Operating Systems
Process management

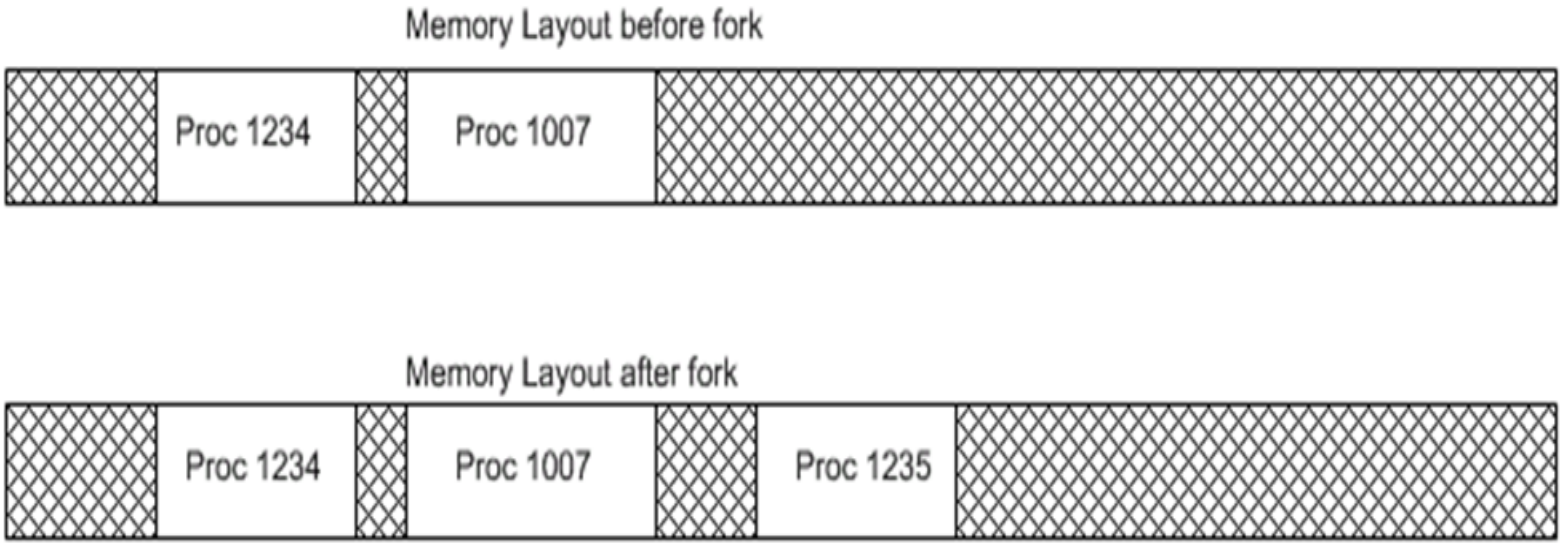
fork system call

- Function prototype:

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

```
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>
extern int errno;
int main()
{
    pid_t pid;
    pid = fork();
    if (pid == 0)
       printf("I'm the child\n");
    else if (pid > 0) {
       printf("I'm the parent, ");
       printf("child pid is %d\n",pid);
   }
    else { /* pid < 0 */
       perror("Error forking");
       fprintf(stderr,"errno is %d\n",errno);
    }
   return 0;
}
```

Memory layout



When will a call to fork fail?

A call to fork() can fail for several reasons, most commonly due to resource exhaustion or hitting system limits. Specifically, it might fail if the system has reached its limit on the number of processes, if there is not enough memory or swap space, or if a user's process limit has been reached. Additionally, certain system configurations or security mechanisms, like those in macOS's System Integrity Protection, can also cause fork() to fail.

Same for parent and child processes

- The text segments (code segments)
- The values of all variables (except the value returned from fork())
- The environment
- The process priority
- The controlling terminal
- The current working directory
- Open file descriptors

