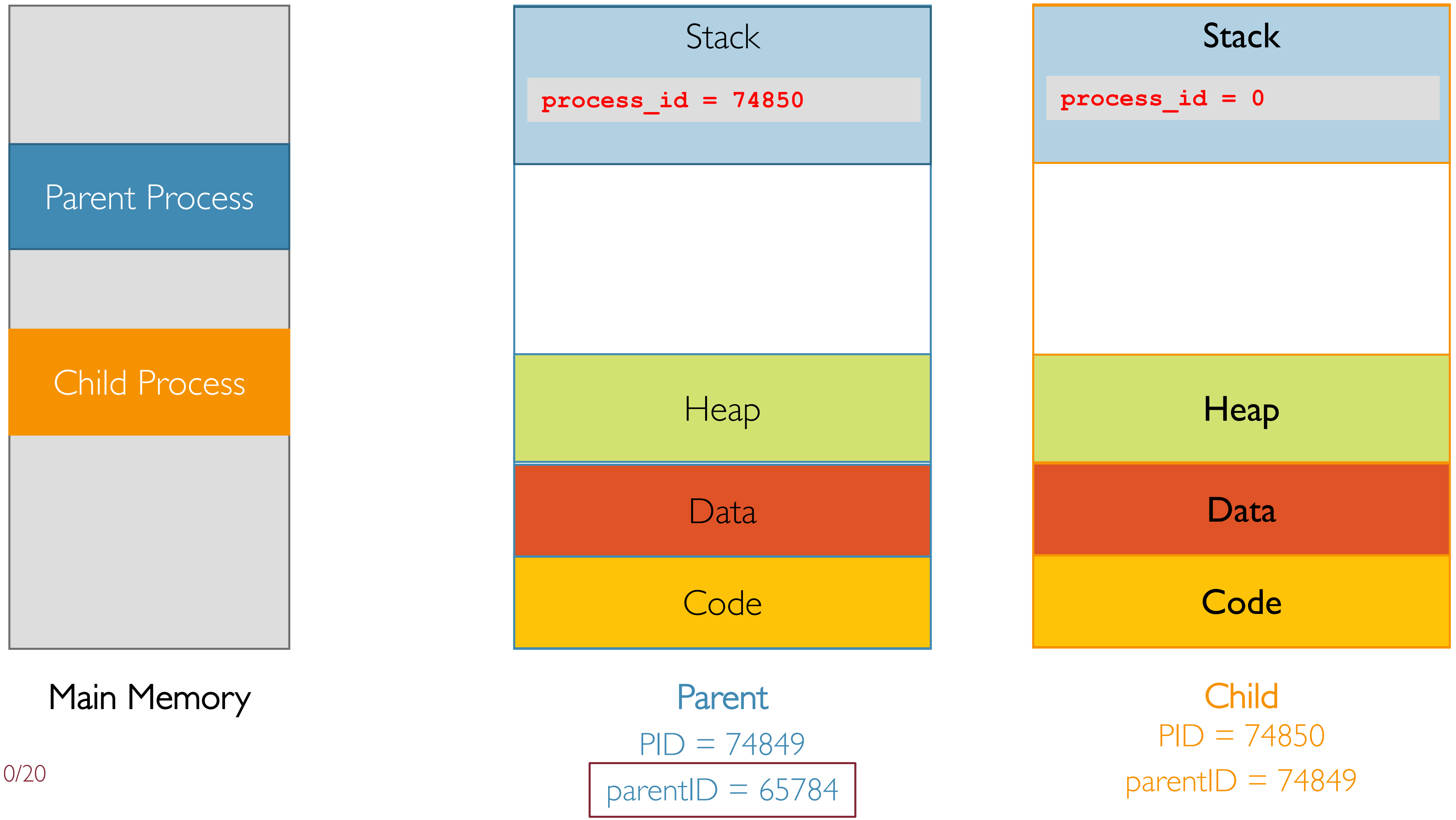
Process Creation: Parent vs. Child Layout



