13 mark

P1 word   P2 Chrome

P1 ⇄ P2

word   spell check

Interprocess Communication

- Processes within a system may be ***independent*** or ***cooperating***
  - ***Independent*** process cannot affect or be affected by the execution of another process
  - ***Cooperating*** process can affect or be affected by the execution of another process

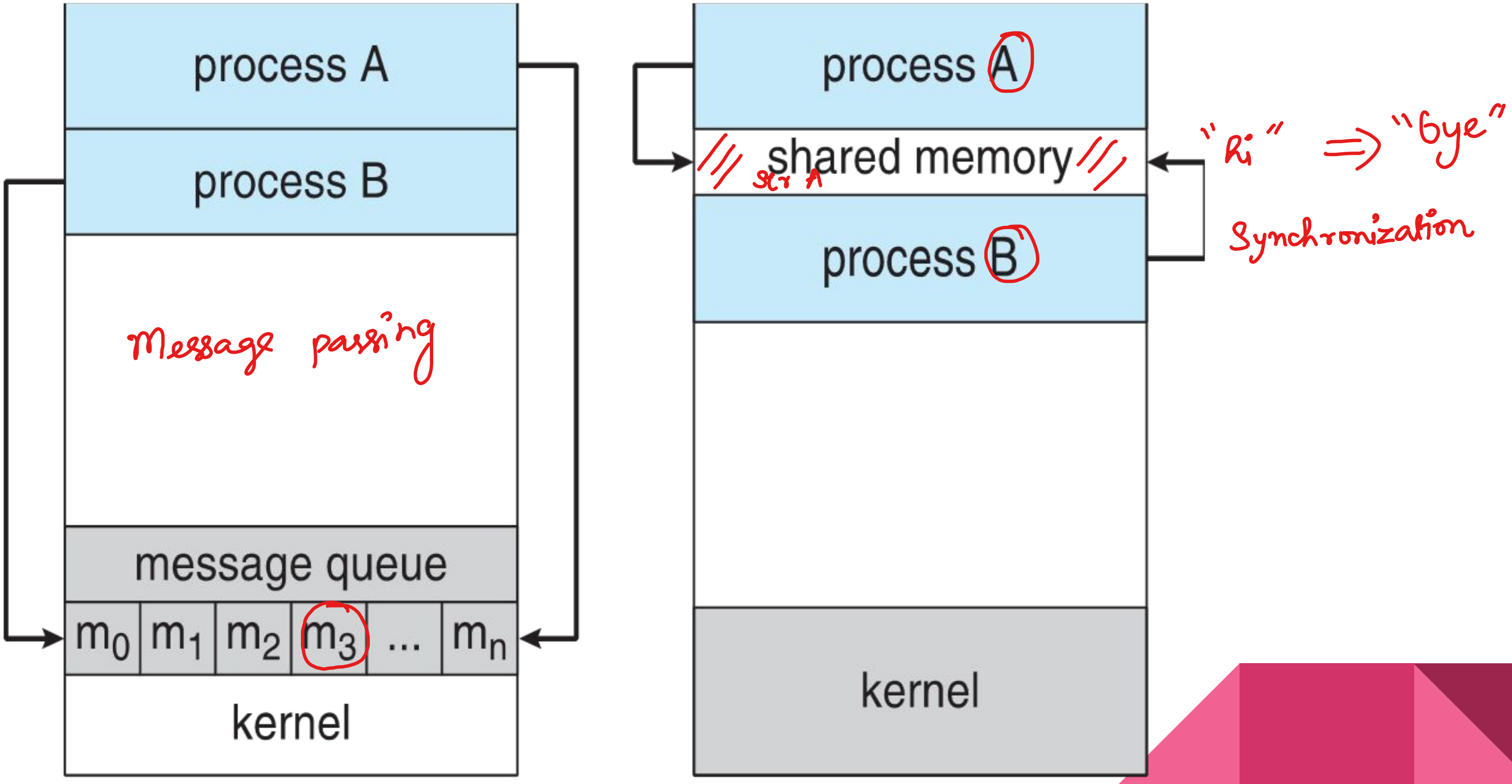
P3   P4

T1   T1

500GB

P1

- Reasons for cooperating processes:
  - 1 Information sharing
  - 2 Computation speedup
  - 3 Modularity
  - 4 Convenience → Users
- Cooperating processes need **interprocess communication** (**IPC**)
- Two models of IPC
  - **Shared memory** ✓
  - **Message passing** ✓


