

# Systems and Networking I

Applied Computer Science and Artificial Intelligence  
2024-2025



**SAPIENZA**  
UNIVERSITÀ DI ROMA

Gabriele Tolomei

Dipartimento di Informatica  
Sapienza Università di Roma

[tolomei@di.uniroma1.it](mailto:tolomei@di.uniroma1.it)

# OS Process Management So Far...

- How the OS abstracts processes from physical memory
  - Virtual Address Space (VAS)
- In which state a process can be while it is managed by the OS
- What data structure the OS uses to keep track of each process info
  - Process Control Block (PCB)

# Outline

- Process creation
- Process termination
- Process scheduling
- Process communication

# Outline

- **Process creation**
- Process termination
- Process scheduling
- Process communication

# Process Creation

- Processes may create other processes through specific system calls

# 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**

# 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

# 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



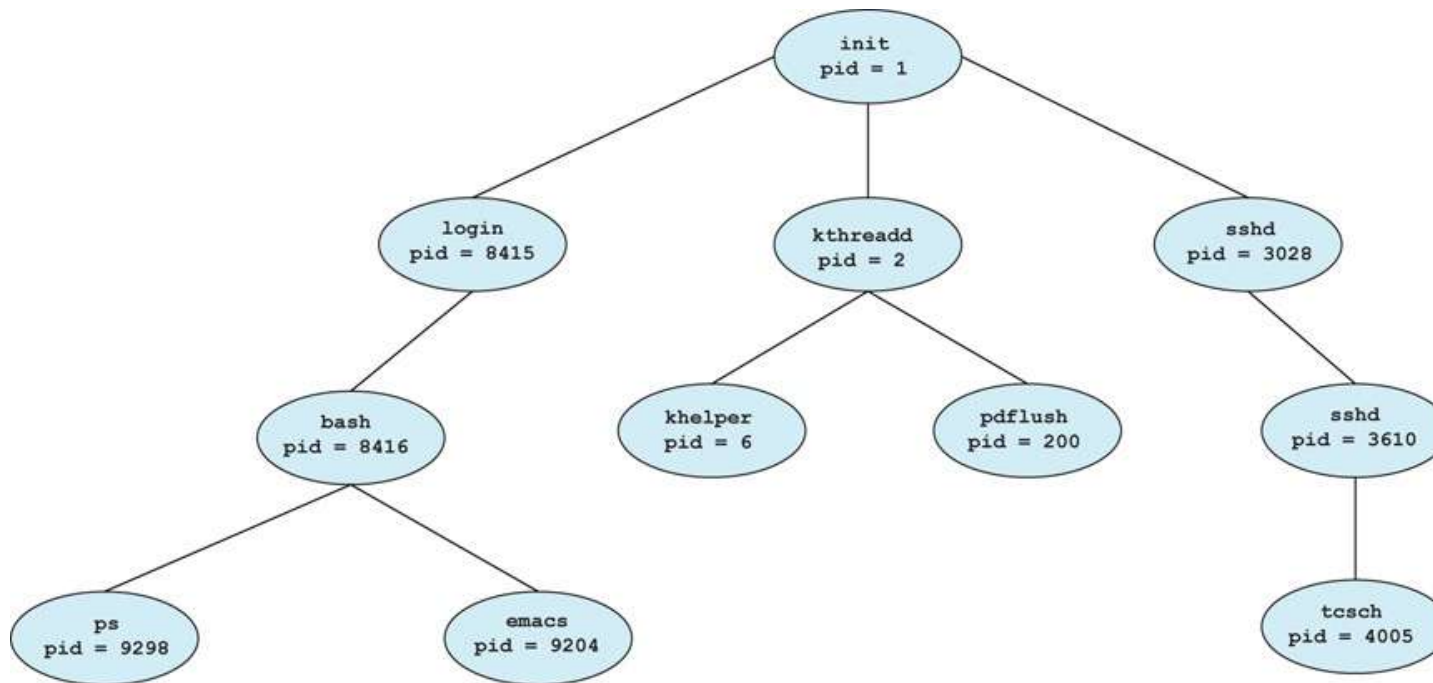
# Process Creation

- 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) and a parent ID (PPID)

# 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

- Relatively to the parent, the address space of the child process can be:
  - **Duplicated** (UNIX/Linux `fork()`)
  - **Brand New** (Windows `spawn()`)

# Process Creation: Parent vs. Child Resources

- When the child is an exact **duplicate of the parent**
  - It shares the same program and data segments in memory
  - Still, each process will have its own PCB, including program counter, registers, and PID

# Process Creation: Parent vs. Child Resources

- When the child contains an **brand new program**
  - Its address space has new code and data segments
  - UNIX systems implement this as a second step, using the **exec** system call

# Process Creation: Parent vs. Child Execution

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

# Process Creation: Parent vs. Child Execution

- **2 options** for the parent after creating the child:
  1. Wait for the child to terminate before proceeding



# Process Creation: Parent vs. Child Execution

- **2 options** for the parent after creating the child:
  1. Wait for the child to terminate before proceeding
    - The parent makes a `wait()` system call, for either a specific child or for any child
    - This causes the parent process to block until the `wait()` returns
    - Usual behavior of UNIX shell that normally waits for their children to complete before issuing a new prompt ">"

# Process Creation: Parent vs. Child Execution

- **2 options** for the parent after creating the child:
  2. Run concurrently with the child, continuing to process without waiting (and blocking)
    - This is the operation seen when a UNIX shell runs a process as a background task "&"
    - The parent may also run for a while, and then wait for the child later, which might occur in a sort of a parallel processing operation

# Process Creation: UNIX/Linux Code

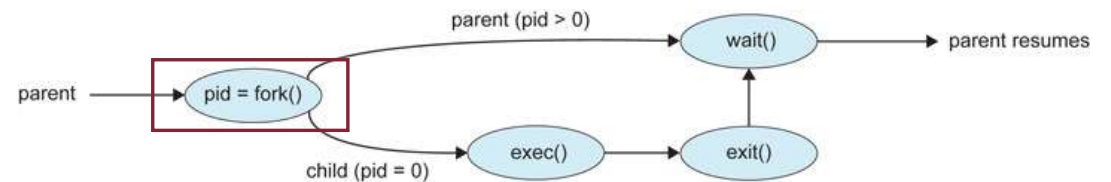
```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

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

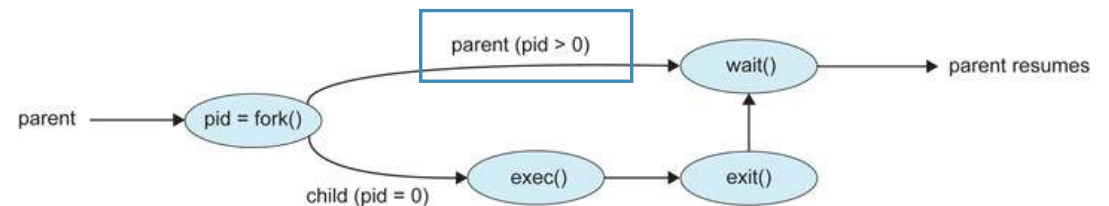
```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