Process Creation: Parent vs. Child Layout



Process Creation: Parent vs. Child Layout



Process Creation: Code Example

```cpp
#include <iostream>
#include <unistd.h>

using namespace std;

int main() {

    cout << "Current process ID is: " << getpid() << endl;
    cout << "\nCurrent parent's process ID is: " << getppid() << endl;

    int pid;
    pid = fork();
    // once the fork() system call returns,
    // both the parent and the child processes will resume from this point onward

    if (pid == 0) { // child
        cout << "\nThis is the child process with process ID = "
             << getpid() << endl;
        cout << "\nThis is the child process with parent's process ID = "
             << getppid() << endl;
    }
    else { // parent
        sleep(1); // to ensure the child process finishes before the parent

        cout << "\nThis is the parent process with process ID = "
             << getpid() << endl;
        cout << "\nThis is the parent process with parent's process ID = "
             << getppid() << endl;

    }
    return 0;
}
```

Process Creation: Code Example

```cpp
#include <iostream>
#include <unistd.h>

using namespace std;

int main() {

    cout << "Current process ID is: " << getpid() << endl;
    cout << "\nCurrent parent's process ID is: " << getppid() << endl;

    int pid;
    pid = fork();
    // once the fork() system call returns,
    // both the parent and the child processes will resume from this point onward

    if (pid == 0) { // child
        cout << "\nThis is the child process with process ID = "
             << getpid() << endl;
        cout << "\nThis is the child process with parent's process ID = "
             << getppid() << endl;
    }
    else { // parent
        sleep(1); // to ensure the child process finishes before the parent

        cout << "\nThis is the parent process with process ID = "
             << getpid() << endl;
        cout << "\nThis is the parent process with parent's process ID = "
             << getppid() << endl;

    }
    return 0;
}
```

What happens if the child sleeps rather than the parent?

Process Creation: What's Next?

- So far, we have seen how **`fork`** system call is able to make a complete copy of an existing process
- However, this ability alone is not that useful, right?
- Our ultimate goal is to create new yet different processes, not just copies of a single one!

Process Creation: The Example of UNIX Shell

- When we log in to a UNIX machine a shell process is usually started

Process Creation: The Example of UNIX Shell

- When we log in to a UNIX machine a shell process is usually started
- Every command we type into the shell creates a new child process whose parent is the shell itself

Process Creation: The Example of UNIX Shell

- When we log in to a UNIX machine a shell process is usually started
- Every command we type into the shell creates a new child process whose parent is the shell itself
- Implicitly, 2 system calls take place: **`fork`** and **`exec`**
  - the former creates a new process, whilst the latter execute the new process
  - e.g., try typing **`emacs`** on your shell

Process Creation: The Example of UNIX Shell

- When we log in to a UNIX machine a shell process is usually started
- Every command we type into the shell creates a new child process whose parent is the shell itself
- Implicitly, 2 system calls take place: **`fork`** and **`exec`**
  - the former creates a new process, whilst the latter execute the new process
  - e.g., try typing **`emacs`** on your shell
- NOTE: adding "&" at the end of the command will run the child process in parallel with the parent shell (background)

Process Creation and Execution: Example

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

using namespace std;

int main() {

    int current_pid = getpid();
    cout << "Current process ID is: " << current_pid << endl;

    string progStr;
    // read the name of the program we want to start
    getline(cin, progStr);
    const char *prog = progStr.c_str();

    int pid = fork();

    if (pid == 0) { // child
        execlp(prog, prog, 0); // load the program
        // if prog can actually be started, we will never get to the
        // following statement, as the child process will be replaced by prog!
        printf("Can't load the program %s\n", prog);
    }
    else { // parent
        sleep(1); // give some time to the child process to starting up
        waitpid(pid, 0, 0); // wait for child process to terminate
        printf("Program %s finished!\n", prog);
    }
    return 0;
}
```

`execlp` loads the program whose name is read from **`stdin`**

```
int execlp(const char *file, const char *arg, ...);
```

Process Creation and Execution: Example

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

using namespace std;

int main() {

    int current_pid = getpid();
    cout << "Current process ID is: " << current_pid << endl;

    string progStr;
    // read the name of the program we want to start
    getline(cin, progStr);
    const char *prog = progStr.c_str();

    int pid = fork();

    if (pid == 0) { // child
        execlp(prog, prog, 0); // load the program
        // if prog can actually be started, we will never get to the
        // following statement, as the child process will be replaced by prog!
        printf("Can't load the program %s\n", prog);
    }
    else { // parent
        sleep(1); // give some time to the child process to starting up
        waitpid(pid, 0, 0); // wait for child process to terminate
        printf("Program %s finished!\n", prog);
    }
    return 0;
}
```