process A
process B
Message passing
message queue
m_0 m_1 m_2 m_3 ... m_n
kernel
process A
shared memory
str A
process B
kernel
"hi" ⇒ "bye"
Synchronization

(a)

(b)

Shared Memory ✓

XOS

- An area of memory shared among the processes that wish to communicate ✓
- The communication is under the control of the users processes not the operating system.
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.

Eg.

Producer-Consumer Problem

$P_1 \rightarrow 2, 3, 4, 5$

$P_2 \rightarrow$ consume

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
  - **unbounded-buffer** places no practical limit on the size of the buffer
  - **bounded-buffer** assumes that there is a fixed buffer size

Buffer $\Rightarrow$ Temp storage

Bounded-Buffer – Shared-Memory Solution

Shared data

```
#define BUFFER_SIZE 10
typedef struct {
   . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

P:-

```
item next_produced;
while (true) {
        /* produce an item in next produced */
        while (((in + 1) % BUFFER_SIZE) == out)
                ; /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
}
```

C:-

```
item next_consumed;
while (true) {
        while (in == out)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;

        /* consume the item in next consumed */
```

Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  1. **`send`**(*message*)
  2. **`receive`**(*message*)
- The *message* size is either fixed or variable

- If processes *P* and *Q* wish to communicate, they need to:
  - Establish a ***communication link*** between them
  - Exchange messages via send/receive
- Implementation issues:
  - How are links established? ✓
  - Can a link be associated with more than two processes? ✓
  - How many links can there be between every pair of communicating processes? ✓
  - What is the capacity of a link? ✓
  - Is the size of a message that the link can accommodate fixed or variable? ✓
  - Is a link unidirectional or bi-directional? ✓
- Logical:
  - Direct or indirect