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

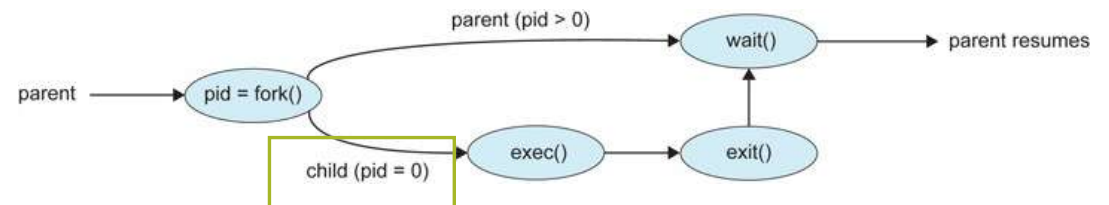
```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
    pid_t pid;

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

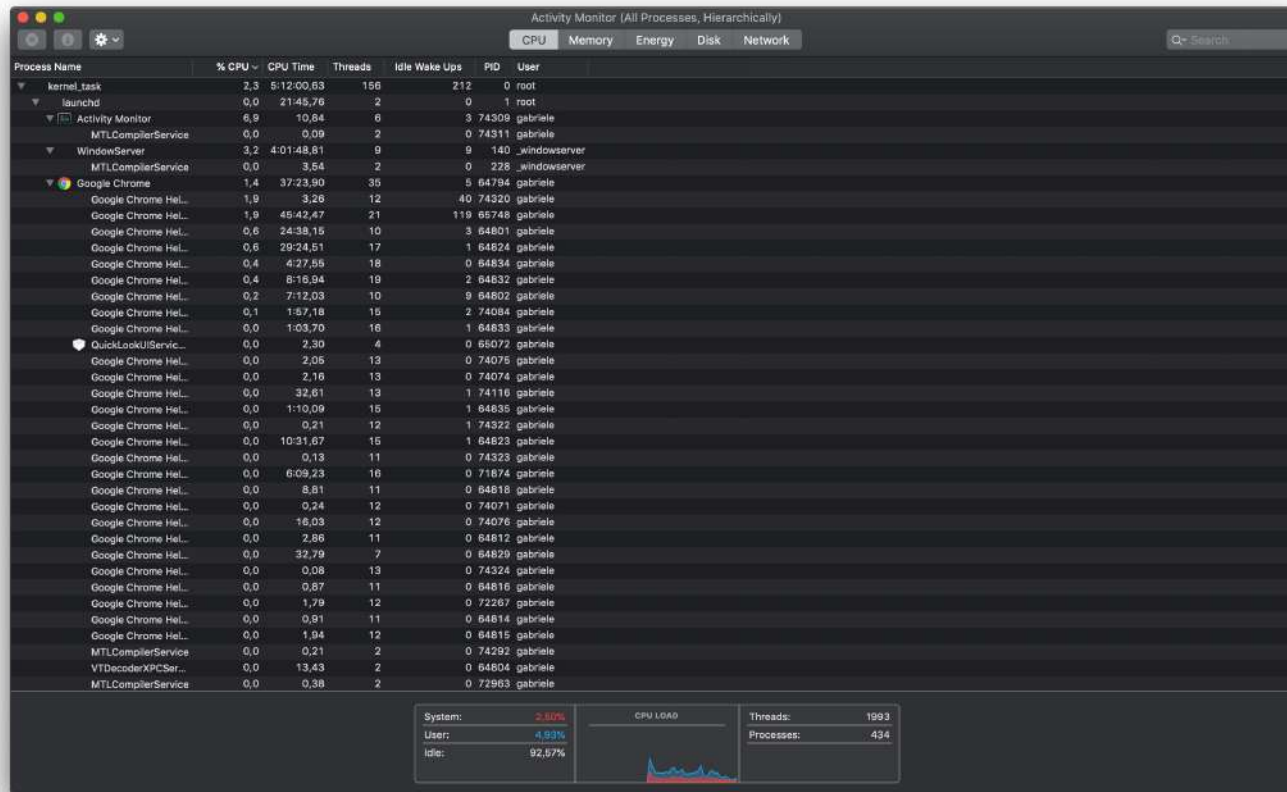
    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        exit(-1);
    }
    else if (pid == 0) { /* child process */
        execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
 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

Figure 3.10 C program forking a separate process.