`execlp` loads the program whose name is read from **`stdin`**

```
int execlp(const char *file, const char *arg, ...);
```

path to executable

Process Creation and Execution: Example

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

using namespace std;

int main() {

    int current_pid = getpid();
    cout << "Current process ID is: " << current_pid << endl;

    string progStr;
    // read the name of the program we want to start
    getline(cin, progStr);
    const char *prog = progStr.c_str();

    int pid = fork();

    if (pid == 0) { // child
        execlp(prog, prog, 0); // load the program
        // if prog can actually be started, we will never get to the
        // following statement, as the child process will be replaced by prog!
        printf("Can't load the program %s\n", prog);
    }
    else { // parent
        sleep(1); // give some time to the child process to starting up
        waitpid(pid, 0, 0); // wait for child process to terminate
        printf("Program %s finished!\n", prog);
    }
    return 0;
}
```

`execlp` loads the program whose name is read from **`stdin`**

```
int execlp(const char *file, const char *arg, ...);
```

`argv[0]`

Process Creation and Execution: Example

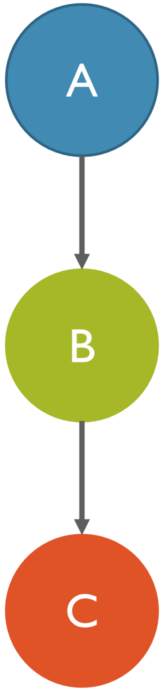
```cpp
#include <iostream>
#include <unistd.h>
#include <sys/wait.h>
#include <stdio.h>
#include <string.h>

using namespace std;

int main() {

    int current_pid = getpid();
    cout << "Current process ID is: " << current_pid << endl;

    string progStr;
    // read the name of the program we want to start
    getline(cin, progStr);
    const char *prog = progStr.c_str();

    int pid = fork();

    if (pid == 0) { // child
        execlp(prog, prog, 0); // load the program
        // if prog can actually be started, we will never get to the
        // following statement, as the child process will be replaced by prog!
        printf("Can't load the program %s\n", prog);
    }
    else { // parent
        sleep(1); // give some time to the child process to starting up
        waitpid(pid, 0, 0); // wait for child process to terminate
        printf("Program %s finished!\n", prog);
    }
    return 0;
}
```

`waitpid` allows the parent to wait for a child process to finish

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

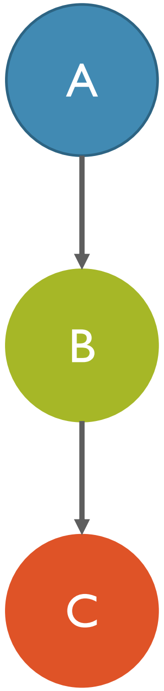
Process Creation and Execution: Exercise

How do we create the following process hierarchy using **`fork`** and possibly **`exec`**?



Process Creation and Execution: Exercise

How do we create the following process hierarchy using `fork` and possibly `exec`?

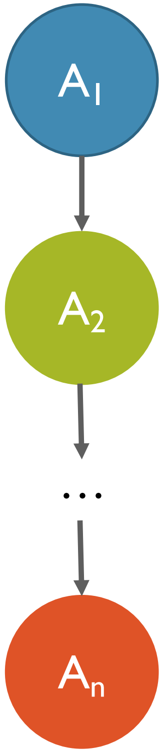



```
int pid = fork();

if(pid == 0) { // A's child (B)

    pid = fork();

    if(pid == 0) { // B's child (C)