  - Synchronous or asynchronous
  - Automatic or explicit buffering

$P \underset{\text{link}}{=\!=} Q$

$P_1 \overset{\longrightarrow}{=} Q_1$

$P_2 \longleftrightarrow Q_2$

SMS

Arun → send(Senthil, "hi")

Senthil → receive(Arun, 'hi')

Direct Communication

$Q \xrightarrow{L_1} P$

send(P, msg)

- Processes must name each other explicitly:
  - **`send`** (*P*, *message*) – send a message to process P
  - **`receive`**(*Q*, *message*) – receive a message from process Q
- Properties of communication link
  - 1) Links are established automatically
  - 2) A link is associated with exactly one pair of communicating processes L_1 Q→P
  - 3) Between each pair there exists exactly one link Q → P #link = 1
  - 4) The link may be unidirectional, but is usually bi-directional

$Q \rightarrow \boxed{\text{msg}} \rightarrow P$

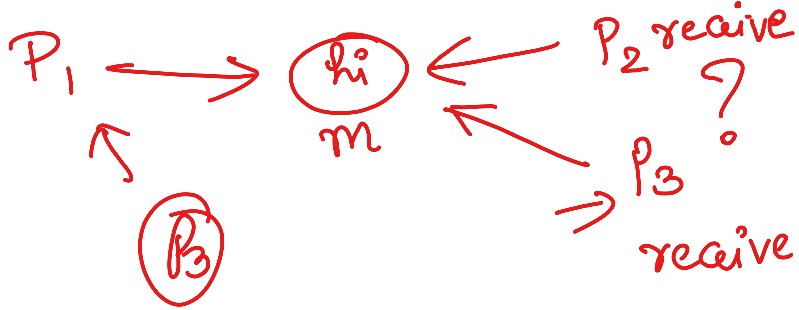
mailbox
123

Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports)
  - Each mailbox has a unique id ✓
  - Processes can communicate only if they share a mailbox

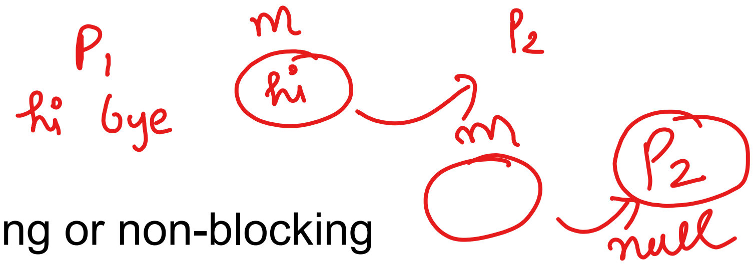
Q P

- Properties of communication link
  1) Link established only if processes share a common mailbox
  2) A link may be associated with many processes
  3) Each pair of processes may share several communication links ✓
  4) Link may be unidirectional or bi-directional



- Mailbox sharing
  - P_1, P_2, and P_3 share mailbox A
  - P_1, sends; P_2 and P_3 receive
  - Who gets the message?
- Solutions
  - 1 Allow a link to be associated with at most two processes
  - 2 Allow only one process at a time to execute a receive operation ✓
  - 3 Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

Synchronization



- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
    - **Blocking send** -- the sender is blocked until the message is received
    - **Blocking receive** -- the receiver is blocked until a message is available
- **Non-blocking** is considered **asynchronous**
    - **Non-blocking send** -- the sender sends the message and continue