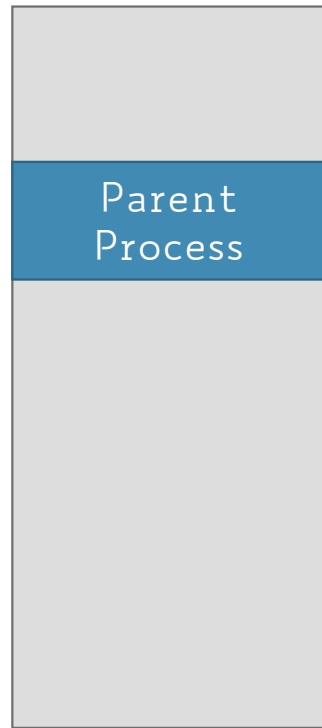
In the child process, it returns 0

# Process Creation: Activity Monitor



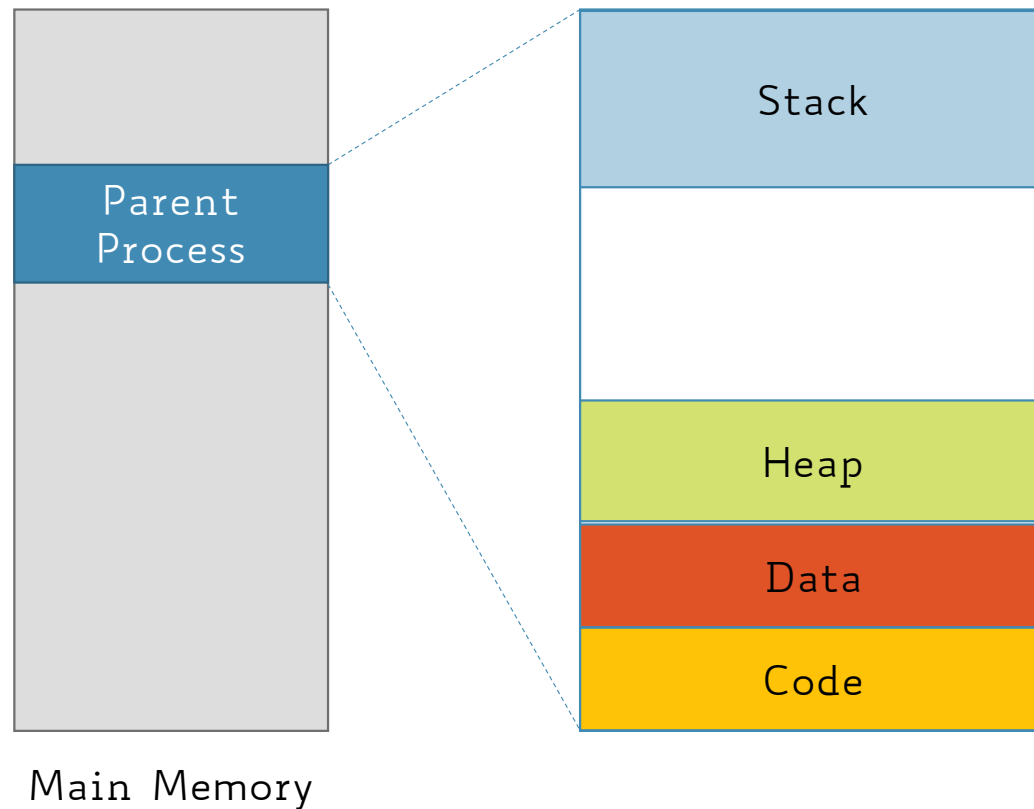
Hierarchy of  
Processes  
(i.e., process tree)

# Process Creation: Parent vs. Child Layout



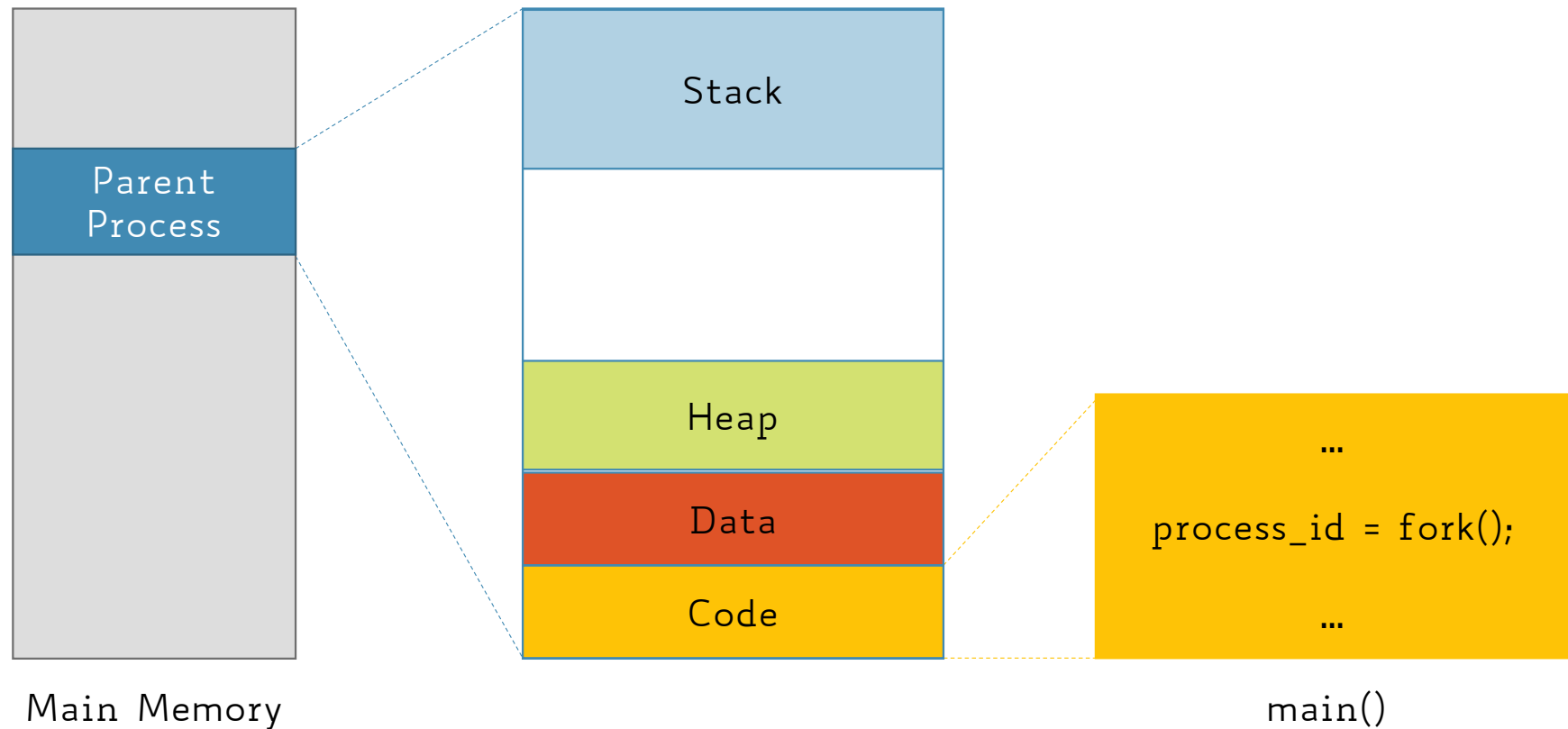
Main Memory

# Process Creation: Parent vs. Child Layout

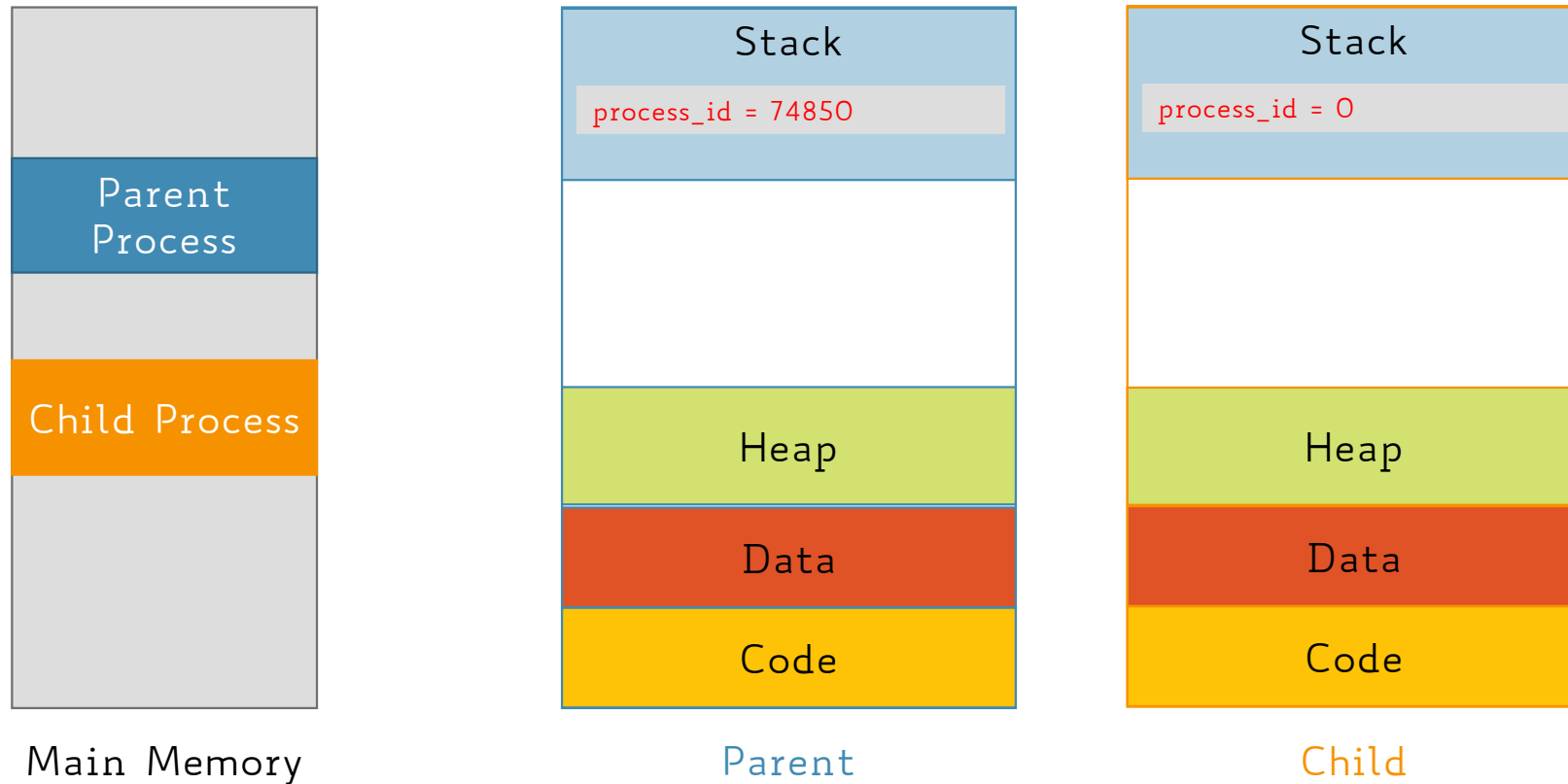




# Process Creation: Parent vs. Child Layout



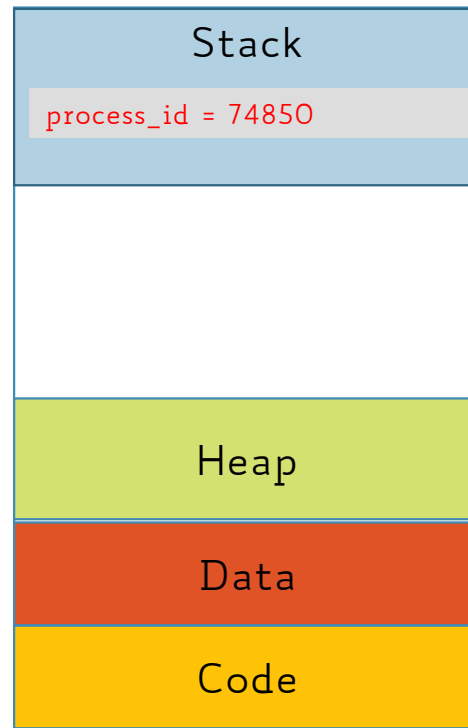
# Process Creation: Parent vs. Child Layout



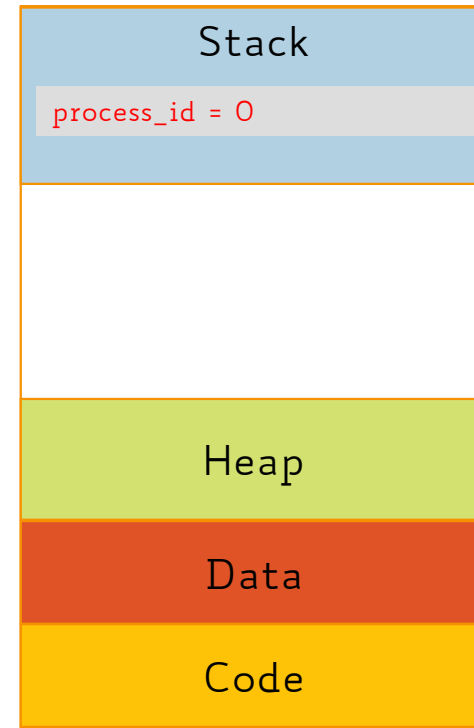