        …
        execlp(…);
    }
    else { // B
        …
    }
}
else { // A
    …
}
```

Process Creation and Execution: Exercise

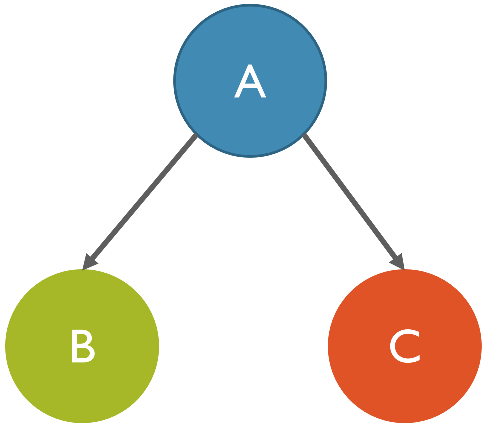
More generally, we will need *n-1* **`fork`** and **`if-else`**

if we want to create a sequence of *n* processes



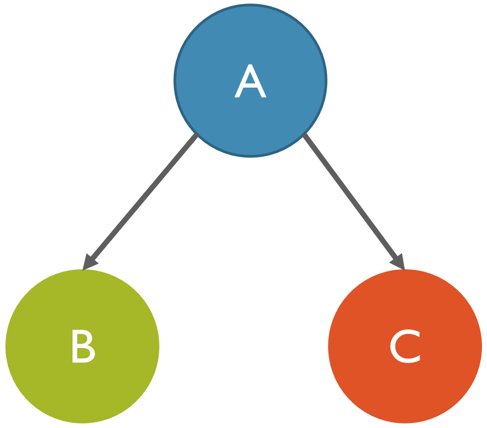
Process Creation and Execution: Exercise

How do we create the following process hierarchy using **`fork`** and possibly **`exec`**?



Process Creation and Execution: Exercise

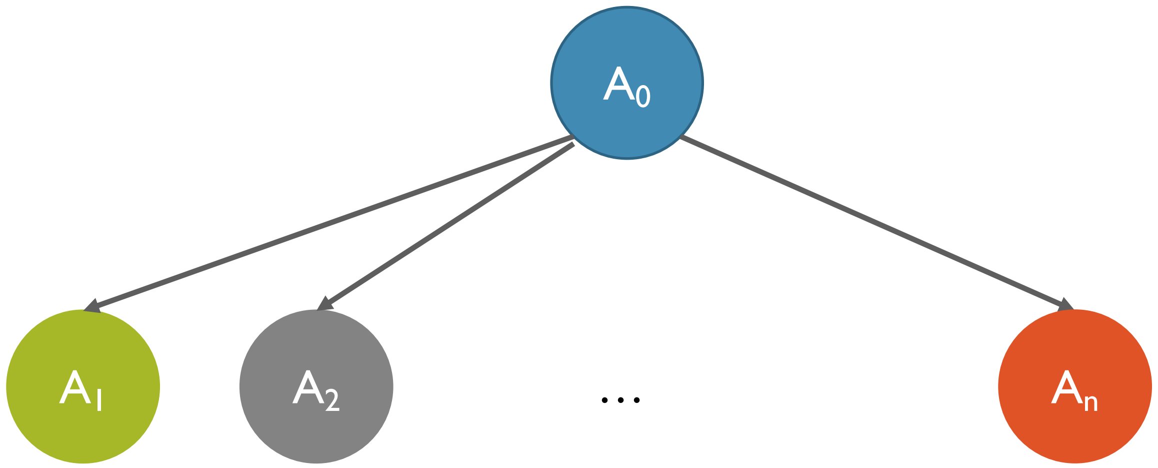
How do we create the following process hierarchy using `fork` and possibly `exec`?




```
int pid = fork();

if(pid == 0) { // A's child (B)
        …
        execlp(…);
}
else { // A

    pid = fork();

    if(pid == 0) { // A's child (C)
        …
        execlp(…);
    }
}
```

Process Creation and Execution: Exercise

More generally, if we want to create *n* child processes all having the same parent




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

    if(fork() == 0) { // A0's child
        …
        execlp(…);
    }
    // else we are in the parent: keep forking
}
// back in the parent A0

// wait for all children to terminate
for(int i=0;i<n;i++) {
    wait(NULL);
}
```

Process Creation and Execution: Be Careful!

What will happen if we do the following?

```
while(1) {
    fork();
}
```

Process Creation and Execution: Be Careful!

What will happen if we do the following?

```
while(1) {
    fork();
}
```

Infinite number of child processes growing with an exponential rate

Recap of System Calls Seen So Far

- **`fork`** → spawn a new child process as an exact copy of the parent

Recap of System Calls Seen So Far

- **`fork`** → spawn a new child process as an exact copy of the parent
- **`execlp`** → replaces the program of the current process with the input named program

Recap of System Calls Seen So Far

- **`fork`** → spawn a new child process as an exact copy of the parent
- **`execlp`** → replaces the program of the current process with the input named program
- **`sleep`** → suspends the execution for a certain amount of seconds