    - **Non-blocking receive** -- the receiver receives:
        - A valid message, or
        - Null message

Buffering

P_1 P_2

- Queue of messages attached to the link.
- implemented in one of three ways

1. Zero capacity – no messages are queued on a link.

   Sender must wait for receiver (rendezvous)

2. Bounded capacity – finite length of *n* messages

   Sender must wait if link full

3. Unbounded capacity – infinite length

   Sender never waits

IPC What? Need?	
Need - Info sharing, Computation speedup, Modularity , Convenience 4	
Diagram - Shared vs Message Passing	

Shared Memory ✓	**Message Passing** ✓
Definition ✓	Definition ✓
Types of buffers - Unbounded & bounded ✓	Direct vs Indirect - Syntax & link ✓
Eg Producer Consumer - Code & explanation ✓	Synch vs Asynch - Blocking vs Non-blocking ✓
	Buffering - Zero, Bounded & Unbounded ✓

Operations on Processes

Creation
Termination

pid

parent P1 (word)

fork()

return child's pid

returns 0

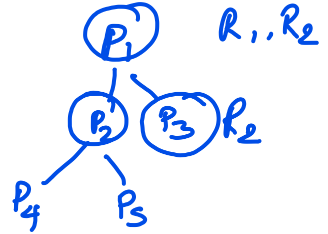
child P2 (sub-task)
spell check

P1

P2 P3 P4 P5

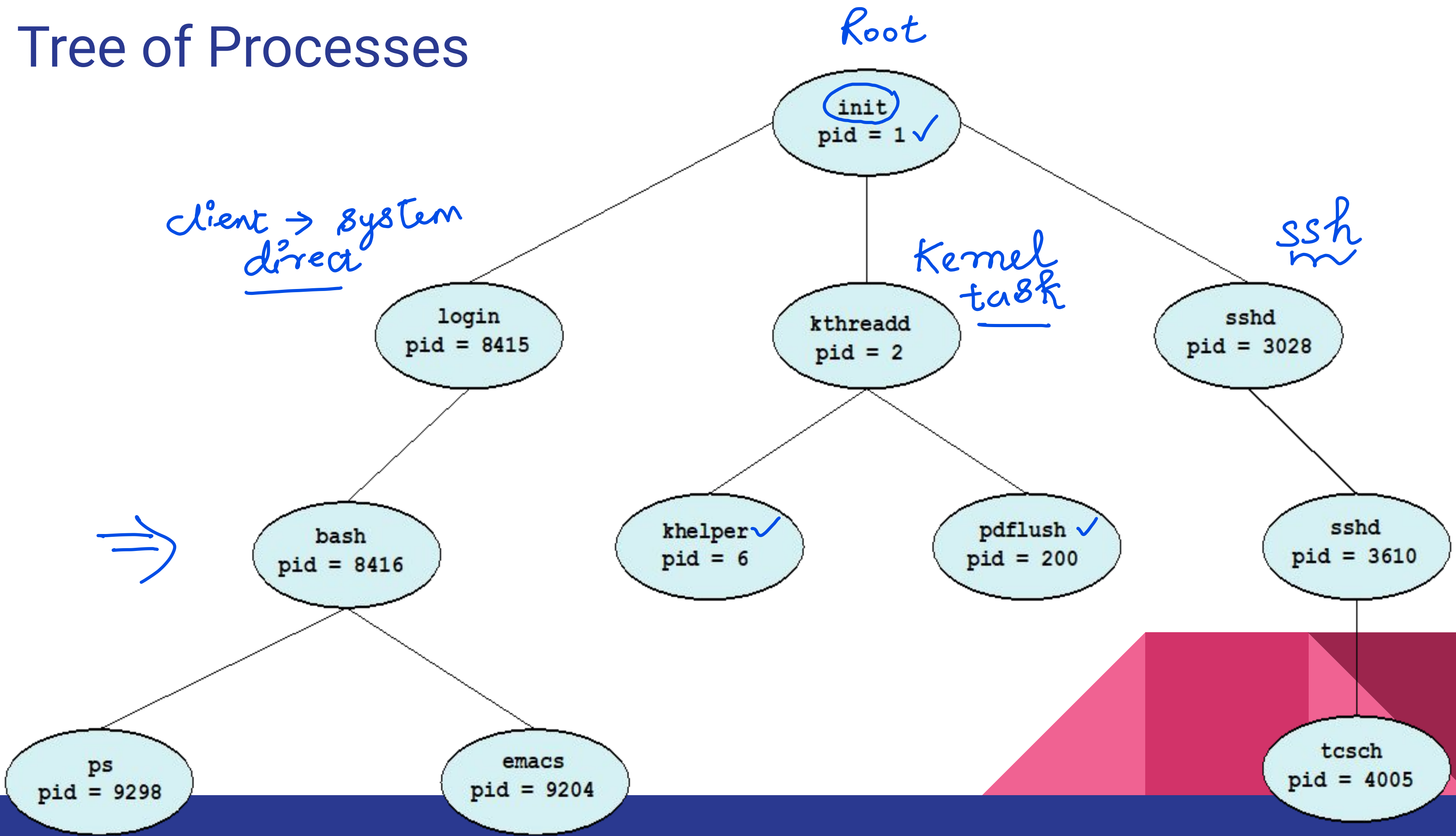
1 Process Creation

Resources
m/y, files, i/o



- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes
- Generally, process identified and managed via a **process identifier** (**pid**)
- Resource sharing options
  - 1 Parent and children share all resources ✓
  - 2 Children share subset of parent's resources
  - 3 Parent and child share no resources
- Execution options
  - 1 Parent and children execute concurrently ✓
  - 2 Parent waits until children terminate ✓

Tree of Processes



P1 (swap)

P2

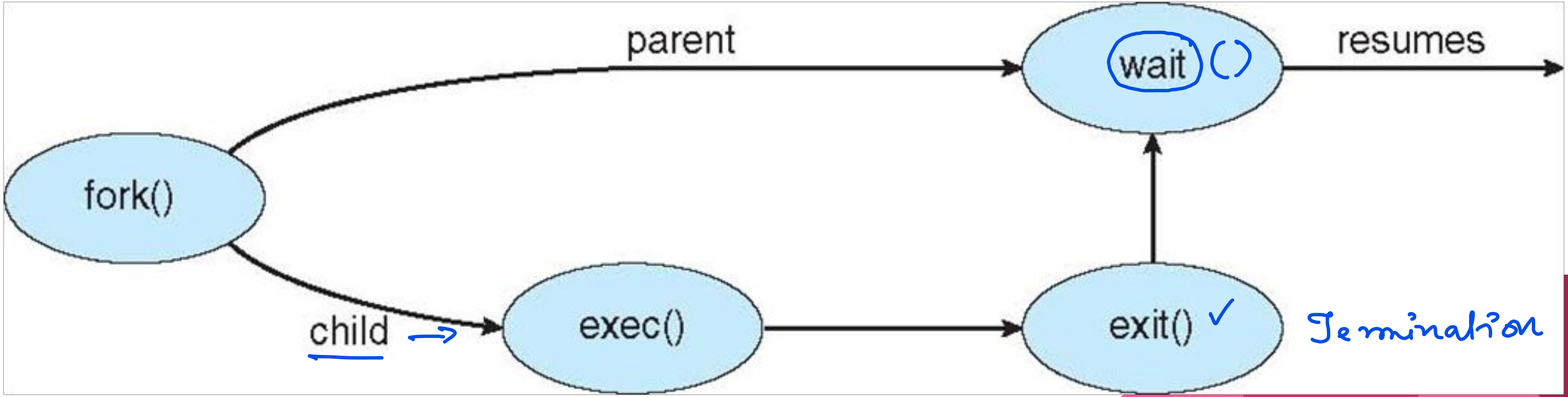
1 swap 2 multiplication

m/y

~~swap~~

- Address space
  - 1 Child duplicate of parent ✓
  - 2 Child has a program loaded into it ⇐
- UNIX examples
  - 1 **fork()** system call creates new process
  - 2 **exec()** system call used after a **fork()** to replace the process' memory space with a new program

exec()