# Process Creation: Parent vs. Child Layout



Main Memory



Parent

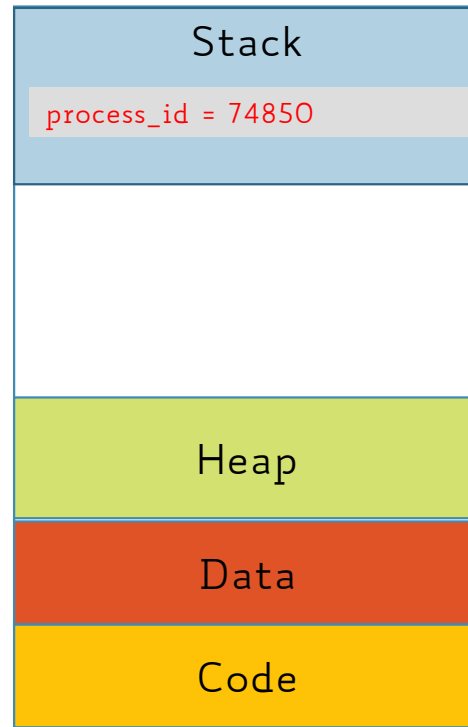


Child  
PID = 74850

# Process Creation: Parent vs. Child Layout

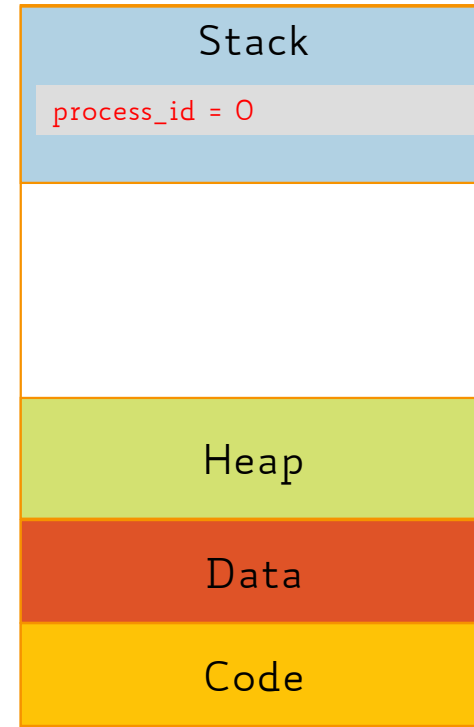


Main Memory



Parent

PID = 74849



Child

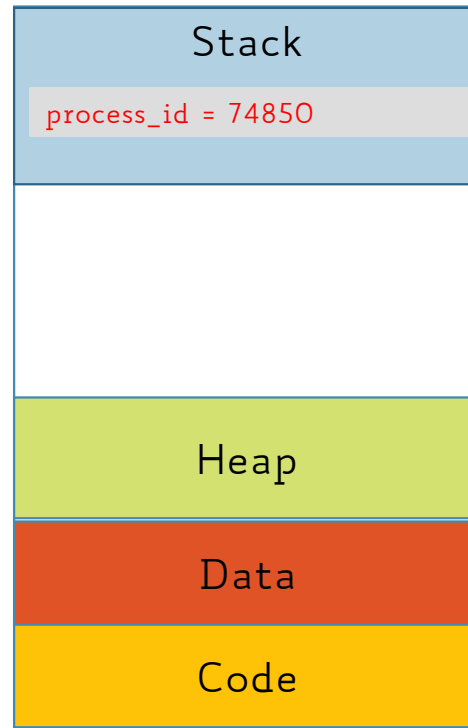
PID = 74850

parentID = 74849

# Process Creation: Parent vs. Child Layout



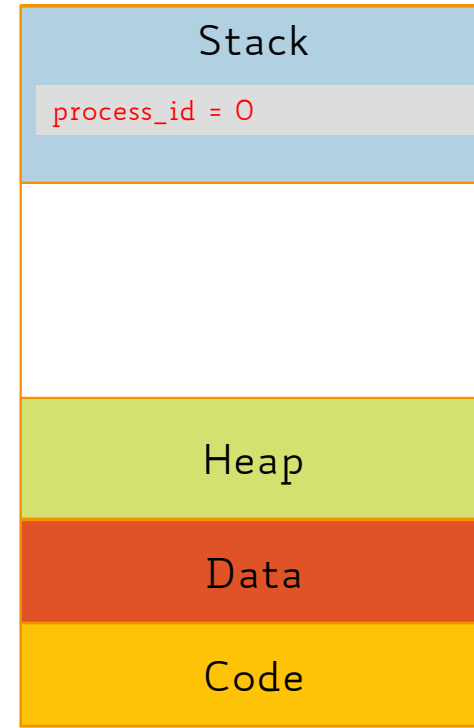
Main Memory



Parent

PID = 74849

parentID = 65784



Child

PID = 74850

parentID = 74849

# Process Creation: Code Example