Recap of System Calls Seen So Far

- **`fork`** → spawn a new child process as an exact copy of the parent
- **`execlp`** → replaces the program of the current process with the input named program
- **`sleep`** → suspends the execution for a certain amount of seconds
- **`wait`/`waitpid`** → wait for any/a specific process to finish execution

Process Termination

- Processes may request their own termination by making the **`exit`** system call, typically returning an int
- This int is passed along to the parent if it is doing a **`wait`**
- It is usually 0 on successful completion and some non-zero in the event of problems

Process Termination

- Processes may also be terminated by the system for a variety of reasons:
  - The inability of the system to deliver necessary system resources
  - In response to a `kill` command, or other un handled process interrupt

Process Termination

- Processes may also be terminated by the system for a variety of reasons:
  - The inability of the system to deliver necessary system resources
  - In response to a **`kill`** command, or other un handled process interrupt
- A parent may kill its children if the task assigned to them is no longer needed

Process Termination

- Processes may also be terminated by the system for a variety of reasons:
  - The inability of the system to deliver necessary system resources
  - In response to a **`kill`** command, or other un handled process interrupt
- A parent may kill its children if the task assigned to them is no longer needed
- If the parent exits, the system may or may not allow the child to continue without a parent
  - On UNIX systems, orphaned processes are generally inherited by **`init`**, which then proceeds to kill them

Process Termination

- When a process terminates, all of its system resources are freed up, open files flushed and closed, etc.

Process Termination

- When a process terminates, all of its system resources are freed up, open files flushed and closed, etc.
- The process termination status and execution times are returned to the parent if this is waiting for the child to terminate
  - Or eventually to **`init`** if the process becomes an orphan

Process Termination

- When a process terminates, all of its system resources are freed up, open files flushed and closed, etc.
- The process termination status and execution times are returned to the parent if this is waiting for the child to terminate
  - Or eventually to **`init`** if the process becomes an orphan
- Processes which are trying to terminate but cannot because their parent is not waiting for them are called zombies
  - Eventually inherited by **`init`** as orphans and killed

Interprocess Communication

- Processes can be either independent or cooperating

Interprocess Communication

- Processes can be either independent or cooperating
- Independent processes → operate concurrently on a system and can neither affect or be affected by other processes

Interprocess Communication

- Processes can be either independent or cooperating
- Independent processes → operate concurrently on a system and can neither affect or be affected by other processes
- Cooperating processes → can affect or be affected by other processes in order to achieve a common task

Cooperating Processes: Why Do We Need Them?

- Information sharing → There may be several processes which need access to the same file (e.g., pipelines)

Cooperating Processes: Why Do We Need Them?

- Information sharing → There may be several processes which need access to the same file (e.g., pipelines)
- Computation speedup → A problem can be solved faster if it can be broken down into sub-tasks to be solved simultaneously

Cooperating Processes: Why Do We Need Them?

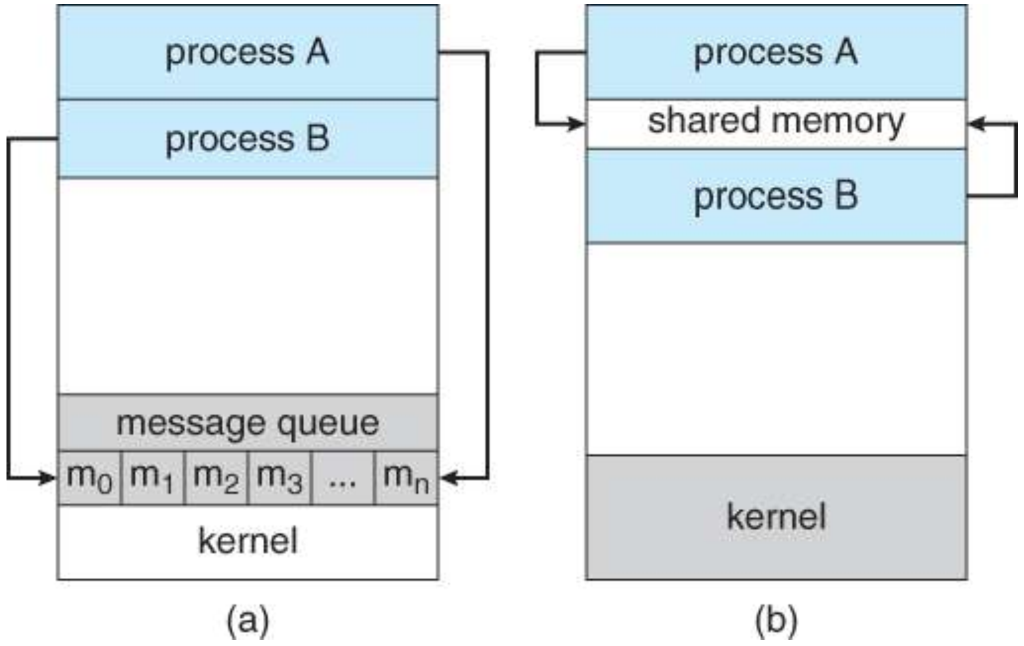
- Information sharing → There may be several processes which need access to the same file (e.g., pipelines)
- Computation speedup → A problem can be solved faster if it can be broken down into sub-tasks to be solved simultaneously
- Modularity → The most efficient architecture may be to break a system down into cooperating modules