Different for parent and child processes

- The process id
- The parent process id
- Data on resource allocation

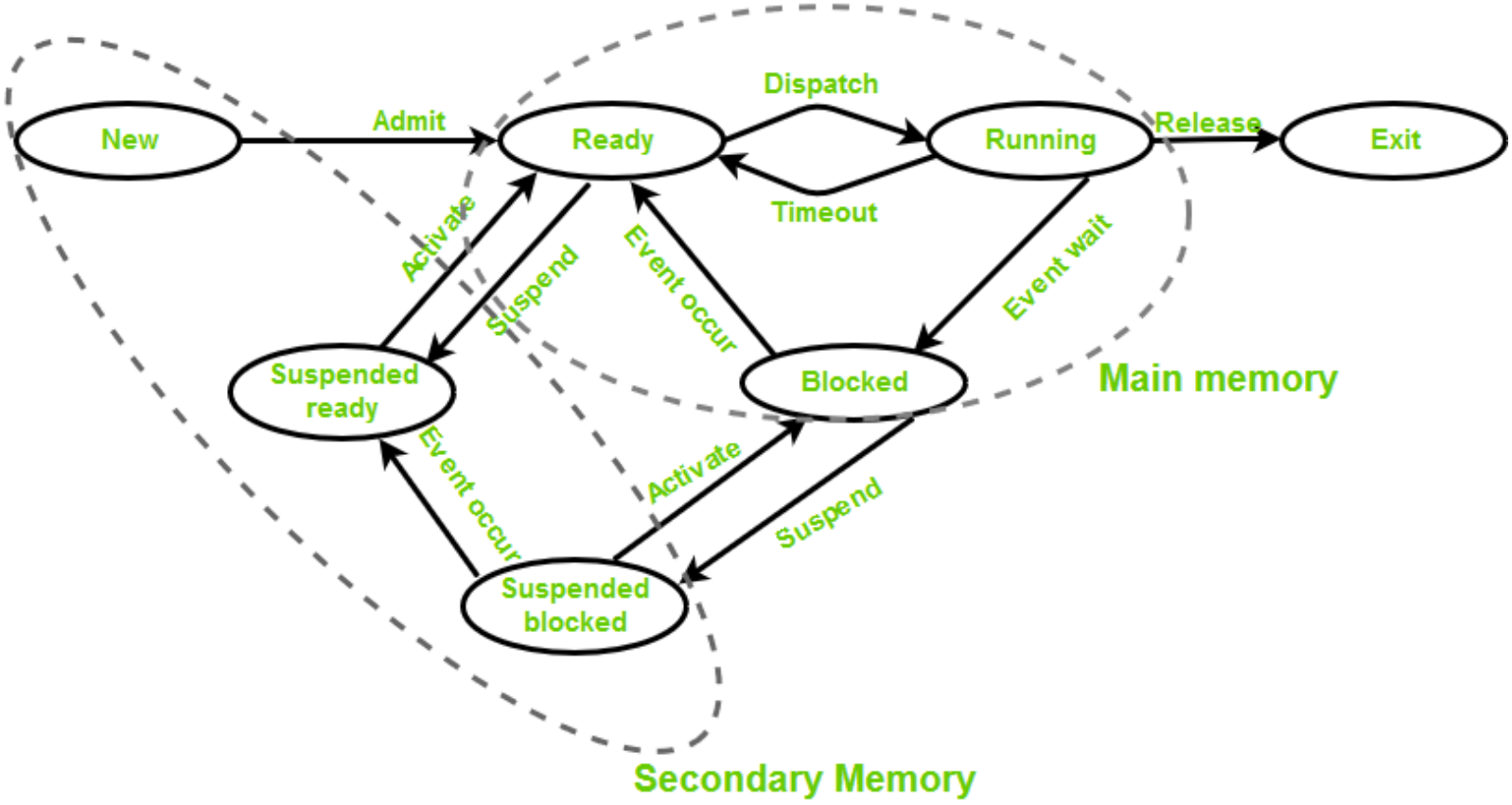
(For example, total run time is set to zero in the child, and process start time for the child is set to the current time)

every process except the init process (init has pid 0 and is the first process created at boot time. It runs until the system shuts down.) has a parent process so there is a tree of processes with init as the root.

What does a fork do?

- Reserve swap space for the child's data and stack
- allocate a new pid and kernel proc structure
- initialize the kernel proc structure. Some fields (i.e. user id, group id, signal masks) are copied from the parent, some set to zero (i.e. cpu usage), others such as ppid point to child specific values
- allocate address translation maps for the child
- add the child to the set of processes sharing the text region of the program that the parent is executing
- duplicate the parent's data and stack regions
- acquire references to shared resources inherited by the child such as open files
- initialize the hardware context by copying the parent's registers
- make the child runnable and place it on the scheduler queue
- arrange for the child to return from fork with a value of zero
- return the pid of the child to the parent

Process state



wait sys call

- Whether the parent or the child will be run first is undetermined; the term for this is a *race condition*. Thus there are two possible outputs for the following program.