```
1  #include <iostream>
2  #include <unistd.h>
3
4  using namespace std;
5
6  int main() {
7
8      cout << "Current process ID is: " << getpid() << endl;
9      cout << "\nCurrent parent's process ID is: " << getppid() << endl;
10
11     int pid;
12     pid = fork();
13     // once the fork() system call returns,
14     // both the parent and the child processes will resume from this point onward
15
16     if (pid == 0) { // child
17         cout << "\nThis is the child process with process ID = "
18             << getpid() << endl;
19         cout << "\nThis is the child process with parent's process ID = "
20             << getppid() << endl;
21     }
22     else { // parent
23         sleep(1); // to ensure the child process finishes before the parent
24
25         cout << "\nThis is the parent process with process ID = "
26             << getpid() << endl;
27         cout << "\nThis is the parent process with parent's process ID = "
28             << getppid() << endl;
29     }
30
31     return 0;
32 }
```

# Process Creation: Code Example

```
1  #include <iostream>
2  #include <unistd.h>
3
4  using namespace std;
5
6  int main() {
7
8      cout << "Current process ID is: " << getpid() << endl;
9      cout << "\nCurrent parent's process ID is: " << getppid() << endl;
10
11     int pid;
12     pid = fork();
13     // once the fork() system call returns,
14     // both the parent and the child processes will resume from this point onward
15
16     if (pid == 0) { // child
17         cout << "\nThis is the child process with process ID = "
18             << getpid() << endl;
19         cout << "\nThis is the child process with parent's process ID = "
20             << getppid() << endl;
21     }
22     else { // parent
23         sleep(1); // to ensure the child process finishes before the parent
24
25         cout << "\nThis is the parent process with process ID = "
26             << getpid() << endl;
27         cout << "\nThis is the parent process with parent's process ID = "
28             << getppid() << endl;
29     }
30
31     return 0;
32 }
```

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 started

# Process Creation: The Example of UNIX Shell

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