Cooperating Processes: Why Do We Need Them?

- Information sharing → There may be several processes which need access to the same file (e.g., pipelines)
- Computation speedup → A problem can be solved faster if it can be broken down into sub-tasks to be solved simultaneously
- Modularity → The most efficient architecture may be to break a system down into cooperating modules
- Convenience → Even a single user may be multi-tasking, such as editing, compiling, printing, and running the same code in different windows

Cooperating Processes: Communication

- 2 possible ways for cooperating processes to communicate:



Message Passing

Shared Memory

Shared Memory vs. Message Passing

- Shared Memory
  - Faster once it is set up, as no system calls are needed
  - More complicated to set up, and doesn't work as well across multiple computers
  - Preferable when (large amount of) information must be shared on the same computer

Shared Memory vs. Message Passing

- Shared Memory
  - Faster once it is set up, as no system calls are needed
  - More complicated to set up, and doesn't work as well across multiple computers
  - Preferable when (large amount of) information must be shared on the same computer
- Message Passing
  - Slower as it requires system calls for every message transfer
  - Simpler to set up and works well across multiple computers
  - Preferable when the amount and/or frequency of data transfers is small, or when multiple computers are involved

Shared Memory Systems

- The memory to be shared is initially within the address space of a particular process
- This needs to make system calls in order to make that memory publicly available to other processes
- Other processes must make their own system calls to attach the shared memory onto their address space

Message Passing Systems

- Must support at least system calls for sending and receiving messages
- A communication link must be established between the cooperating processes before messages can be sent
- 3 key issues to be solved:
  - direct or indirect communication (i.e., naming)
  - synchronous or asynchronous communication
  - automatic or explicit buffering

Message Passing Systems: Naming

- Direct communication → the sender must know the name of the receiver to which it wishes to send a message
  - one-to-one link between every sender-receiver pair
  - for symmetric communication, the receiver must also know the name of the sender

Message Passing Systems: Naming

- Direct communication → the sender must know the name of the receiver to which it wishes to send a message
  - one-to-one link between every sender-receiver pair
  - for symmetric communication, the receiver must also know the name of the sender
- Indirect communication → uses shared mailboxes or ports
  - multiple processes can share the same mailbox or port
  - only one process can read any given message in a mailbox
  - the OS must provide system calls to create and delete mailboxes, and to send and receive messages to/from mailboxes

Message Passing Systems: Buffering and Synchronization

- Zero capacity → Messages cannot be stored in the queue, so senders must block until receivers accept the messages

Message Passing Systems: Buffering and Synchronization

- Zero capacity → Messages cannot be stored in the queue, so senders must block until receivers accept the messages
- Bounded capacity → There is a pre-determined finite capacity in the queue, so senders must block if the queue is full, otherwise may be either blocking or non-blocking

Message Passing Systems: Buffering and Synchronization

- Zero capacity → Messages cannot be stored in the queue, so senders must block until receivers accept the messages
- Bounded capacity → There is a pre-determined finite capacity in the queue, so senders must block if the queue is full, otherwise may be either blocking or non-blocking
- Unbounded capacity → The queue has a theoretical infinite capacity, so senders are never forced to block

Summary

- Process is the unit of execution (running on a single CPU)
- OS keeps track of process-related information using an ad hoc data structure called Process Control Block (PCB)
- Process can be in one of 5 possible states: new, ready, waiting, running, or terminated
- Context switch to intertwine the execution of multiple processes
- Process communication either via message passing or shared memory