```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");
       return 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");
    }

    return 0;
}
```

pid>0

1

fork()

pid = 0 ⇒ child process

pid > 0 ⇒ parent

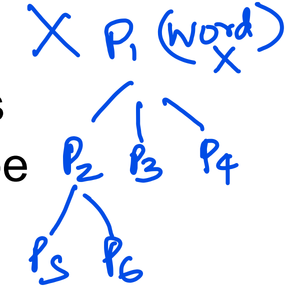
pid < 0 ⇒ failure

Process Termination

Normal
AGnormal

P_1 word
P_2 R_1, R_2, R_3
spelle

- Process executes last statement and then asks the operating system to delete it using the **`exit()`** system call.
  - Returns status data from child to parent (via **`wait()`**)
  - Process' resources are deallocated by operating system
- (2) Parent may terminate the execution of children processes using the **`abort()`** system call. Some reasons for doing so:
  - 1) Child has exceeded allocated resources
  - 2) Task assigned to child is no longer required
  - 3) The parent is exiting and the operating systems does not allow a child to continue if its parent terminates



- Some operating systems do not allow child to exists if its parent has terminated. If a process terminates, then all its children must also be terminated.
    - **cascading termination.** All children, grandchildren, etc. are terminated.
    - The termination is initiated by the operating system. ✓
- The parent process may wait for termination of a child process by using the **`wait()`** system call. The call returns status information and the pid of the terminated process
- **`pid = wait(&status);`**
- If no parent waiting (did not invoke **`wait()`**) process is a **zombie** ✓
- If parent terminated without invoking **`wait`**, process is an **orphan**

P1 — P2 exit()

P1 ~~wait()~~ — P2 exit()

init() → periodically calls wait()