```
1  #include <iostream>
2  #include <unistd.h>
3  #include <sys/wait.h>
4  #include <stdio.h>
5  #include <string.h>
6
7  using namespace std;
8
9  int main() {
10
11     int current_pid = getpid();
12     cout << "Current process ID is: " << current_pid << endl;
13
14     string progStr;
15     // read the name of the program we want to start
16     getline(cin, progStr);
17     const char *prog = progStr.c_str();
18
19     int pid = fork();
20
21     if (pid == 0) { // child
22         execlp(prog, prog, 0); // load the program
23         // if prog can actually be started, we will never get to the
24         // following statement, as the child process will be replaced by prog!
25         printf("Can't load the program %s\n", prog);
26     }
27     else { // parent
28         sleep(1); // give some time to the child process to starting up
29         waitpid(pid, 0, 0); // wait for child process to terminate
30         printf("Program %s finished!\n", prog);
31     }
32     return 0;
33 }
```

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

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

# Process Creation and Execution: Example

```
1  #include <iostream>
2  #include <unistd.h>
3  #include <sys/wait.h>
4  #include <stdio.h>
5  #include <string.h>
6
7  using namespace std;
8
9  int main() {
10
11     int current_pid = getpid();
12     cout << "Current process ID is: " << current_pid << endl;
13
14     string progStr;
15     // read the name of the program we want to start
16     getline(cin, progStr);
17     const char *prog = progStr.c_str();
18
19     int pid = fork();
20
21     if (pid == 0) { // child
22         execlp(prog, prog, 0); // load the program
23         // if prog can actually be started, we will never get to the
24         // following statement, as the child process will be replaced by prog!
25         printf("Can't load the program %s\n", prog);
26     }
27     else { // parent
28         sleep(1); // give some time to the child process to starting up
29         waitpid(pid, 0, 0); // wait for child process to terminate
30         printf("Program %s finished!\n", prog);
31     }
32     return 0;
33 }
```

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

```
1  #include <iostream>
2  #include <unistd.h>
3  #include <sys/wait.h>
4  #include <stdio.h>
5  #include <string.h>
6
7  using namespace std;
8
9  int main() {
10
11     int current_pid = getpid();
12     cout << "Current process ID is: " << current_pid << endl;
13
14     string progStr;
15     // read the name of the program we want to start
16     getline(cin, progStr);
17     const char *prog = progStr.c_str();
18
19     int pid = fork();
20
21     if (pid == 0) { // child
22         execlp(prog, prog, 0); // load the program
23         // if prog can actually be started, we will never get to the
24         // following statement, as the child process will be replaced by prog!
25         printf("Can't load the program %s\n", prog);
26     }
27     else { // parent
28         sleep(1); // give some time to the child process to starting up
29         waitpid(pid, 0, 0); // wait for child process to terminate
30         printf("Program %s finished!\n", prog);
31     }
32     return 0;
33 }
```

`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

```
1  #include <iostream>
2  #include <unistd.h>
3  #include <sys/wait.h>
4  #include <stdio.h>
5  #include <string.h>
6
7  using namespace std;
8
9  int main() {
10
11     int current_pid = getpid();
12     cout << "Current process ID is: " << current_pid << endl;
13
14     string progStr;
15     // read the name of the program we want to start
16     getline(cin, progStr);
17     const char *prog = progStr.c_str();
18
19     int pid = fork();
20
21     if (pid == 0) { // child
22         execlp(prog, prog, 0); // load the program
23         // if prog can actually be started, we will never get to the
24         // following statement, as the child process will be replaced by prog!
25         printf("Can't load the program %s\n", prog);
26     }
27     else { // parent
28         sleep(1); // give some time to the child process to starting up
29         waitpid(pid, 0, 0); // wait for child process to terminate
30         printf("Program %s finished!\n", prog);
31     }
32     return 0;
33 }
```

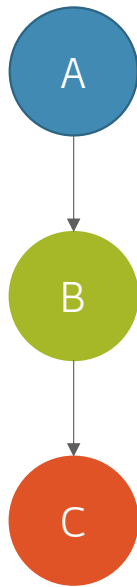
`waitpid` allows the parent  
to wait for a child  
process to finish

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



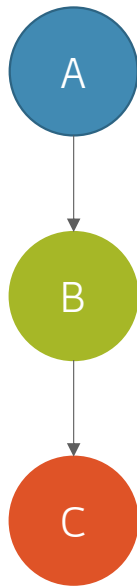
# Process Creation and Execution: Exercise

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
    ...
}
```

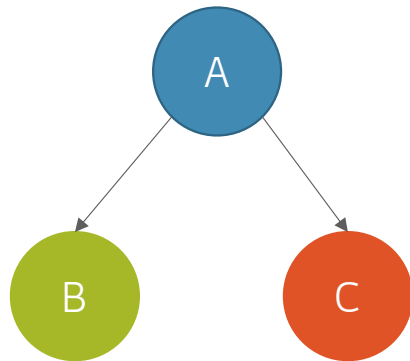
# Process Creation and Execution: Exercise

More generally, we will need  $n-1$  `fork` and `if-else`  
if we want to create a sequence of  $n$  processes



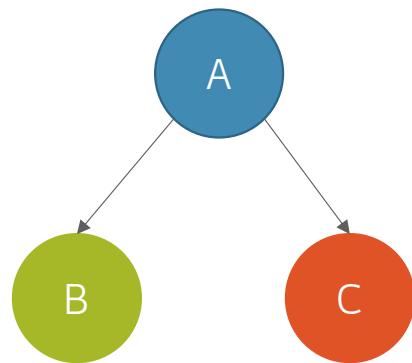
# Process Creation and Execution: Exercise

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?