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
extern int errno;
int main()
{
   pid_t pid, retval;
   int status;

   pid = fork();
   if (pid == 0)
      printf("I'm the child\n");
   else if (pid > 0) {
      retval = wait(&status);
      printf("I'm the parent, ");
      printf("the child %d has died\n",retval);
  }
   else { /* pid < 0 */
      perror("Error forking");
      fprintf(stderr,"errno is %d\n",errno);
   }
  return 0;
}
```

The exec family of system calls

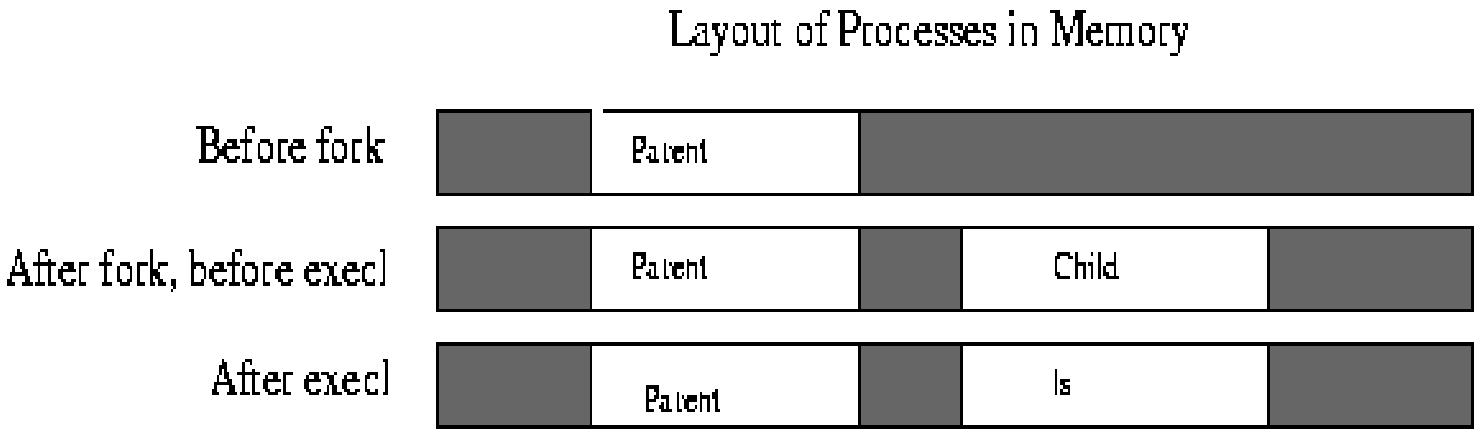
- The fork call is usually used with another system call, exec, which overwrites the entire process space with a completely new process image
- Execution of the new image starts at the beginning
- Exec is actually a family of six system calls, the simplest of which is execl

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <wait.h>
extern int errno;
int main()
{
   pid_t p;
   p=fork();
   if (p == 0)  { /* child */
        execl("/bin/ls", "ls", "-l", NULL);
        perror("Exec failed");
   }
   else if (p > 0) {
      wait(NULL);
      printf("Child is done\n");
   }
   else {
      perror("Could not fork");
   }
   return 0;
}
```

Layout of processes in memory

innovate achieve lead

```
1.  ....
2.  pid = fork();
3.  if (pid == 0) {
4.       execv("/bin/ls",...
5.       ...
6.  ]
7.  else if (pid > 0) {
8.     /* more code for parent */
```



- `int execv(const char *path, char *const argv[])` This call is the same as execl except that it takes only two arguments, the second being an argument vector.
  Here is a short program which demonstrates the use of execv
- `int execle(const char *path, const char *arg0, ..., const char *argn, char * /*NULL*/, char *const envp[])` Like `execl`, this call takes a variable number of arguments, but its final argument is a vector which represents the new environment.

  By default, the environment of the process which is exec'ed is the same as that of the parent, but this allows the user to change the environment.
  Here is a short program which demonstrates the use of execle

- `int execve(const char *path, char *const argv[], char *const envp[])` this is the same as `execv` except that it passes the environment vector as a third argument.

`int execlp(const char *file, const char *arg0, ..., const char *argn, char * /*NULL*/)` This differs from the above calls in that its first argument is just a filename rather than a path, and the call searches the PATH