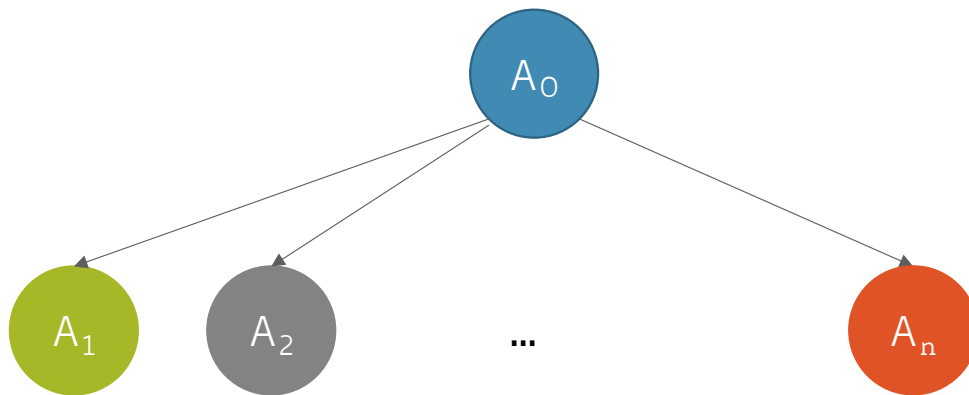


Left-to-Right (by convention)

```
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
- `execvp` → 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
- `execvp` → 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
- `execvp` → 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

# Outline

- Process creation
- **Process termination**
- Process scheduling
- Process communication

# 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

- If the parent exits, the system may or may not allow the child to continue without a parent

# Process Termination

- 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 its system resources are freed up, open files flushed and closed, etc.

# Process Termination

- When a process terminates, all 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

# Process Termination

- When a process terminates, all 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

- Processes which are trying to terminate but cannot because their parent is not waiting for them are called **zombies**

# Process Termination

- 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

# Outline

- Process creation
- Process termination
- **Process scheduling**
- Process communication



# Process Scheduling

- **2 main goals** of the process scheduling system:
  - i. keep the CPU busy at all times

# Process Scheduling

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

# Process Scheduling

- The process scheduler must meet these objectives by implementing suitable policies for swapping processes in and out of the CPU

# Process Scheduling

- 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 maintains the PCBs of all the processes in state queues

# Process State Queues

- The OS maintains the PCBs of all the processes in state queues
- There is one queue for each of the **5 states** a process can be in

# Process State Queues

- The OS maintains 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 sending/receiving data to/from a device)

# Process State Queues

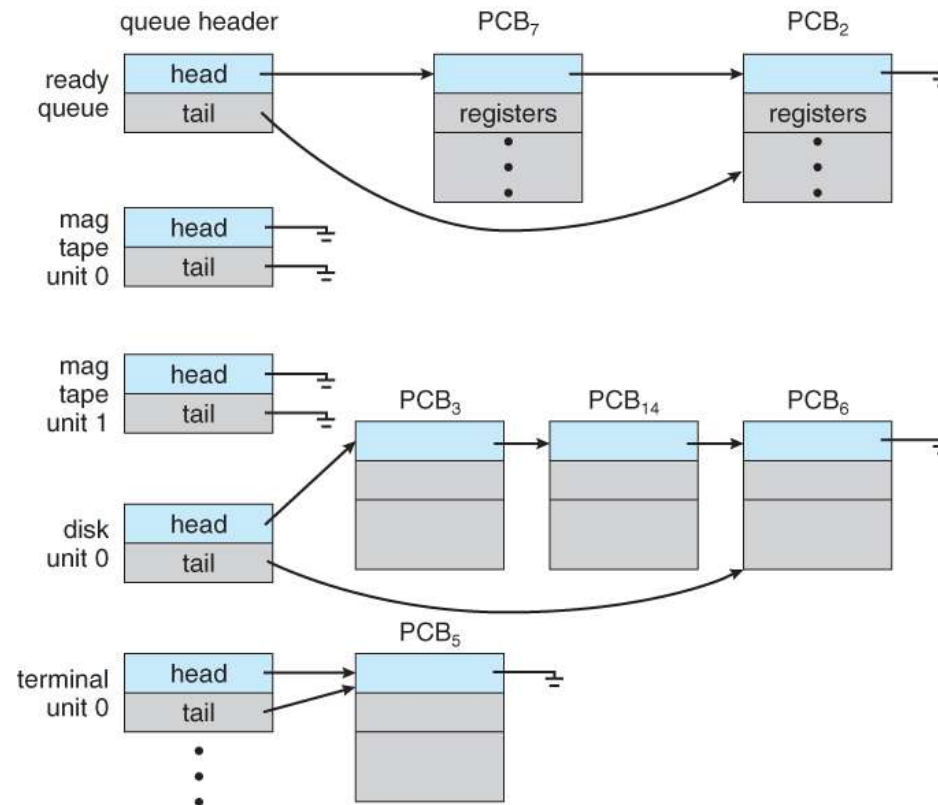
- The OS maintains the PCBs of all the processes in state queues
- There is one queue for each of the **5 states** a process can be in
- There is typically one queue for each I/O device (where processes wait for sending/receiving data to/from a device)
- When the OS changes the status of a process (e.g., from ready to running) the PCB is unlinked from one queue and moved to appropriate one



# Process State Queues

- The OS maintains 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 sending/receiving data to/from a device)
- When the OS changes the status of a process (e.g., from ready to running) the PCB is unlinked from one queue and moved to appropriate 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



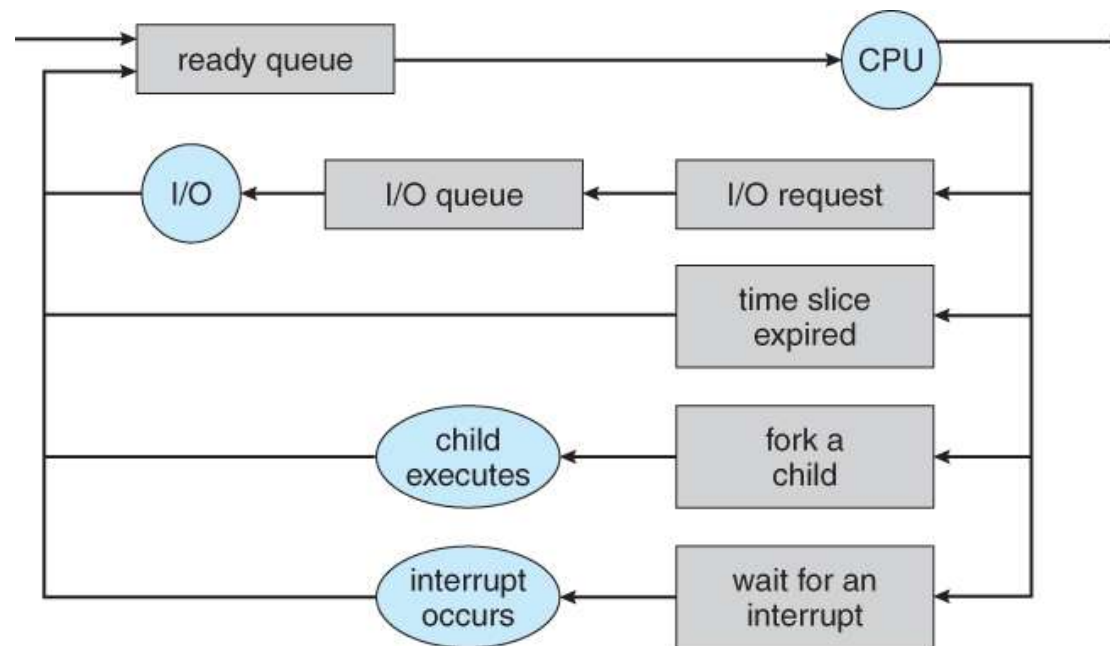
# Schedulers

- 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
- When system loads get high, a medium-term 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
- When system loads get high, a medium-term 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 a 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

# 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

# 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

# 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

- Can be easily implemented in HW through timer interrupt

# Context Switch: Time Slice

- 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

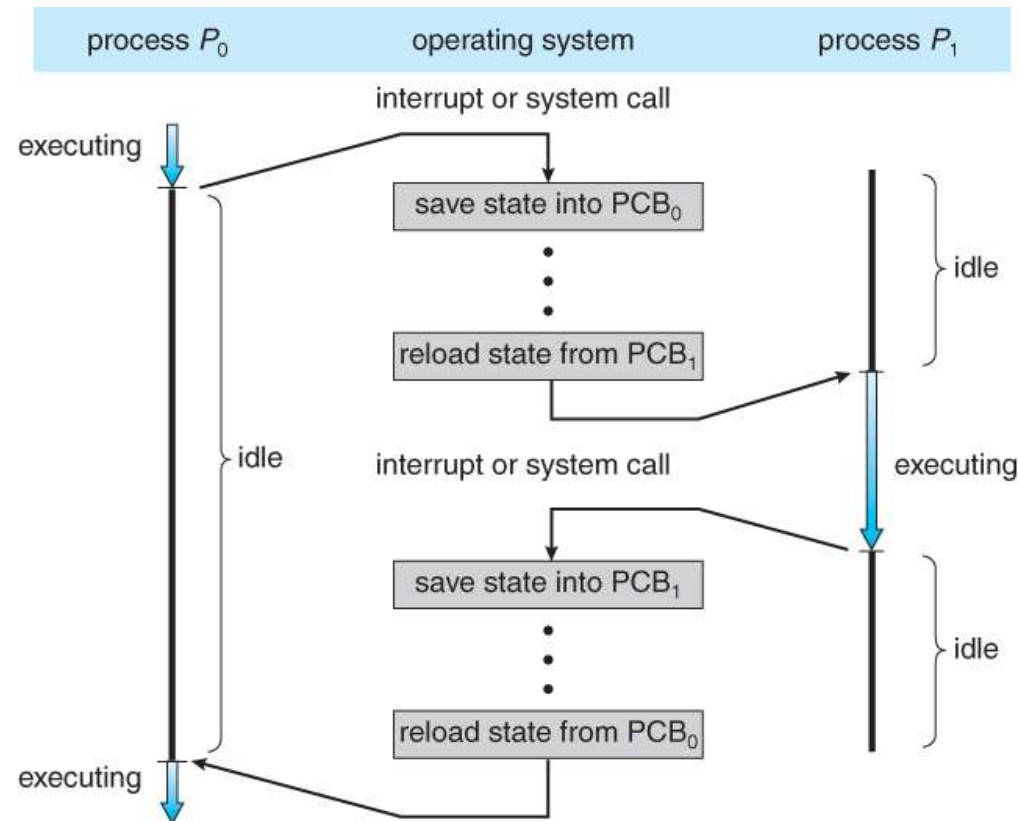
## Trade-off

- 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: A Few Numbers

- Typical values of time slice range between 10 and 100 ms
- Context switch takes around 10  $\mu$ s, so the overhead is small relative to time slice
- $10^{-2}$  to  $10^{-1}$  vs.  $10^{-8}$  seconds

# Context Switch: Example



# Outline

- Process creation
- Process termination
- Process scheduling
- **Process communication**

# 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?

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

# Cooperating Processes: Why?

- **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?

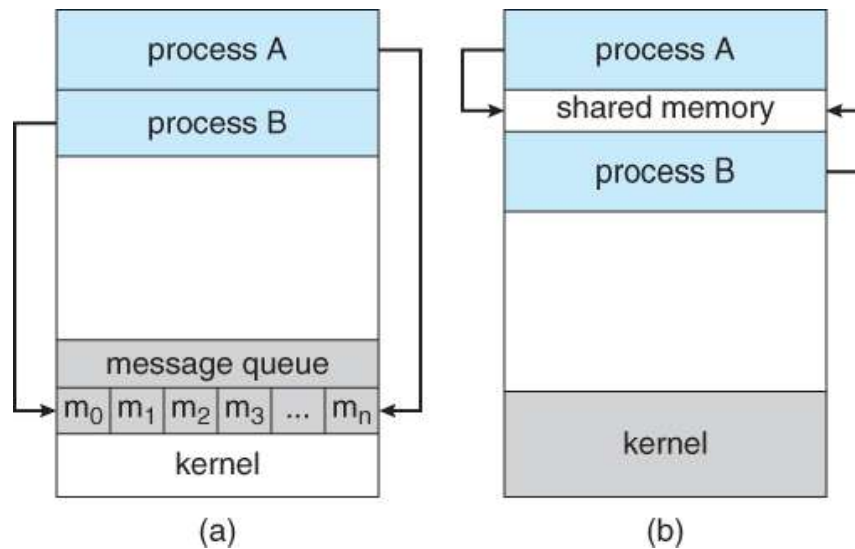
- **Information sharing** → There may be several processes which need access to the same file (e.g., pipelines)
- **Computation speedup** → A problem can be solved faster if it can be broken down into sub-tasks to be solved simultaneously
- **Modularity** → The most efficient architecture may be to break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

# Cooperating Processes: Why?

- **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

- **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 are created programmatically via system calls (e.g., `fork/exec`)

# Summary

- Process are created programmatically via system calls (e.g., `fork/exec`)
- Scheduling policies to maximize CPU utilization for process execution

# Summary

- Process are created programmatically via system calls (e.g., `fork/exec`)
- Scheduling policies to maximize CPU utilization for process execution
- **Context switch** to intertwine the execution of multiple processes

# Summary

- Process are created programmatically via system calls (e.g., `fork/exec`)
- Scheduling policies to maximize CPU utilization for process execution
- **Context switch** to intertwine the execution of multiple processes
- Process communication either via **shared memory** or **message passing**